Scalable Computational Science

Carmine Spagnuolo

Università degli Studi di Salerno

Dipartimento di Informatica

Dottorato di Ricerca in Informatica e Ingegneria dell'Informazione

DOCTOR OF PHILOSOPHY

Computer Science

Parallel and Distributed Computing

Scalable Computational Science

Carmine Spagnuolo

Supervisor Prof. Vittorio Scarano

Supervisor Dott. Gennaro Cordasco

2017

Carmine Spagnuolo
Scalable Computational Science
Parallel and Distributed Computing, Supervisors: Prof. Vittorio Scarano and Dott. Gennaro Cordasco

Università degli Studi di Salerno

Dottorato di Ricerca in Informatica e Ingegneria dell'Informazione
Dipartimento di Informatica
Via Giovanni Paolo II, 132
84084 , Salerno

Abstract

Computational science also know as scientific computing is a rapidly growing novel field that uses advanced computing in order to solve complex problems. This new discipline combines technologies, modern computational methods and simulations to address problems too complex to be reliably predicted only by theory and too dangerous or expensive to be reproduced in laboratories.

Successes in computational science over the past twenty years have caused demand of supercomputing, to improve the performance of the solutions and to allow the growth of the models, in terms of sizes and quality. From a computer scientist's perspective, it is natural to think to distribute the computation required to study a complex systems among multiple machines: it is well known that the speed of single-processor computers is reaching some physical limits. For these reasons, parallel and distributed computing has become the dominant paradigm for computational scientists who need the latest development on computing resources in order to solve their problems and the "Scalability" has been recognized as the central challenge in this science.

In this dissertation the design and implementation of *Frameworks, Parallel Languages* and *Architectures*, which enable to improve the state of the art on *Scalable Computational Science*, are discussed. The main features of this contribution are:

- The proposal of D-MASON, a distributed version of MASON, a well-known and popular Java toolkit for writing and running Agent-Based Simulations (ABSs). D-MASON introduces a framework level parallelization so that scientists that use the framework (e.g., a domain expert with limited knowledge of distributed programming) could be only minimally aware of such distribution. The main features of D-MASON are:
  - Partitioning and balancing. D-MASON provides several mechanism that enable an efficient distribution of the simulation – either space-base or network-based – work on multiple, even heterogeneous, machines.
  - Scalable communication. D-MASON communication is based on a Publish/Subscribe paradigm. Two communication strategy –centralized (using Java Message Services) and decentralized (using Message Passing

Interface) – are provided in order to meet both the flexibility and scalability requirements.
  - Memory consistency. D-MASON provides a memory consistency mechanism at framework level which enables the modeler to design the simulation without any specific knowledge about the underlying distributed memory environment.
  - Support diverse computing environments. D-MASON was initially conceived to harness the amount of unused computing power available in common, even heterogeneous, installations like educational laboratories. Thereafter the focus moved to dedicated homogenous installations, such as massively parallel machines or supercomputing centers. Eventually, D-MASON has been extended in order to provide a SIMulation-as-a-Service (SIMaaS) infrastructure that simplifies the process of setting up and running distributed simulations in a Cloud Computing environment.
- The proposal of an architecture, which enable to invoke code supported by a Java Virtual Machine (JVM) from code written in C language. Swift/T is a parallel programming language that enables to compose and execute a series of computational or data manipulation steps in a scientific application. Swift/T enables to easily execute code written in other languages, as C, C++, Fortran, Python, R, Tcl, Julia, Qt Script. The proposed architecture has been integrated in Swift/T (since the version 1.0) enabling the support for others kinds of interpreted languages.
- The proposal of two tools, which exploit the computing power of parallel systems to improve the effectiveness and the efficiency of Simulation Optimization strategies. Simulations Optimization (SO) is used to refer to the techniques studied for ascertaining the parameters of a complex model that minimize (or maximize) given criteria (one or many), which can only be computed by performing a simulation run. Due to the the high dimensionality of the search space, the heterogeneity of parameters, the irregular shape and the stochastic nature of the objective evaluation function, the tuning of such systems is extremely demanding from the computational point of view. The proposed tools are SOF (Simulation Optimization and exploration Framework on the cloud) and EMEWS (Extreme-scale Model Exploration With Swift/T) which focus respectively on Cloud Environment and HPC systems.
- The proposal of an open-source, extensible, architecture for the visualization of data in HTML pages, exploiting a distributed web computing. Following the Edge-centric Computing paradigm, the data visualization is performed edge side ensuring data trustiness, privacy, scalability and dynamic data loading. The architecture has been exploited in the Social Platform for Open Data (SPOD).

Acknowledgement

> *La doccia è milanese perché ci si lava meglio, consuma meno acqua e fa perdere meno tempo. Il bagno invece è napoletano. E' un incontro con i pensieri, un appuntamento con la fantasia.*
>
> **— Così parlò Bellavista**

Il mio primo ringraziamento va al professore Vittorio Scarano, mio supervisore. Voglio rimarcare la dedizione e la cura con cui mi ha seguito passo dopo passo in questi anni. Un'attenzione costante che ha favorito la mia formazione e mi ha permesso di avere nuove ed emozionati opportunità di crescita. Ritenerlo solo un supervisore sarebbe riduttivo e non potrò mai ringraziarlo abbastanza. Ringrazio, altresì, con massima stima e affetto il dottore Gennaro Cordasco, mentore e amico insostituibile (*Gennà!!*). Sempre pronto ad ascoltarmi mi ha spronato ad avere un approccio alla ricerca e una visione della realtà mai banale. Ringrazio ancora tutti i membri ed ex-membri del laboratorio ISISLab (*Grande Diletta!!*), una fucina di idee in cui ho potuto approfondire e avvicinarmi a numerose discipline. Incontrando le persone migliori con le quali potessi confrontarmi, ma sopratutto condividere la vita fuori il lavoro (*Lucariè e in che modooo!*).

Ringrazio, in modo particolare, il dottore Jonathan Ozik e sua moglie Sam, i quali mi hanno dato la possibilità di poter affrontare un'esperienza indimenticabile negli Stati Uniti. Per tutto ciò che hai fatto per me: *Grazie Jonathan! Yeah, gotcha*.

Ringrazio, infine, la mia famiglia e *Mauro*; i miei più cari amici: *Prufessò*, *Antonio*, *Simone*, *Antonio G.*, *Umberto*, *Giovanni*; tutti i *Biker&Bikerini* e tutte le persone che ho involontariamente omesso di nominare ma che mi hanno accompagnato nella vita e sostenuto in questo mio percorso.
Desidero ricordare una persona che è stata indispensabile per me, salutandola come posso: *Ciao Clara spero non ci siano troppi errori ortografici!*.

A mia madre Lena!
A mio padre Enzo!
A mia sorella Rox!
A Skipper!
Al mio Sogno!

Contents

1 Introduction

> *The complexity of parallel, networked platforms and highly parallel and distributed systems is rising dramatically. Today's 1,000-processor parallel computing systems will rapidly evolve into the 100,000-processor systems of tomorrow. Hence, perhaps the greatest challenge in computational science today is software that is scalable at all hardware levels (processor, node, and system) . In addition, to achieve the maximum benefit from parallel hardware configurations that require such underlying software, the software must provide enough concurrent operations to exploit multiple hardware levels gracefully and efficiently.*
>
> — **Computational Science: Ensuring America's Competitiveness.** (President's Information Technology Advisory Committee, **2005**)

1.1 Computational Science

In the last twenty years, due to the introduction of scientific computing, the scientific methodological approach has changed. Today in all realms of science, physics, social science, biomedical, and engineering research, defense and national security, and industrial innovation, problems are addressed more and more by a computational point of view.

Computational Science [Com05], also know as scientific computing or scientific computation (SC), is a rapidly growing field that uses advanced computing and data analysis to study real-world complex problems. SC aims to tackle problems using predictive capability to support the traditional experimentation and theory according to a computational approach to problem solving. This new discipline in science combines computational thinking, modern computational methods, hardware and software to face problems, overcoming the limitations of traditional ways.

SC could be summed up in three distinct areas:

1. **Algorithms** (numerical and non-numerical) **and modeling and simulation software** developed to solve science (e.g., biological, physical, and social), engineering, and humanities problems;
2. **Computer and information science** that develops and optimizes the advanced system hardware, software, networking, and data management components needed to solve computationally demanding problems;
3. **The computing infrastructure** that supports both the science and engineering problem solving and the developmental computer and information science.

SC is an interdisciplinary field, and appears primarily by the needs arising from the World War II and the dawn of the digital computer age, when scientists of various disciplines such as mathematics, chemistry, physics and engineering, have collaborated to build and deploy the first electronic computing machines for code-breaking and automated ballistics calculations. Advances in theory and applications of computer science, enable scientists and engineers R&D to address the problems in a way that was impossible to make before.

The Figure 1.1 describes the definition of SC and its relation with this work. As shown in the Figure, SC tackles complex problems using multidisciplinary skills in combination with the computational thinking. Each contribution of this work aims to face a key-aspect of the SC areas. The Figure shows also the contributions of this work (see the circles) to the SC field. These contributions can be divided into three categories:

1. **Frameworks.** Software solutions to develop distributed simulations (DS) and run simulation optimization (SO) processes on High Performance Computing (HPC) as well as Cloud infrastructures. These contributions falls in the SC areas 1 and 2;
2. **Architectures.** Software architectures for visualization of scientific data on the Web. These contributions falls in the SC area 2;
3. **Parallel Languages.** Contributions to improve the effectiveness of parallel languages for HPC systems. These contributions falls in the SC area 2.

A detailed description of the contributions of this thesis will be provided in Section 1.6.

1.2 Motivation

The SC field comprises many disciplines and scientific fields. From the Computer Science point of view, the challenge is to improve the current status of methods, algorithms and applications in order to enhance support for SC in terms of both efficiency and effectiveness of the solutions.

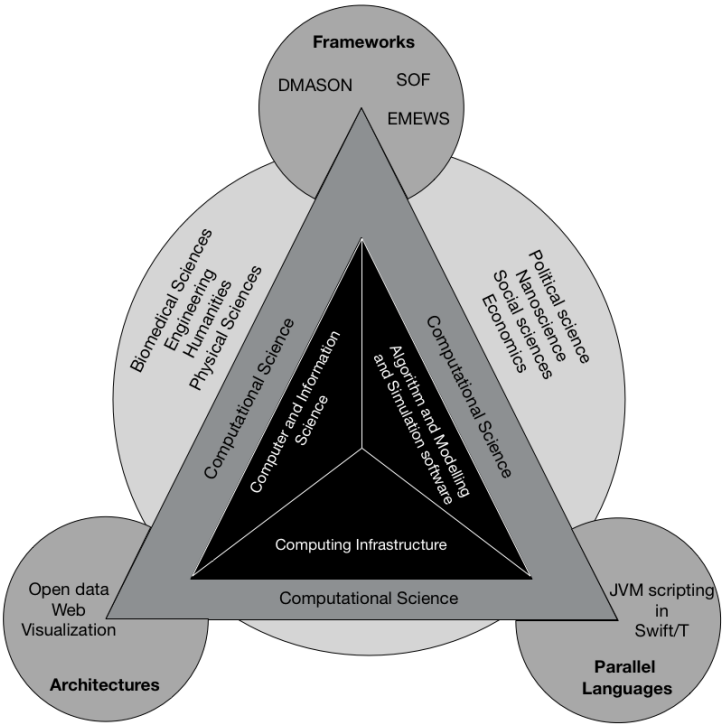

Fig. 1.1.: Computational Science areas and their relations with the contributions of this work.

For instance, one can consider what is needed for the development of new simulation software or for a novel simulation model to study natural disaster or epidemiological emergency. In order to study the effect on a population, it is desirable that our system enables to perform experiments increasing:

- the number of people in the population;
- the complexity of the simulation model and humans behaviors;
- the geographical areas in analysis;
- the complexity of the social interaction between the individuals;
- ...

Unfortunately, there is not a generic answer. It is not possible to describe all possible requirements, their are dependent on the problem itself. Nevertheless, it is important to identify methodologies and solutions that could be helpful to tackle real complex problems.

As described in [Com05] the most important constrains that should guide our research is the scalability. Our solutions, software, models, algorithms, systems should be scalable according to the problem itself. This requirement is not only on the architectures and frameworks software development, but comprises also design solutions to a problem in the SC field.

Scalability is a frequently-claimed attribute of system or solution to face complex problems using computer systems. The definition of Scalability is not generally-accepted (as described in [Hil90]). Nevertheless, we can use some of its principles to better design scalable solutions for the SC field. According to this idea it is possible to state that the Computational Science should be scalable at different software and hardware levels.

In the following a briefly review of some definitions of Scalability is provided.

1.3 What is scalability?

The scalability requirement measures the capability of the system to react to the computational resources used (e.g. It is desirable that our system provides better performance when the number of nodes involved increase). Scalability can be seen in a lot of forms: speed, efficiency, high reliable applications and heterogeneity, [ERAEB05].

The system scalability requirement also refers to the application of very large compute clusters to solve computational problems. A compute cluster is a set of computing resources that work and collaborate together to solve a problem. In the following, we denote with *supercomputing system* a system with a large number of nodes and dedicated hardware architecture.

The availability of supercomputing systems become much affordable day by day, and also Cloud Computing systems - which offer a large number of high performance resources at low-cost - is an attractive opportunity in the SC field. The Top500 Supercomputer site, provides a detailed description of the most powerful available systems; this list is updated twice a year. The Figure 1.2 shows the trend of supercomputing architectures over time 1995–2015. We will refer to this systems as Super-scale and Extra-scale computing systems.

As described in [Bon00] there are different types of system scalability:

1. *Load scalability.* A system is load scalable if it is able to exploits the available resources in heavy loads conditions. This is affected by: the scheduling of the resources and the degree of parallelism exploitation.
2. *Space scalability.* A system is space scalable when it is able to maintain the memory requirements under some reasonable levels (also when the size of the input is large). A particular application or data structure is space scalable if its memory requirements increase at most sub-linearly with the problem input size.
3. *Space-time scalability.* A system is space-time scalable when it provides the same performance whether the system is of moderate size or large. For instance a search engine may use an hash table or balanced tree data structure to index

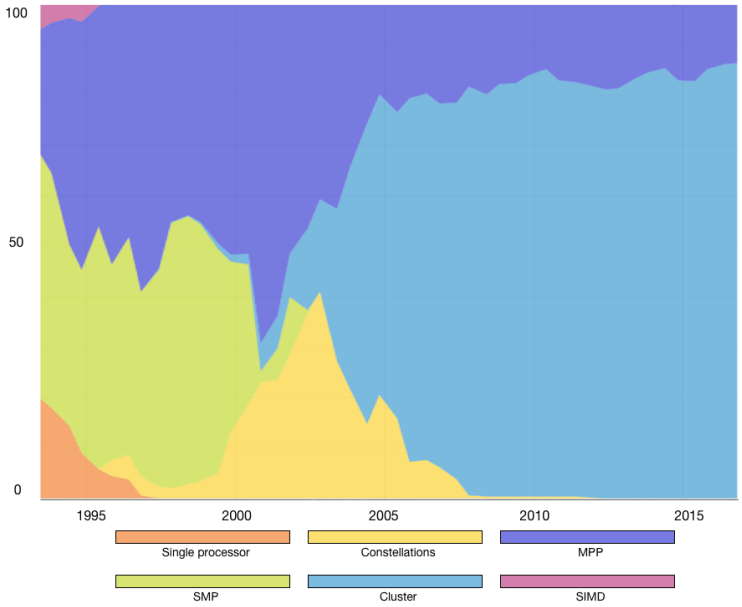

Fig. 1.2.: Systems Architecture Share from Top500.

pages in order to be space-time scalable, while using a list is not space-time scalable. Often the space scalability is a requirement to ensure space-time scalability.

4. *Structural scalability.* A system is structurally scalable if its implementation or architecture does not limit the number of resources or data input. For instance, we can consider the telephone numbering scheme of North American, which uses a fixed number of digits. This system is structurally scalable if the number of objects to assign is significantly lower than the number of possible telephone numbers.

In order to better understand the concept of scalability in the next Section 1.3.1 in the following is provided some background concepts about parallel and distributed computing.

1.3.1 Parallel and Distributed Programming Background

Why we are talking of parallel machine? – An accepted classification of the state of art of computing, divide the history of computing in four eras: batch, time-sharing, desktop, and network [ERAEB05]. Today the trend in computing is to discard expensive and specialized parallel machines in favor of the more cost-effective clusters of workstations and cloud services. Anyway the following concepts may be the same also for this novel systems.

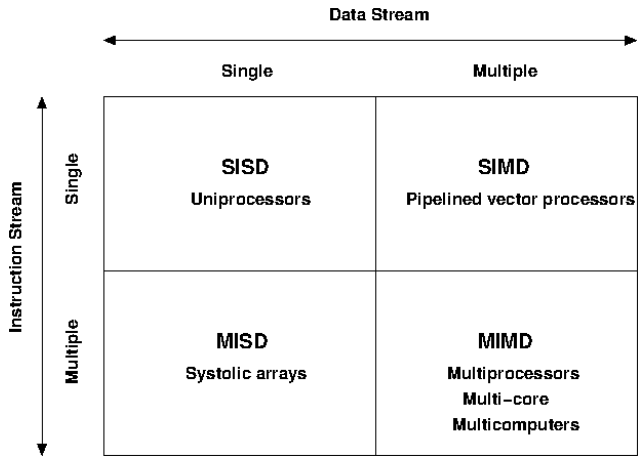

Fig. 1.3.: Flynn's Taxnomy.

Flynn's Taxnomy

To understand the parallel computing, the first concept that is essential to know is the classification of the computer architectures. An important and well know classification scheme is the Flynn taxonomy. Figure 1.3 shows the taxonomy defined by Flynn in 1966.

The classification relies on the notion of stream of information. A processor unit could accept two type of information flow: instructions and data. The former is the sequence of instructions performed by the processing unit, while the latter is the data given in input to the processing unit from the central memory. Both the streams can be single or multiple.

Flynn's taxonomy comprises four categories:

1. *SISD*, Single-Instruction Single-Data streams;
2. *SIMD* Single-Instruction Multiple-Data streams;
3. *MISD* Multiple-Instruction Single-Data streams;
4. *MISD* Multiple-Instruction Multiple-Data streams.

According to the previous definition scheme, the single-processor von Neumann computers are SISD systems while parallel computers (with more processor units) can be either SIMD or MIMD. The classification enables to better classify also parallel machines: in SIMD machines the same instruction is executed on different data in a synchronized fashion; in the MIMD machines it is possible to execute different instructions on different data. The last category, MISD, executes different instructions on the same data stream.

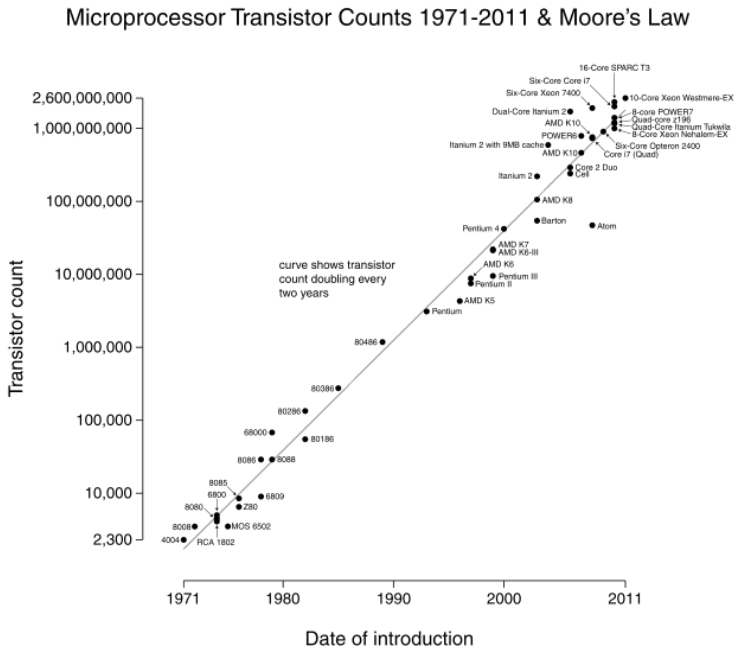

Fig. 1.4.: Microprocessor Transistor counts from 1971 to 2011.

Moore's law

Historically, the way to achieve better computing performances has depended on hardware advances. Now this trend is permanently changed and the parallel and distributed computing is the only way to achieve better performance. The Figure 1.4 depicts the Moore's Law, 1965, which clearly explains this idea.

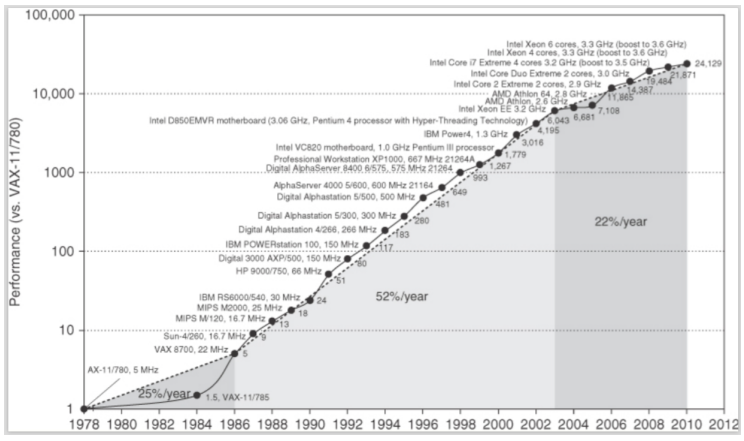

Fig. 1.5.: CPU performance 1978 to 2010.

The Moore's Law shows that the count of CPU and RAM transistor doubled each year. Nonetheless due to physical limits, heat emission and the size of the atom, this trend has ended around 2008 stabilizing the speed processors. That is clearly shown in Figure 1.5 and 1.6).

The actual trend of CPU vendors is to increase the number of core for machine in order to have better performance. In other words, this means that each science has

to face problems exploiting parallel and distributed computing, in order to increment the complexity of the problems or to improve the performances.

Parallel computing architectures

This section describes different approaches to do parallel and distributed computing, depending on system architectures.

Parallel computing architectures are historically categorized in two main groups: shared memory and distributed memory. However the actual trend on parallel computing architectures is to design different architectures combining these two models. Today, three main architectures are considered:

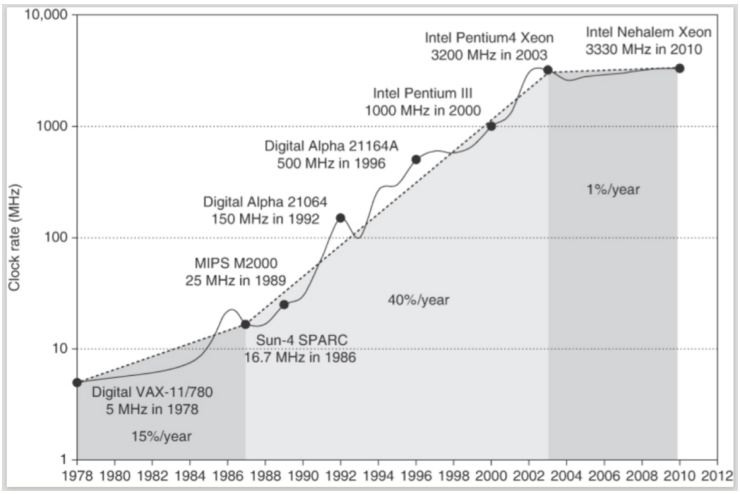

Fig. 1.6.: Clock speed 1978 to 2010.

- *Symmetric multiprocessing (SMP)*, shown in Figure 1.7. In this model, multiple processors use the same memory. This is the most easily and common approach for parallelism, but is also the most expensive for vendors. Sharing the same memory enables to synchronize the processors using shared variables. The limits of this architecture is the bus bandwidth that may represent a performance bottleneck.
- *Multi-core*, shown in Figure 1.8. This is the model adopted by modern processors that employ multiple core in a single processors. This architecture allows to use single processor as a SMP machine.
- *Multi-computers*, shown in Figure 1.9. This architecture is basically the distributed computing architecture. The computers are connected across a network, and the single processor can access only to its local memory space while interaction are based on messages. This architecture is the architecture for clusters and supercomputing machines. Multi-computers is the architecture used for the construction of large parallel machines.

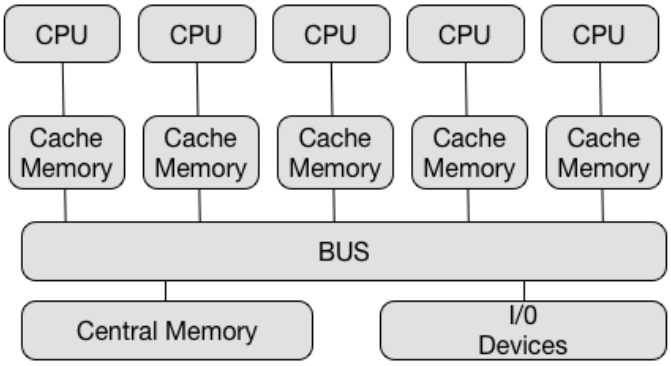

Fig. 1.7.: Symmetric multiprocessing architecture.

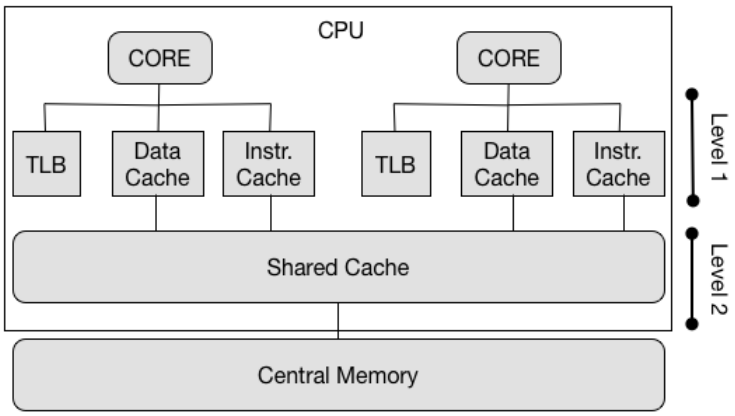

Fig. 1.8.: Multi-core architecture.

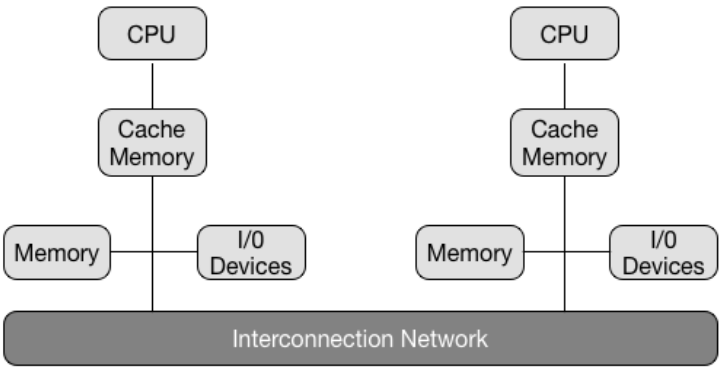

Fig. 1.9.: Multi-computers architecture.

Computational Models

Parallel processing is based on two actions: dividing a computation into a set of execution tasks and assigning them to the computational resources available according to the parallel computing architectures used.

This section describes two theoretical computational models that are independent on the parallel computing architectures used. These models aim to measure the quality

of a solution using parallel computing approach. Starting from these two model it is possible also to describe the concept of scalability for the parallel computing architectures described before.

Equal Duration Model. These model measure the quality of a solution in terms of its execution time. Consider a given task that can be decomposed into n equal sub-tasks, each one executable on a different processor. Let t_s (resp. t_m) be the execution time of the whole task on a single processor unit (resp. concurrently on n processors). Since this model consider that all processors execute their task concurrently, we have $t_m = \frac{t_s}{n}$. So, it is possible to compute the speedup factor ($S(n)$) of a parallel system, as the ratio between the time to execute the whole computation on a single processors (t_s) and the time obtained exploiting n processors unit (t_m).

$$S(n) \stackrel{\text{def}}{=} \frac{t_s}{t_m} = \frac{t_s}{t_s/n} = n$$

Obviously, the previous definition is not enough to describe the speed obtained. One need to consider the communication overhead introduced by the parallel or distributed computation. Let t_c be the communication overhead, the total parallel time is given by $t_m = (t_s/n) + t_c$ and the speedup become.

$$S(n) \stackrel{\text{def}}{=} \frac{t_s}{t_m} = \frac{t_s}{\frac{t_s}{n} + t_c} = \frac{n}{1 + n \times \frac{t_c}{t_s}}$$

This value normalized by n is named efficiency ξ and can be seen as the speedup per processor.

$$\xi = \frac{S(n)}{n} = \frac{1}{1 + n \times \frac{t_c}{t_s}}$$

The value of the efficiency ranges between 0 and 1. Again, this model is not realistic, because it is based on the assumption that a given task can be divided, in "equal" sub-tasks, among n processors. On the other hand, real algorithms contains some serial parts that cannot be divided among processors. Furthermore, a parallel algorithm is also characterized by same sub-tasks that cannot be executed concurrently by processors. These sub-tasks includes synchronization or other special instructions, and are named critical sections. The Figure 1.10 shows a program that have some code segments that could be executed in parallel while some other segments must be executed sequentially (due to interdependencies).

This consideration is the main idea of the next model, Parallel Computation with Serial Sections Model.

Parallel Computation with Serial Sections Model This model assumes that only a fraction f of a task can be divided into concurrent sub-tasks and the remaining fraction $1 - f$ has to be executed in serial way. Like the previous model, the total

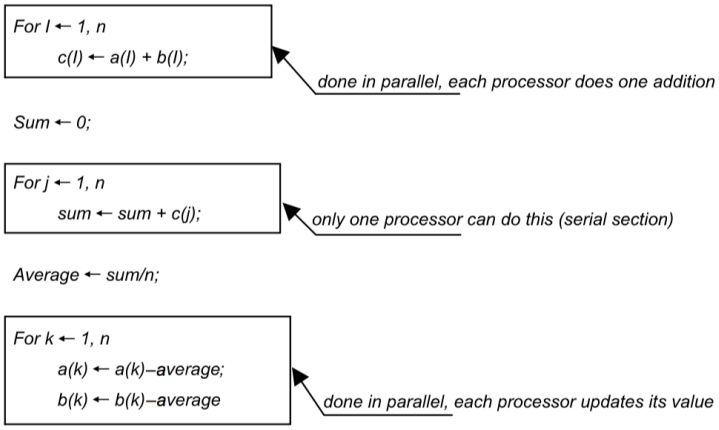

Fig. 1.10.: Example program segments.

time to execute the computation on n processors is $t_m = ft_s + (1-f)(t_s/n)$. In this case the speedup become

$$S(n) = \frac{t_s}{ft_s + (1-f)\left(\frac{t_s}{n}\right)} = \frac{n}{1+(n-1)f} \tag{1.1}$$

As in the equal duration model, the speedup factor considering the communication overhead is given by

$$S(n) = \frac{t_s}{ft_s + (1-f)\left(\frac{t_s}{n}\right) + t_c} = \frac{n}{f(n-1) + 1 + n\left(\frac{t_c}{t_s}\right)}$$

Considering the limit of the number of processors used, we have that the maximum speedup factor is given by

$$\lim_{n\to\infty} S(n) = \lim_{n\to\infty} \frac{n}{f(n-1) + 1 + n\left(\frac{t_c}{t_s}\right)} = \frac{1}{f + \left(\frac{t_c}{t_s}\right)} \tag{1.2}$$

According to equation 1.2, it is worth to notice that the maximum speedup factor depends on the fraction of the computation that cannot be parallelized and on the communication overhead. In this model the efficiency is given by

$$\xi = \frac{1}{1+(n-1)f}$$

without considering the communication overhead, while taking into account the communication overhead we have,

$$\xi = \frac{1}{f + \left(\frac{t_c}{t_s}\right)}$$

This last equation shows that it is difficult to maintain a high level of efficiency as the number of of processors increase.

Parallel Computation Laws

This section introduces two laws that aim to describe the benefits of using parallel computing.

Grosch's Law. H. Grosh, in the 1940, postulated that the power of a computer system P increases in proportion to the square of its cost C, $P = K \times C^2$ with K a positive constant. The Figure 1.11 depicts the relationship between the power and the cost of a computer system. Today this law is clearly abrogated while the research communities and computational scientists are looking for strategies to make the most of HPC and heterogeneous distributed systems. The SC field is one of the most attractive science from this point of view, this field is involved in many real problems that very often could have advantages in using HPC systems, in terms of problems size achievable, speedup and efficiency.

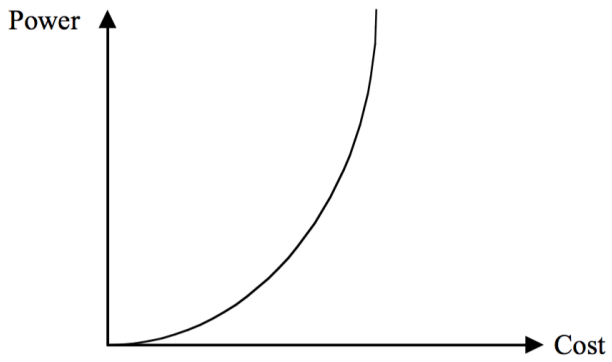

Fig. 1.11.: Power–cost relationship according to Grosch's law.

Amdahl's Law. Starting from the definition of speedup, it is possible to study the maximum speed achievable independently from the number of processors involved in a given computation.

According to equation (1.1), also known as Amdahl's law, the potential speedup, using n processors, is defined by the size of the sequential fraction of the code f. The Amdahl's principle states that the maximum speedup factor is given by

$$\lim_{n\to\infty} S(n) = \frac{1}{f}$$

Nevertheless there are real problems that have a sequential part f that is a function of n, such that $\lim_{n\to\infty} f(n) = 0$. In this cases, the speedup limit is

$$\lim_{n\to\infty} S(n) = \lim_{n\to\infty} \frac{n}{1+(n-1)f(n)} = n$$

This contradicts the Amdahl's law considering that is possible to achieve linear speedup increasing the problem size. This statement has been verified by researchers at the Sandia National Laboratories which show a linear speed up factor can be possible for some engineering problems.

This reassures and supports the hypothesis that it is worth to invest in the SC fields in order to improve the current solutions using parallel and distributed computing. Indeedu SC fields face complex problems also in terms of input dimensions.

1.4 When a system is scalable?

In the SC fields a common principle that lead the research from the computer science point of view is needed. The assumption of this work until now is that the scalability feature could be this principle. In the following we answer the question: "When a system is scalable?".

In the first analysis, a system is scalable when it is able to efficiently exploit an increasing number of processing resources. In practice, the scalability of a system can be analyzed according to different properties:

- *speed*, that is increasing the number of processing resources provides an increasing speed;
- *efficiency*, the efficiency remain unchanged when the processors and the problem size increases;
- *size*, the maximum number of computational resources that a system can accommodate;
- *application*, a software is scalable when it provides better performance when it is executed on a larger system;
- *generation*, a scalable system should achieve better performance according to the generation of components used;
- *heterogeneous*, a scalable system should be able to exploit different hardware and software components.

1.4.1 How to Measure Scalability Efficiency

Given the previous definitions, and postulates, it is possible to measure the scalability of a system. The idea is to measure how efficient an application is when using increasing numbers of computational resources [MO15] (that are cores, processors, threads, machines, CPUs, etc.).

There are two basic ways to measure the parallel performance of a given system, depending on whether or not one is cpu-bound or memory-bound. These are referred to as strong and weak scalability, respectively.

Strong Scalability (SS)

The strong scalability aims to study the number of resources needed, to a given application in order to complete the computation in a reasonable time. For this reason the problem size stays fixed but the number of processing elements are increased. An application scales linearly when the speedup obtained is equal to the computational resources used, n, but it is harder to obtain a linear scalability due to the communication overhead, that typically increases in proportion to the size of n.

Given the completion time of a single task, t_1, and t_m the completion time of the same task on n computational resources, the strong scalability is given by:

$$SS = \frac{t_s}{(n \times t_m)} \times 100$$

Weak Scalability (WS)

The weak scalability aims to define the efficiency of an application fixing the problem size for each computational resource. This measure is useful for studying the memory or resources consumption of an application.

In the case of weak scaling, linear scaling is achieved if the run time stays constant while the workload is increased in direct proportion to the number of computational resources

Given the completion time of a single work unit, t_1, and t_m the completion time of n works unit on n computational resources, the weak scalability is given by:

$$WS = (\frac{t_1}{t_N}) \times 100$$

1.5 Scalable Computational Science

The definition of scalability, for hardware and software system, given in the previous sections, promotes the use of scalability as a major design objective for problem solving in the SC field. The SC aims to solve complex problem though a computational approach. This new way to do science has to exploit efficiently resources available in HPC and Cloud infrastructures.

This work refers to the idea to use scalability as a major design objective for problem solving in SC. Henceforth we call this approach Scalable Computational Science (SCS).

The recommendations presented, at beginning of 2005, in *Computational Science Ensuring America's Competitiveness* by the Information Technology Advisor Committee provides a robust support to the idea of SCS:

> "The Federal government must rebalance its R&D investments to:
>
> - create a new generation of well-engineered, scalable, easy-to-use software suitable for computational science that can reduce the complexity and time to solution for today's challenging scientific applications and can create accurate simulations that answer new questions;
> - design, prototype, and evaluate new hardware architectures that can deliver larger fractions of peak hardware performance on scientific applications;
> - focus on sensor- and data-intensive computational science applications in light of the explosive growth of data.

1.5.1 Real world example

This section describes an example of SCS, in order to better understand in which way the scalability design objective should be involved in the SC field.

Colorectal Cancer Agent-Based Simulation – In 2016, Ozik *et al* show in the paper *High performance model exploration of mutation patterns in an agent-based model of colorectal cancer* [Ozi+16b] an example of SCS that aims to study colorectal cancer (CRC). This work presents an innovative use of Agent-Based Models (ABM) in the field of health studies. ABM simulations are known to have numerous challenges, most of them related to high computational cost of executing simulation in order to calibrate and use it.

The authors uses a framework, Extreme-scale Model Exploration with Swift/T (EMEWS) (also described in the Chapter Simulation Optimization), for executing a large number of simulation concurrently on a large supercomputer IBM Blue Gene/Q at the Argonne Leadership Computing Facility at Argonne National Laboratory. Running concurrently simulation enables to quickly calibrate the ABMs to emulate the CRC evolution's. The calibration of the model was realized validating the ABM's results to an historical database of patients affected of CRC.

The complexity of the ABM for CRC and the dimensions of the parameters space is huge and the calibration of the model was unimaginable without running simulations on a HPC system. This example shows how the use of an HPC ME framework's

enables a deeper study of problems, using a computational approach that is impracticable without the scalability requirements. The Figure 1.12 shows the scaling study of this example on the IBM Blue Gene/Q.

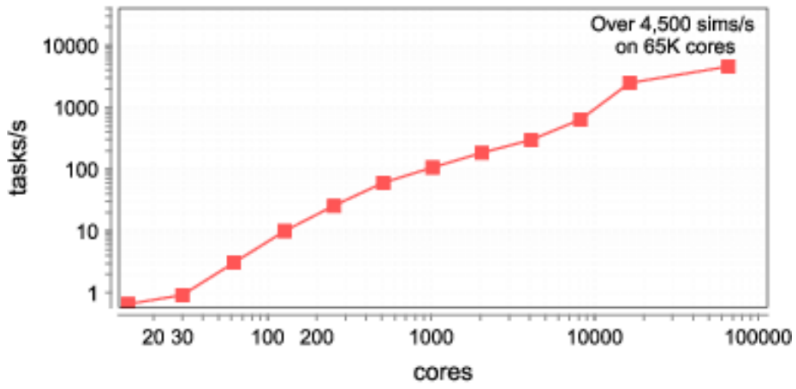

Fig. 1.12.: Scaling study results for EMEWS ABM calibaration workflow on IBM Blue Gene/Q.

1.6 Dissertation Structure

This dissertation discusses Frameworks, Architecture and Parallel Languages for SCS. Chapter 2 provides the results about Distributed Agent-Based Simulation. In details a novel framework D-MASON is presented. Chapter 3 presents the contributions in developing Java scripting integration in the parallel language Swif/T. Thereafter, Chapter 4 describes two framework for Simulation Optimization (SO): *SOF: Simulation Optimization Framework* on the Cloud Computing infrastructure and *EMEWS: Extreme-scale Model Exploration with Swift/T* on HPC systems. In the Chapter 5 is discussed a scalable architecture for scientific visualization of Open Data on the Web. Finally, Chapter 6 presents a summary of the results and analyzes some directions for future research.

1.6.1 Distributed Agent-Based Simulation

In Chapter 2 is described D-MASON, a framework for Distributed and Parallel Agent-Based Simulation (DPABS). D-MASON is based on MASON [Luk+05], a discrete-event multi-agent simulation library core in Java.

D-MASON is composed by four layers: Distributed Simulation, Communication, Visualization and System Management.

D-MASON, was began to be developed since 2011, the main purpose of the project was overcoming the limits of the sequentially computation of MASON, using distributed computing. D-MASON enables to do more than MASON in terms of size of simulations (number of agents and complexity of agents behaviors), but allows also to reduce the simulation time of simulations written in MASON. For this reason, one of the most important feature of D-MASON is that it requires a limited number of changing on the MASON's code in order to execute simulations on distributed systems.

D-MASON, based on Master-Worker paradigm, was initially designed for heterogeneous computing in order to exploit the unused computational resources in labs, but it also provides functionality to be executed in homogeneous systems (as HPC systems) as well as cloud infrastructures.

The architecture of D-MASON is presented in the following three papers, which describes all D-MASON layers:

> [Cor+16b] Cordasco G., Spagnuolo C. and Scarano V. *Toward the new version of D-MASON: Efficiency, Effectiveness and Correctness in Parallel and Distributed Agent-based Simulations.* 1st IEEE Workshop on Parallel and Distributed Processing for Computational Social Systems. IEEE International Parallel & Distributed Processing Symposium 2016.

[Cor+13a] Cordasco G., De Chiara R., Mancuso A., Mazzeo D., Scarano V. and Spagnuolo C. *Bringing together efficiency and effectiveness in distributed simulations: the experience with D-MASON.* SIMULATION: Transactions of The Society for Modeling and Simulation International, June 11, 2013.

[Cor+11] Cordasco G., De Chiara R., Mancuso A., Mazzeo D., Scarano V. and Spagnuolo C. *A Framework for distributing Agent-based simulations.* Ninth International Workshop Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms of Euro-Par 2011 conference.

Much effort has been made, on the Communication Layer, to improve the communication efficiency in the case of homogeneous systems. D-MASON is based on Publish/Subscribe (PS) communication paradigm and uses a centralized message broker (based on the Java Message Service standard) to deal with heterogeneous systems. The communication for homogeneous system uses the Message Passing Interface (MPI) standard and is also based on PS. In order to use MPI within Java, D-MASON uses a Java binding of MPI. Unfortunately, this binding is relatively new and does not provides all MPI functionalities. Several communication strategies were designed, implemented and evaluated. These strategies were presented in two papers:

[Cor+14b] Cordasco G., Milone F., Spagnuolo C. and Vicidomini L. *Exploiting D-MASON on Parallel Platforms: A Novel Communication Strategy* 2st Workshop on Parallel and Distributed Agent-Based Simulations of Euro-Par 2014 conference.

[Cor+14a] Cordasco G., Mancuso A., Milone F. and Spagnuolo C. *Communication strategies in Distributed Agent-Based Simulations: the experience with D-MASON* 1st Workshop on Parallel and Distributed Agent-Based Simulations of Euro-Par 2013 conference.

D-MASON provides also mechanisms for the visualization and gathering of the data in distributed simulation (available on the Visualization Layer). These solutions are presented in the paper:

[Cor+13b] Cordasco G., De Chiara R., Raia F., Scarano V., Spagnuolo C. and Vicidomini L. *Designing Computational Steering Facilities for Distributed Agent Based Simulations.* Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation 2013.

In DABS one of the most complex problem is the partitioning and balancing of the computation. D-MASON provides, in the Distributed Simulation layer, mechanisms for partitioning and dynamically balancing the computation. D-MASON uses field partitioning mechanism to divide the computation among the distributed system.

The field partitioning mechanism provides a nice trade-off between balancing and communication effort. Nevertheless a lot of ABS are not based on 2D- or 3D-fields and are based on a communication graph that models the relationship among the agents. In this case the field partitioning mechanism does not ensure good simulation performance.

Therefore D-MASON provides also a specific mechanisms to manage simulation that uses a graph to describe agent interactions, see Appendix A. These solutions were presented in the following publication:

> [Ant+15] Antelmi A., Cordasco G., Spagnuolo C. and Vicidomini L.. *On Evaluating Graph Partitioning Algorithms for Distributed Agent Based Models on Networks*. 3rd Workshop on Parallel and Distributed Agent-Based Simulations of Euro-Par 2015 conference.

The field partitioning mechanism, intuitively, enables the mono and bi-dimensional partitioning of an Euclidean space. This approach is also know as uniform partitioning. But in some cases, e.g. simulations that simulate urban areas using a Geographical Information System (GIS), the uniform partitioning degrades the simulation performance, due to the unbalanced distribution of the agents on the field and consequently on the computational resources, see Appendix B. In such a case, D-MASON provides a non-uniform partitioning mechanism (inspired by Quad-Tree data structure), presented in the following papers:

> [Let+15] Lettieri N., Spagnuolo C. and Vicidomini L.. *Distributed Agent-based Simulation and GIS: An Experiment With the dynamics of Social Norms*. 3rd Workshop on Parallel and Distributed Agent-Based Simulations of Euro-Par 2015 conference.

> [Cor+17b] G. Cordasco and C. Spagnuolo and V. Scarano. *Work Partitioning on Parallel and Distributed Agent-Based Simulation*. IEEE Workshop on Parallel and Distributed Processing for Computational Social Systems of International Parallel & Distributed Processing Symposium, 2017.

The latest version of D-MASON provides a web-based System Management, to better use D-MASON in Cloud infrastructures. D-MASON on the Amazon EC2 Cloud infrastructure and its performance in terms of speed and cost were compared against D-MASON on an HPC environment. The obtained results, and the new System Management Layer are presented in the following paper:

> [Car+16a] M Carillo, G Cordasco, F Serrapica, C Spagnuolo, P. Szufel, and L. Vicidomini. *D-Mason on the Cloud: an Experience with Amazon Web Services.* 4rd Workshop on Parallel and Distributed Agent-Based Simulations of Euro-Par 2016 conference.

1.6.2 SWIFT/T Parallel Language and JVM scripting

Swft/T ([Com]), is a parallel scripting language for programming highly concurrent applications in parallel and distributed environments. Swift/T is the reimplemented version of Swift language, with a new compiler and runtime. Swift/T improve Swift, allowing scalability over 500 tasks per second, load balancing feature, distributed data structures, and dataflow-driven concurrent task execution.

Swif/T provides an interesting feature the one of calling easily and natively other languages (as Python, R, Julia, C) by using special language functions named leaf functions.

Considering the actual trend of some supercomputing vendors (such as Cray Inc.) that support in its processors Java Virtual Machines (JVM), it is desirable to provide methods to call also Java code from Swift/T. In particular is really attractive to be able to call scripting languages for JVM as Clojure, Scala, Groovy, JavaScript etc.

For this purpose a C binding to instanziate and call JVM was designed, and is described in the Chapter 3. This binding is used in Swif/T (since the version 1.0) to develop leaf functions that call Java code. The code are public available at GitHub project page .

1.6.3 Simulation Optimization

Complex system simulation gains relevance in business and academic fields as powerful experimental tools for research and management. Simulations are mainly used to analyze behaviors that are too complex to be studied analytically, or too risky/expensive to be tested experimentally [Law07; TS04].

The representation of such complex systems results in a mathematical model comprising several parameters. Hence, the need for tuning a simulation model arises, that is finding optimal parameter values which maximize the effectiveness of the model.

Considering a multi-dimensionality of the parameter space, finding out the optimal parameters configuration is not an easy undertaking and requires extensive computing power. Simulations Optimization (SO) [TS04; He+10] is used to refer to the techniques studied for ascertaining the parameters of the model that minimize (or maximize) given criteria (one or many), which can only be computed by performing a simulation run.

Chapter 4 describes two frameworks for SO process, respectively, primarily designed for Cloud infrastructure and HPC systems.

The first frameworks is *SOF: Zero Configuration Simulation Optimization Framework on the Cloud*, it was designed to run SO process in the cloud. SOF is based on the Apache Hadoop [Tur13] infrastructure and is presented in the following paper:

> [Car+16b] Carillo M., Cordasco G., Scarano V., Serrapica F., Spagnuolo C. and Szufel P. *SOF: Zero Configuration Simulation Optimization Framework on the Cloud.* Parallel, Distributed, and Network-Based Processing 2016.

The second framework is *EMEWS: Extreme-scale Model Exploration with Swift/T*, it has been designed at Argonne National Laboratory (USA). EMEWS as SOF allows to perform SO processes in distributed system. Both the frameworks are mainly designed for ABS. In particular EMEWS was tested using the ABS simulation toolkit Repast. Initially, EMEWS was not able to easily execute out of the box simulations written in MASON and NetLogo [TW04]. This thesis presents new functionalities of EMEWS and solutions to easily execute MASON and NetLogo simulations on it.

The EMEWS use cases are presented in the following paper:

> [Ozi+16a] J. Ozik, N. T. Collier, J. M. Wozniak and C. Spagnuolo *From Desktop To Large-scale Model Exploration with Swift/T.* Winter Simulation Conference 2016.

1.6.4 Scalable Web Scientific Visualization

In the SC field, an important area of interest regards methods and tools for scientific visualization of data. This work describes an architecture for the visualization of data on the Web, tailored for Open Data.

Open data is data freely available to everyone, without restrictions from copyright, patents or other mechanisms of control. The presented architecture allows to easily visualize data in classical HTML pages. The most important design feature concerns the rendering of the visualization that is made on the client side, and not on the server side, as in other architecture. This ensure the scalability in terms of number of concurrent visualizations, and dependability of the data (because the data are dynamically loaded client side, without any server interactions).

This Chapter 5 describes the proposed architecture, that has also appeared in the following papers:

> [Cor+17a] G. Cordasco, D. Malandrino, P. Palmieri, A. Petta, D. Pirozzi, V. Scarano, L. Serra, C. Spagnuolo, L. Vicidomini *A Scalable Data Web Visualization Architecture.* Parallel, Distributed, and Network-Based Processing 2017.
>
> [Mal+16] G. Cordasco, D. Malandrino, P. Palmieri, A. Petta, D. Pirozzi, V. Scarano, L. Serra, C. Spagnuolo, L. Vicidomini *An Architecture for Social Sharing and Collaboration around Open Data Visualisation.* In Poster Proc. of

the 19th ACM conference on "Computer-Supported Cooperative Work and Social Computing 2016.
[Cor+15] G. Cordasco, D. Malandrino, P. Palmieri, A. Petta, D. Pirozzi, V. Scarano, L. Serra, C. Spagnuolo, L. Vicidomini *An extensible architecture for an ecosystem of visualization web-components for Open Data* Maximising interoperability Workshop— core vocabularies, location-aware data and more 2015.

2 Distributed Agent-Based Simulation

> *How does variation in the number of interacting units (grid size) affect the main results of an agent-based simulation?.*
>
> — **Claudio Cioffi-Revilla**
> (*Invariance and universality in social agent-based simulations,* ***2002***)

2.1 Introduction

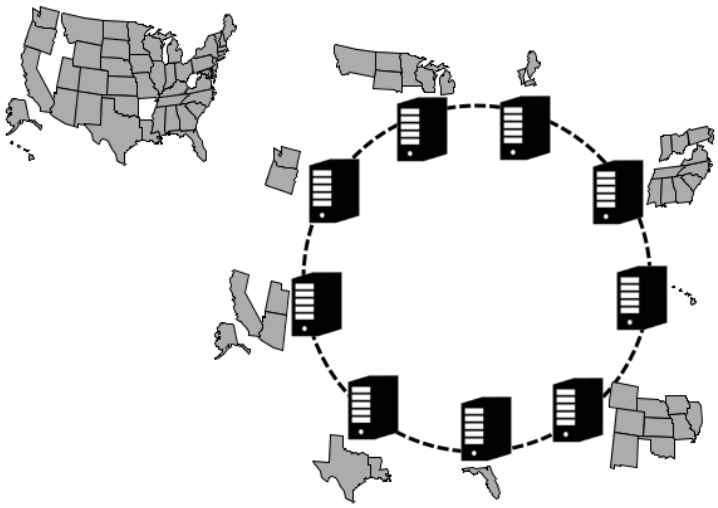

Fig. 2.1.: Distributed Simulation on a ring network.

The traditional answer to the need for HPC is to invest resources in deploying a dedicated installation of dedicated computers. Such solutions can provide the computing power surge needed for highly specialized customers. Nonetheless a large amount of computing power is available, unused, in common installations like educational laboratories, accountant department, library PCs.

SC, as described before, uses computer simulation (see the example in Figure 2.1) to investigate solutions and study complex real problems. This scenario of use is quite common in the context of heterogeneous computing where different computing platforms participate to the same computing effort contributing with its own specific capabilities. One of the most challenging aspect in parallel computing is to balance

the work load among the machines that provides the computation. Indeed, due to synchronization constraints, the entire computation advances with the speed of the slowest machine, which may represent a bottleneck for the overall performances. Of course this issue become particularly challenging in the context of heterogeneous computing.

Software for computer simulation in the context of SC should be able to exploit both HPC, characterized by a large number of homogenous computing machines, and heterogeneous systems. This Chapter focuses on a particular class of computational model, named Agent-Based simulation Models (ABMs).

2.1.1 Agent-Based simulation Models

Motivation. ABMs are an increasingly popular tool for research and management in many fields such as ecology, economics, sociology, etc. In some fields, such as social sciences, ABMs are seen as a key instrument [LP+12] to the generative approach [Eps+07], essential for understanding complex social phenomena. But also in policy making, biology, military simulations, control of mobile robots and economics [Cha10; Mus+09; Cra+10; Ros08; ECoF09], the relevance and effectiveness of ABMs is recently recognized.

The computer science community has responded to the need for platforms that can help the development and testing of new models in each specific field by providing tools, libraries and frameworks that enable to setup and run ABMs.

Parallel and distributed Computing. The exponential growth of local networking and Internet connectivity coupled with continuous increasing of computational power available on average desktop PCs, has made distributed computing very popular in recent years. Two principal desktop grid phenomena emerged: *public resource computing*, such as BOINC [And04], which exploits the power of volunteer desktop computer to perform a specific task; and *Enterprise Desktop Grid Computing* (EDGC) which refers to specific software infrastructures, such as WinGrid [MT09], that allows to harvest the computing power of desktop PC (usually confined to an institution) to support the execution of several enterprise distributed applications. EDGC are typically based on a Master/Worker paradigm [Fos95]. In this paradigm, the user sends a complex job on a master computer which divides the job into a set of independent tasks. Then, each task is sent to an available worker which processes the task and sends back the output. Finally, the master obtains the results of the whole computation reassembling outputs.

Challenges. Several important issues in evaluating different platforms for ABMs, well identified in the reviews of the state of the art such as [Mat08; Naj+01; Rai+06], are speed of execution, flexibility, reproducibility, documentation, open-source and facilities for recording and analysis of data.

While the computational complexity is clearly addressed by achieving efficiency in the simulation, another important aspect is the effectiveness of the solution, which consists of how easily usable and portable are the solutions for the users, i.e. the programmers of the distributed simulation.

2.1.2 The need for scalability and heterogeneity

In [CR10] a fundamental difference is described (from the research point of view) between the computational social science and the complex social systems that have manifold agents that interact with changing natural and/or artificial environments. The nature of such "socially situated agents" [Con99] is highly demanding in terms of computational and storing capabilities. Because of their complexity, these research projects do require consistent multidisciplinary efforts and need also a specific scientific research methodology that is distinct by what is needed for simpler social simulations. The methodology proposed in [CR10] defines a succession of models, from simpler to more complex ones, with several requirements to be followed at each step.

In particular, one of the defining features of complex simulations is their *experimental* capacity, that requires a viable and reliable infrastructure for running (and keeping the records of) several experiments, with the exact procedure that was followed at each run. The resources needed to run and store results of such a sequence of experimental runs can be cheaply ensured only by *heterogeneous* cluster of workstations (available in the research lab), since the nature of interactive experiments, led by the social scientists with their multidisciplinary team, requires interaction with the computing infrastructure, which is often extremely expensive and technically demanding to get from supercomputing centers (that may, in principle, provide massive homogeneous environment).

As a matter of fact, the research in many fields that uses the simulation toolkits for ABMs is often conducted interactively, since the "generative" paradigm [Eps+07] describes an iterative methodology where models are designed tested and refined to reach the generation of an outcome with a simple generative model. In this context, given that scientists of the specific domain often are not computer scientists, usually they do not have access to systems for high performances computations for a long time, and usually they have to perform preliminary studies within their limited resources and, only later (if needed), allow extensive testing on large supercomputing centers. In social sciences, for example, the need for "*the capacity to model and make up in parallel, reactive and cognitive systems, and the means to observe their interactions and emerging effects*" [CC95] clearly outlined, since 1995, the needs of flexible, though powerful, tools.

Then, the design of ABMs is usually done by domain experts who seldom are computer scientists and have limited knowledge of managing a modern parallel

infrastructure as well as developing parallel code. In this context, our goal is to offer such scientists a setting where a simulation program can be run on one desktop, first, and can, later, harness the power of other desktops in the same laboratory, thereby allowing them to scale up the size they can treat or to significantly reduce the time needed to run the simulation. The scientist, then, is able to run extensive tests by enrolling the different machines available, maybe, during off-peak hours.

Of course, it means that the resulting distributed system, by collecting hardware from research labs, administration offices, etc., can be highly heterogeneous in nature and, then, the challenge is how to efficiently use such hardware without an impact on the "legitimate" user (i.e., the owner of the desktop) both on performances and on installation/customization of the machine. On the other hand, a program in MASON should not be very different than the corresponding program in D-MASON so that the scientist can easily modify it to run over an increasing number of hosts. These ensures that also an easily migration on a HPC system.

This work is aiming at efficient *and* effective distributed simulations by adopting a framework-level approach, ensuring these properties is possible to achieve scalability in this scenario. A novel framework for distributed simulations, named D-MASON, has been designed and implemented. D-MASON is a parallel version of the MASON [Luk+04] library for writing and running simulations of ABMs. D-MASON addresses, in particular, speed of execution with no harm on other features that characterize MASON. The intent of D-MASON is to provide an efficient and effective way of parallelizing MASON ABMs: efficient because D-MASON is faster and can handle larger simulations than MASON; effective because, in order to be able to use this additional computing power, the developer has to do simple incremental modifications to existing MASON ABMs, without re-designing them.

2.2 Agent-Based Simulation: State of Art

D-MASON was designed, using MASON core, and inspired by two widespread toolkits for ABMs, NetLogo [TW04] and Repast [Nor+07]. They have been already studied, from different point of view, in [Mat08] but the next section intends to extend this comparison by spotting the differences between these systems and D-MASON in the way they approach the problem of scalability.

Repast

Repast is available as a suite of open-source applications: Repast Simphony and Repast for High Performance Computing (RepastHPC) [CN11]. Simphony allows the user to approach its functionalities from different entry points: it allows to develop programs in ReLogo, that is a Logo-like language easy to be handled with little

programming knowledge; there is also a visual programming option called Repast Flowchart; it provides a C++ interface that permits the development of portable code to be executed on Repast HPC.

Repast HPC is the version of Repast that offers facilities to leverage the computing power of clusters. In Repast HPC the programmer has to take into account that some of the information of the simulation must be propagated across the cluster to each of the workers involved in the computation. In [Che+08] the authors present HLA_Grid_RePast, a middleware for the execution of collaborating Repast applications on the Grid. The system considers Repast applications as services on a Grid while the effort that allows the interoperability of models is left to the programmer.

Repast is available as a suite of open-source applications; it allows to develop programs in ReLogo, which is a Logo-like language easy to be handled with little programming knowledge. RepastHPC [CN11] and HLA_Grid_RePast [Che+08] explicitly address the problem of developing ABMs capable of exploiting the computational power of parallel/distributed platforms. In particular RepastHPC and HLA_Grid_RePast specialize the functionalities of Repast Symphony for the context of HPC and Grid, respectively, and require that design of the ABM explicitly takes into account the fact that it is implemented in a parallel environment where there is the need for synchronization between the different machines that carry out the computation.

D-MASON is different because the parallelization is implemented at framework-level without modifying the interface on which simulations are implemented, letting them gracefully capable to run in parallel on a distributed computing system.

NetLogo

NetLogo allows the user to design the simulation by using a functional language inspired by Logo and, for this reason, it is considered to be easily grasped by a wider audience; furthermore NetLogo offers numerous facilities to support the analysis of the simulation. For example, NetLogo offers BehaviorSpace that allows to automatically running a given simulation with different parameters settings, allowing the automatic exploration of the parameters' space. BehaviorSpace can exploit the parallelism of the machine by running more simulations at the same time. *These functionalities of NetLogo are similar to those described in Chapter Simulation Optimization.* The exploration of the parameters' space can exploit the parallelism of the machine by running more simulations at the same time but is not useful to run massive simulation (i.e., simulating a large number of agents) and/or simulations which deal with complex agents, that is, computationally intensive agents.

MASON

MASON toolkit is a discrete-event simulation core and visualization library written in Java, designed to be used for a wide range of ABMs. The toolkit is written, using the standard Model-View-Controller (MVC) design pattern [Ree79], in two layers (see Figure 2.2): the *simulation* (or model) layer and the *visualization* layer, that plays also the role of controller.

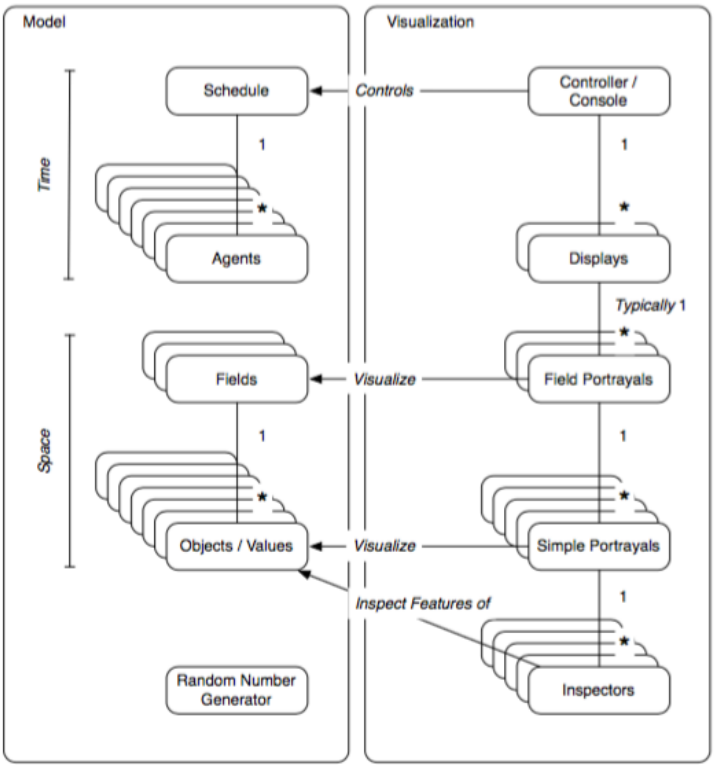

Fig. 2.2.: MASON architecture.

The simulation layer is the core of MASON and is mainly represented by an event scheduler and a variety of fields which hold agents into a given simulation space. MASON is based on *steppable* agent: a computational entity which may be scheduled to perform some action (step), and which can interact (communicate) with other agents. The visualization layer permits both visualization and manipulation of the model. The simulation layer is independent from the visualization layer, which allows us to treat the model as a self-contained entity.

The main reasons that directed us toward a distributed version of MASON are:

- MASON is one of the most expressive and efficient library for ABMs (as reported by many reviews [Mat08; Naj+01; Rai+06]);
- MASON structure, that clearly separates visualization by simulation, makes it particularly well suited to the re-engineering into a distributed "shape" of the framework [Luk+04; Luk+05];

- the significant amount of research and simulations already present in MASON, which makes it particularly cost effective for the scientists.

Distributed Agent-Based Simulation

As observed by many authors [LT00; Hyb+06; Mus+09; Cra+10], the computational requirement of simulators usually exceed the capability of conventional sequential computer. Several distributed simulation systems have been proposed in order to speed up the execution of simulations. Two simulator models have been considered [Hyb+06]: Discrete Event Simulations (DES) have been showed to be efficient, scalable and embarrassingly parallelizable but are not well suited to complex agent based applications; ABMs, indeed, are much more expressive. They implements the *sense-think-act* cycle and offer a real agent-based programming model but unfortunately, due to the high level of interdependencies between agents, these models [Min+96; Luk+05; Nor+07] are not easy to parallelize.

Previous research on distributing ABMs was mainly focused on efficiency. In fact, the need for efficiency among the Agent-Based modeling tools is well recognized in literature: many reviews of state-of-the-art frameworks [Mat08; Naj+01; Rai+06] place "speed" upfront as one of the most general and important issues.

Therefore, the approach followed was that of managing explicitly the distribution of agents on several computing nodes (see for recent examples [CN11; Men+08; PS09]), in order to get the most from the efficiency point of view.

Our framework-level approach here is different since it brings in effectiveness and simplicity: scientists who use the framework (domain experts but with limited knowledge of computer programming and systems) can be mostly unaware of the distribution of agents. Previous work on distributed frameworks, such as [CN11; Men+08; PS09], was focused on the implementation and the architecture of a distributed agent model (dealing with lazy synchronization etc.), while our is on the upper layer of the simulation framework, with the purpose of hiding, as much as possible, the details of the architecture.

A good parallel implementation should address several conflicting issues:

1. balance the overall load distribution;
2. minimize the communication overhead due to tasks' inter-dependencies;
3. synchronize the evolution of the simulation among the machines that provide the computing power;
4. provide reproducibility of the results, meaning that a simulation model should not provide a different behavior for each execution (even using different system infrastructures). This reproducibility feature is considered as a priority for

long simulations because it allows to compare results obtained using different computers [Luk+04].

As an example, a simple way to partition the whole computation into different tasks is to assign a fixed number of agents to each available logical processor (LP) [HX98]. A logical processor is an abstraction of a computational resource. This approach, named *agents partitioning*, enables a balanced workload distribution but it may introduce a significant communication overhead (a very small number of agents' interdependencies, may result in all–to–all communication, required to synchronize the simulation).

Other strategies which partition the work in a smarter way [Cos+11] have been proposed in order to reduce the communication and synchronization time.

Distributed and Parallel Agent-Based Simulation softwares are described by the communication paradigm used and parallel architecture exploited. The table 2.1 shown the mainly popular softwares available, as described in [Rou+14].

Name	**Developer**	**Communication Paradigm**
D-MASON	Università degli Studi di Salerno (Italia)	*Java Message Service* (JMS) and/or *Message Passing Interface* (MPI)
FLAME	University of Sheffield (Inghilterra)	MPI
FLAME GPU (CUDA)	University of Sheffield (Inghilterra)	-
JADE	Telecom Italia lab (Italia)	JAVA *Remote Method Invocation* (RMI)
Pandora	Barcelona Supercomputing center (Spagna)	MPI
RepastHPC	Argonne National Laboratory (USA)	MPI
PDES-Mas	University of Birmingham (Inghilterra)	MPI
SWAGES	University of Notre Dame (USA)	XML-RPC and SSML
Ecolab	University of New South Wales (Australia)	MPI
ABM++	Los Alamos National Laboratory (USA)	MPI

Tab. 2.1.: Distributed Agent-Based Simulation Softwares.

2.3 D-MASON

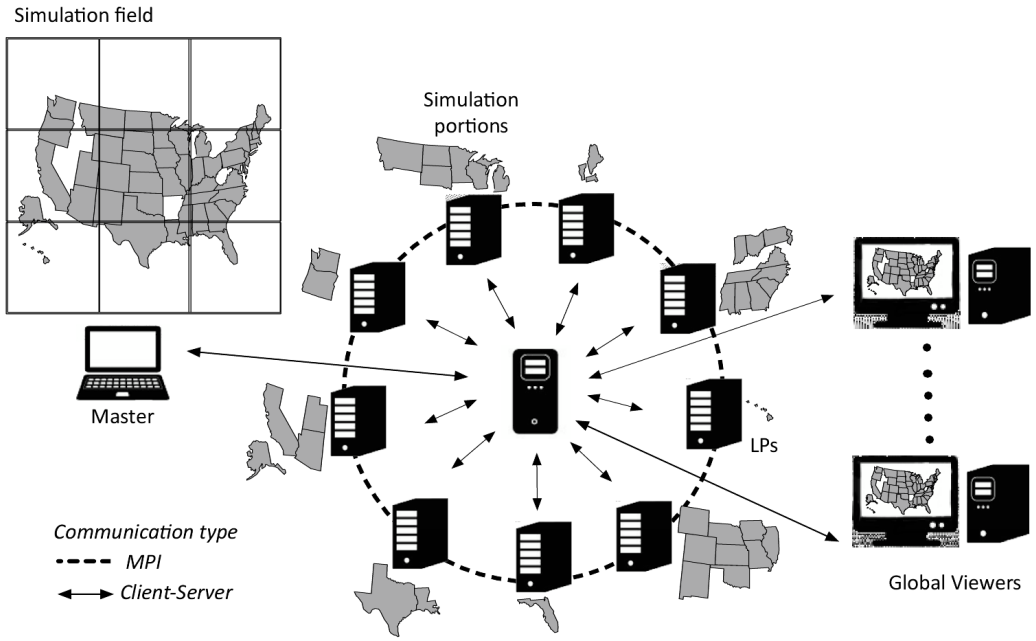

Fig. 2.3.: D-MASON Architecture.

Traditional architecture of computer systems usually includes a stack of layers, each one only interacting directly with the one above and below (if any) (see Figure 2.4). The advantages in flexibility, maintenance and sustainability of this architecture in software design are clear and known to the computer science community since decades.

In our view, two are the main opposite forces that drive the design of scalable, distributed ABMs. The first one is the quest for *efficiency*: the ultimate goal for the simulation of extremely large and time-consuming models. In this context, this force is propelled by the well-known end-to-end principle [D.P+84] that tries to push higher up in the layers any effective attempt to achieve efficiency. In this context the principle states that it is possible to gain substantial improvements on efficiency only if it is possible to address the semantics of the (parallel) application itself, although additional support by the lower layers is helpful. Some examples of this approach include recent results on Land use [Tan+11], on burglary [Mal+12] as well as a comparison of effectiveness of parallelism at the application level with respect to super-individual methods [PE08], where single individuals are aggregated by carefully changing the model itself, so that results of the simpler simulation in the new model is consistent with the original one.

The opposite force is driven by the inherent complexity of the models and (therefore) of the simulations, that requires coordinated work by a multidisciplinary team that is involved in several repeated iterations of the method. It is fundamental, then,

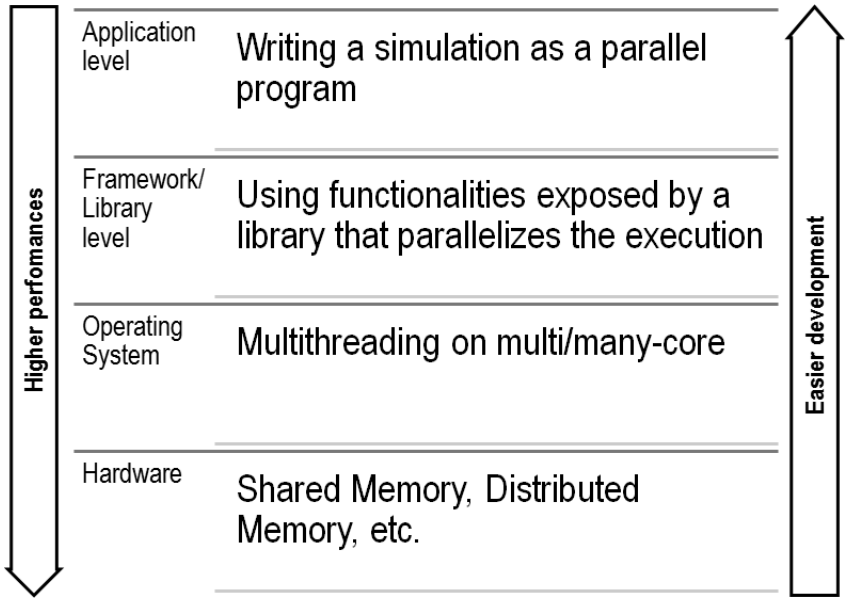

Fig. 2.4.: Trading easiness of development for efficiency of the implementation

in order to achieve team *effectiveness* in such multidisciplinary environment, that the development complexity is significantly reduced, by providing support to the team. Therefore, scientists should not be directly aware of the parallelism embedded into the simulation, just assuming one has "infinite" computational and storage capabilities, sufficient to simulate the massive model without further complications on such "supercomputer". Therefore, to increase the effectiveness of the development the awareness of the parallelism must be pushed down the layers, so that the team can know the least (ideally, nothing) about the parallelization and the distribution of the simulation.

The design of D-MASON is inspired by the need for efficiency, in a distributed setting where computing resources are scarce, heterogeneous, not centrally managed and that are used for other purposes during other periods of the workday. Moreover, the multidisciplinarity of the teams that use ABMs, often, places an important emphasis on easiness of development, thereby suggesting our compromise between efficiency and impact by acting at the **framework-level**. In this way, the scientists still can use the same computing and storage abstractions they are familiar with, in order to build a simulation from a given model. The (modified) framework is able to execute the simulation within a distributed simulation, thereby achieving both efficiency and effectiveness. Finally, our approach is cost-effective since it allows a high degree of backward-compatibility with MASON simulations, because of the moderate number of modifications into the source code of an existing MASON application.

2.3.1 D-MASON Design Principles

D-MASON is based on MASON, and its first architectural requirement was to add functionalities to MASON without changing MASON code. This is not only for

the system scalability, but to ensure that the implementation and the validation of common functionality can be inherited by MASON. This ensures also that a simulation written in MASON and migrated to D-MASON does not need to be changed in its logic but only adapted to be executed in distributed environment.

D-MASON adds a new layer named *Distributed Simulation (D-Simulation)*, which extends the MASON simulation layer. The new layer adds some features that allow the distribution of the work on multiple, even heterogeneous, LPs. It is worth to mention that the new layer does not alter, in any way, the existing layers. Moreover, it has been designed to enable the porting of existing applications to a distributed platform in a transparent and easy way, requiring only a minor revision of the MASON code.

D-MASON is based on the well-know Master/Workers paradigm that exploits a space partitioning approach for work partitioning: the master partitions the space to be simulated (i.e., the field) into cells (see Figure 2.3). Each worker simulates one or more LPs, according to its computational capabilities. Each cell, together with the agents it contains, is assigned to a LP; then each LP is in charge of:

- simulating the agents that belong to the assigned cell;
- handling the migration of agents;
- managing the synchronization between adjacent cells (this information exchange is required in order to let the simulation run consistently).

2.3.2 D-MASON Design Issues

ABMs as step-wise computations; i.e., agents' behavior is computed in successive steps named *simulation step*. Between each simulation step the LPs must synchronize their memory, in order to perform the next step. In the design of D-MASON four design issues have been identified, and described below: Work partitioning, Communication, Synchronization and Reproducibility.

Work partitioning

The problem of decomposing a program to a set of processors (LPs) has been extensively studied (see [HX98] for a comprehensive presentation). In the case of ABMs, a simple way to partition the whole work into different tasks is to assign a fixed number of agents (proportional to the power of the worker) to each available worker. This approach, named *agents partitioning*, allows a balanced workload but introduce a significant communication overhead since, at each step, agents can interact-with/manipulate other agents, then, in principle, an all–to–all communication among workers is required.

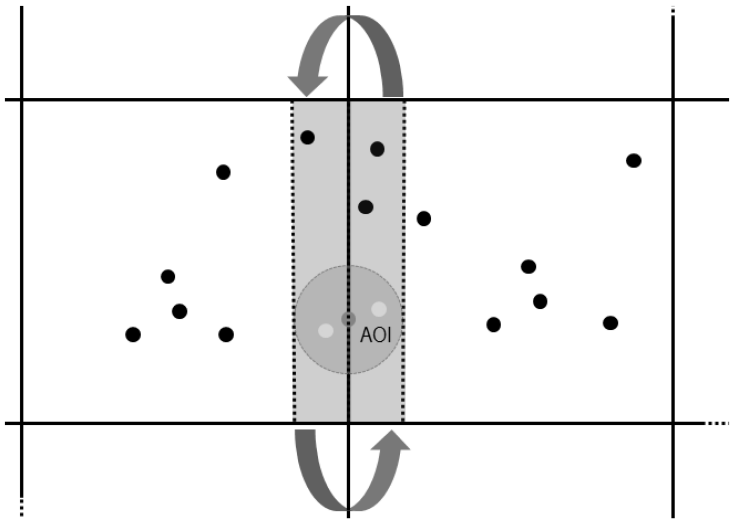

Fig. 2.5.: Field partitioning.

By noticing that most ABMs are inspired by natural models, where agents limited visibility allow to bound the range of interaction to a fixed range named agent's *Area of Interest* (AOI), several field partitioning approaches have been proposed [Cos+11; Zha+10; ZZ04] in order to reduce the communication overhead. Since the AOI radius of an agent is small compared with the size of a cell, the communication is limited to local messages (messages between LPs, managing adjacent spaces, etc.). With this approach agents can migrate between adjacent cells and consequently the association between workers and agents changes during the simulation. In D-MASON by design, agents are allowed to migrate only between adjacent cells. This is consistent with a large family of models (e.g. biology inspired models) that do not need any kind of *"teleportation"*.

In D-MASON are available two principal kind of field partitioning:

- *Uniform partitioning,* which divides the simulation field F in uniform cells (i.e., cells having equal size). The partitioning is described by a matrix $r \times c$ superimposed on the field. The Figure 2.6 depicts a 3×3 uniform partitioning, using 9 LPs, applied to a case study simulation, that uses a 2D field representing the United State of America. In such case, the agents are distributed in a non-uniform way over the field, the zones in gray color represent high population density ones. The regions in yellow, red and green colors are the overlapping regions (aka ghost regions) between the cells, that are defined by the AOI range.
- *Non-uniform partitioning,* which exploits the information available on the simulation field F (e.g., agents' positioning and their complexity) to provide a roughly balanced partitioning. The field partitioning, in this case, is described by a tree data structure [OL82] where each leaf represents a cell (see Figure

2.7 and 2.8). In this case the cell size may vary in order to counterbalance the non-uniformity of agents on the field or their computational complexity.

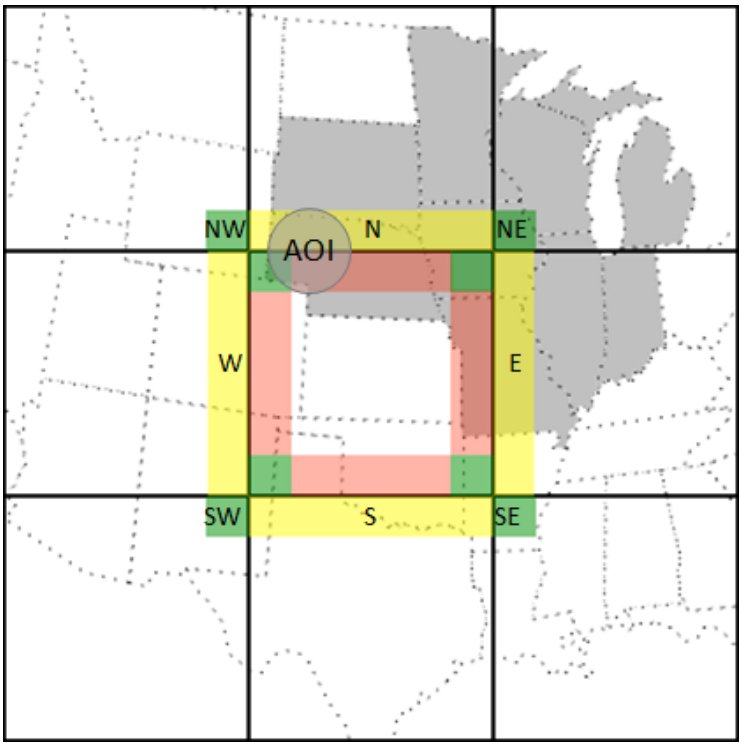

Fig. 2.6.: Uniform field partitioning with 9 LPs.

The Appendix B provides an analytical study about the field partitioning methods, in particular, the study is focused on the communication effort induced by the field partitioning approach used. This study motivates the introduction of the non-uniform field partitioning approach. Scalability of field partitioning strategies will be analyzed in Section 2.6.1.

Load Balancing The problem of a field partitioning approach is that since agents can migrate between adjacent cells, the association between workers and agents changes during the simulation. Moreover, load balancing is not guaranteed and need to be addressed by the application. To better exploit the computing power provided by the workers of the system, it is necessary to design the system to avoid bottlenecks. Since the simulation is synchronized after each step, the system advances with the same speed provided by the slower worker in the system. For this reason it is necessary to balance the load among workers.

The choice of the partitioning strategy is important for the efficiency of the whole system. Two key factors need to be considered: (i) static vs dynamic partitioning and (ii) the granularity of the world decomposition. Dynamic partitioning can be useful, for instance, when the workload of the simulation changes along the time. Unfortunately, the management of dynamic cells requires a large amount of communication between workers that consumes bandwidth and introduces latency.

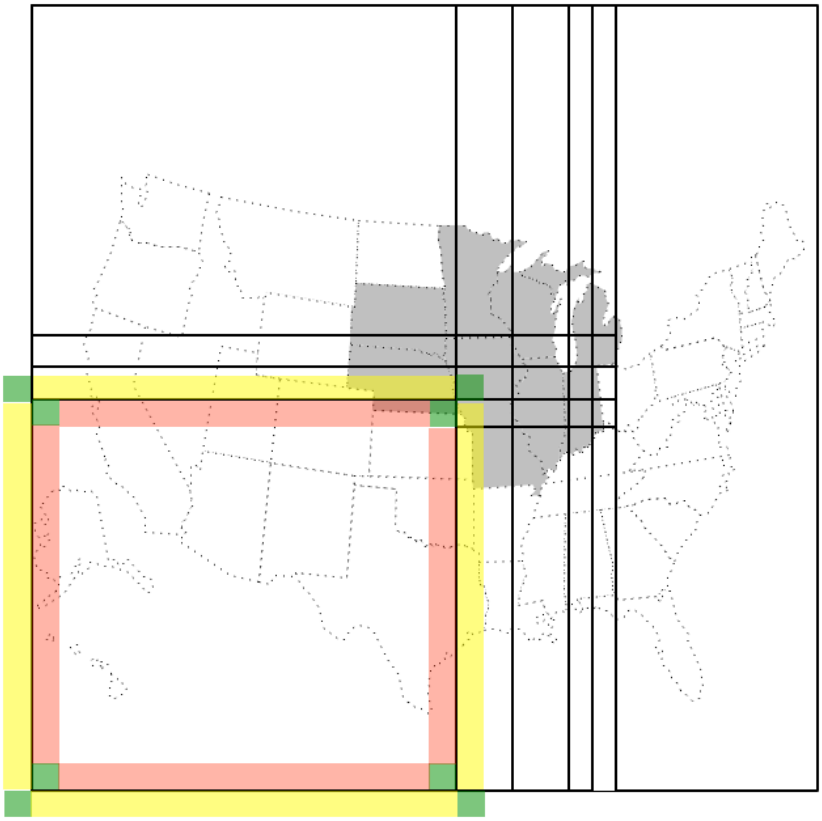

Fig. 2.7.: Non-uniform field partitioning with 25 LPs.

Similarly the granularity of the world decomposition (that is, the cell size and, consequently, the number of cells, which a given space is partitioned into) determines a trade–off between load balancing and communication overhead. The finer is the granularity adopted, the higher is the balancing that, ideally, can be reached by the system. However, due to agents' interdependency and system synchronizations, finer granularity usually determines a huge amount of communication which may harm the overall performances.

D-MASON uses a simple but efficient technique to cope with heterogeneity. The idea is to clone the software ran by high capable workers so that they could serve as multiple workers; i.e., a worker that is x times more powerful than other workers could execute x virtual workers (that is by simulating, concurrently, several cells). D-MASON uses a static partitioning while the granularity of the decomposition is chosen by the user according to the expected unbalancing of the model and the performances of the communication layer.

Communication

D-MASON LPs communicate via a well-known mechanism, based on the Publish/-Subscribe (P/S) design pattern: a multicast channel is assigned to each cell; LPs

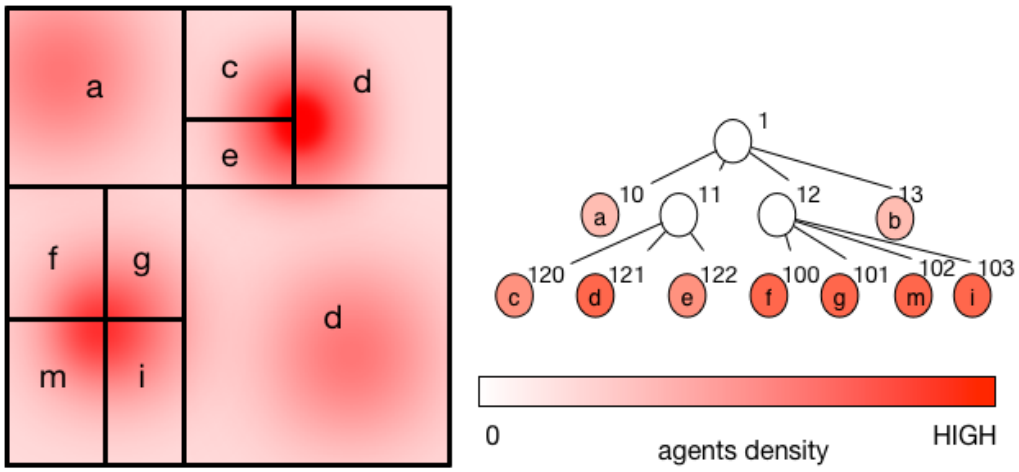

Fig. 2.8.: Non-uniform field partitioning with 9 LPs and the associated decomposition tree. The color map describes the agents density on the field.

then simply subscribe to the topics associated with the cells which overlap with their Area of Interest (AOI) in order to receive relevant message updates. Other topics are also used for system management and visualization.

The first version of D-MASON uses Java Message Service (JMS) for communication between LPs, by running Apache ActiveMQ Server [Apa11] as JMS provider. These functionalities are provided by the Communication Layer of D-MASON described in the section 2.4.2.

D-MASON, however, is designed to be used with *any* Message Oriented Middleware that implements the PS pattern. By providing a mapping between the abstract D-MASON's mechanism and the concrete implementation, it is possible, for instance, to use Scribe [Cas+02], a fully decentralized application-layer multicast built on top of the DHT Pastry [DR01]. Even simpler communication protocols (such as sockets, Remote Method Invocation, etc.) can be used but the effort of the programmer will be more consistent, since a mapping between a semantically rich paradigm, such as the Publish/Subscribe design pattern, and a simpler communication mechanism (stream, remote invocations, etc.) is needed.

D-MASON can also be deployed on HPC systems. In order to better exploit such homogeneous environments D-MASON provides also an MPI-based Publish/Subscribe communication (see Section 2.4.2).

Synchronization

As described above, D-MASON uses a space partitioning model where each LP maintains a portion of the simulated space (cell), and is responsible for the simulation of agents belonging to such cell. In order to guarantee the consistency of

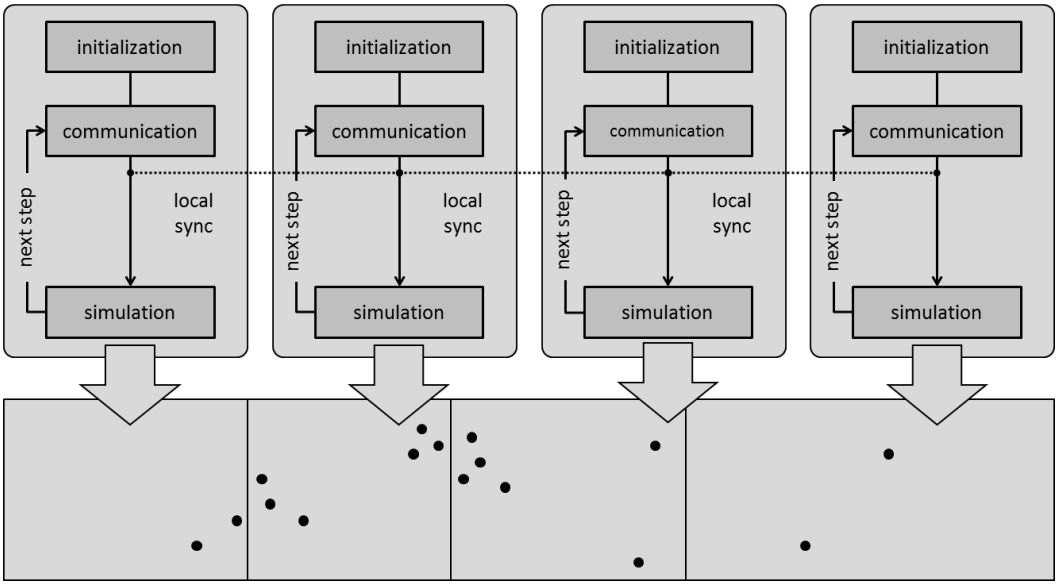

Fig. 2.9.: D-MASON LPs' synchronization.

parallel implementation with respect to the sequential one, each LP needs to collect information about the adjacent cells. Each simulation step is formed by two phases: *communication/synchronization* and *simulation* (see Figure 2.9). First of all, the LP w sends to its neighbors (i.e., the LPs responsible for its adjacent cells) the information about:

- the agents that are migrating to them;
- the agents that may fall into the AOI of the agents in the neighborhood of w.

This information exchange is locally synchronized in order to let the simulation run consistently (see Figure 2.9).

D-MASON uses a conservative–synchronization approach to achieve a consistent integration of the distributed simulations: each step is associated with a fixed state of the simulation. Cells are simulated step by step. Since the step t of cell r is computed by using the states $t-1$ of r's neighborhood, the step t of a cell cannot be executed until the states $t-1$ of its neighborhood have been computed and delivered. In other words, before each simulation steps, the state of each cell is synchronized with its adjacent cells.

Reproducibility

In order to guarantee an easy parallelization and to assure the reproducibility of results, paramount objective of the research areas interested in the ABMs, it is important to design the simulation in such a way that agents evolve simultaneously. In other words, considering a simulation as an iterative process where, at each step, each agent updates its state, in the simultaneous model each agent computes

its state at step i based on the state of its neighbors at step $i-1$. Thereafter all the agents updates their state simultaneously. Using this approach the simulation become embarrassingly parallelizable (there are no dependencies between agents' state), each simulation step can be executed in parallel overall the agents. Moreover, the order in which agents are scheduled does not affect the reproducibility of results. Some simulations, especially those that evolve using a randomized approach, still require a mechanism that allows scheduling agents always in the same order.

When the behavior of a simulations is not deterministic (that is, it is influenced by a random component) the reproducibility of results become hard to achieve in a distributed environment. To achieve reproducibility, each cell must use its own copy of the random source (e.g. `MersenneTwisterFaster`) and the scheduler must ensure that the agents are always elaborated in the same arbitrary order.

2.4 D-MASON Architecture

D-MASON is designed under the scalability principle in all of its forms. D-MASON architecture provides five design requirements and is divided in layers. The design requirements are: ***efficiency*** for exploiting hardware architecture, ***effectiveness*** for modelling different kind of ABM, ***usability*** from the users experience point of view and ***correctness*** of the results. The D-MASON function blocks (or layers) and their interaction aim to meet these requirements. The D-MASON functional blocks (or layers) are: *Distributed Simulation*, *Communication*, *Visualization* and *System Management*. Figure 2.10 depicts the layers interactions and how these meet the design requirements.

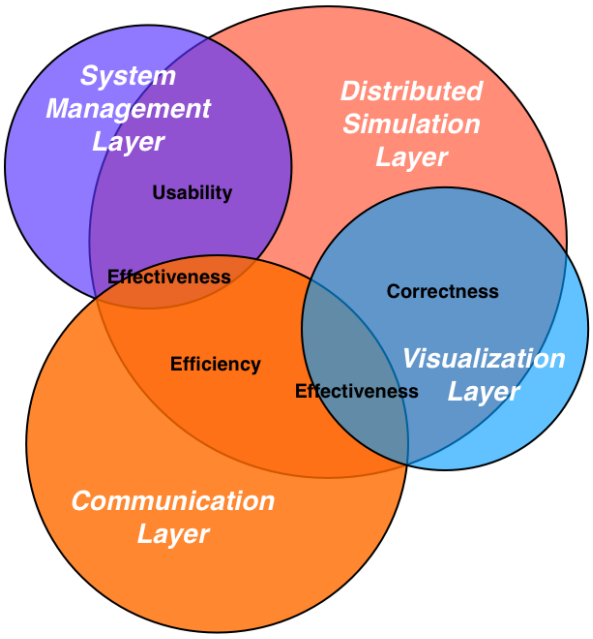

Fig. 2.10.: D-MASON design goals and layers interactions.

Figure 2.11 depicts the D-MASON architecture, showing the corresponding architectural layer for each of the D-MASON component. As shown, Distributed Simulation and Communication layers are dependent on each other, both are needed to build a distributed simulation in D-MASON. But there are no dependencies between the System Management and Visualization layer with other layers, these are independed module of D-MASON.

The next four sections describes the D-MASON layers while the Appendix C provides a comparison between the implementations of a simulation example written in D-MASON and MASON.

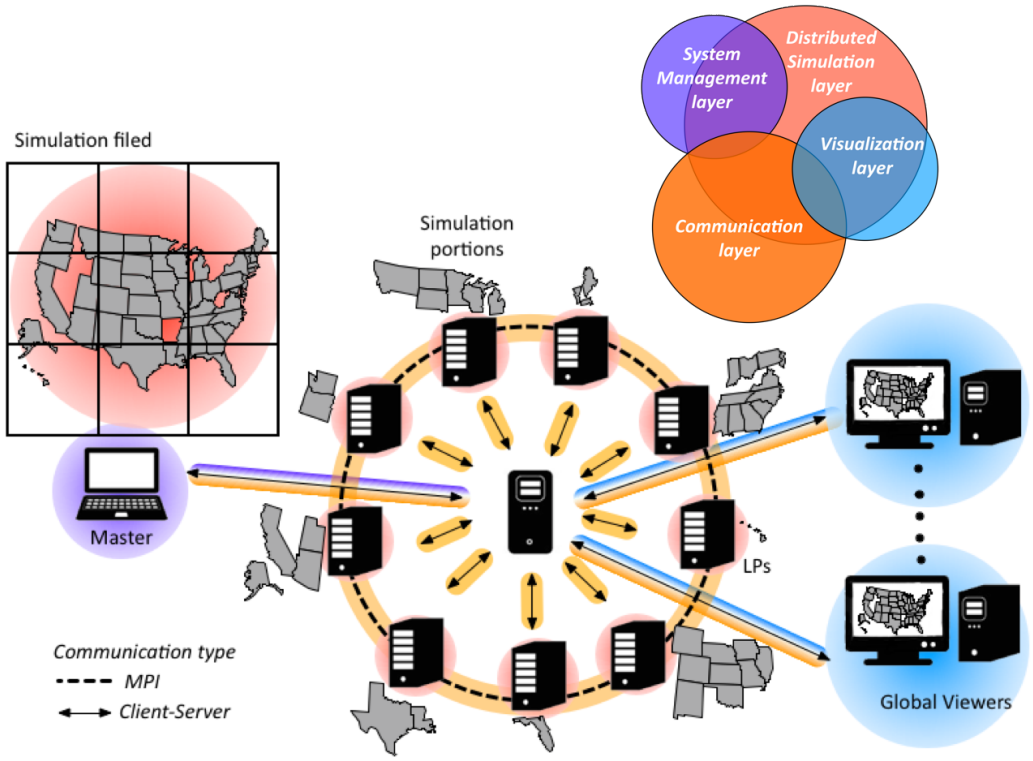

Fig. 2.11.: D-MASON Architecture and Design requirements.

2.4.1 Distributed Simulation Layer

D-MASON uses a framework-level solution to deal with the scalability issues earlier discussed in this thesis. The *Distributed Simulation (DS)* layer adds some features to the simulation layer that allows the distribution of the simulation work on multiple, even heterogeneous machines. Notice that the new layer does not alter in any way the existing MASON layers. Moreover, it has been designed so as to enable the porting of existing applications on distributed platforms in a transparent and easy way.

D-MASON DS layer consists of two main packages: *Engine* and *Field* (see Figure 2.12). This choice is dictated by choice to maintain the same structure as MASON in order to provide the developer with a friendly environment.

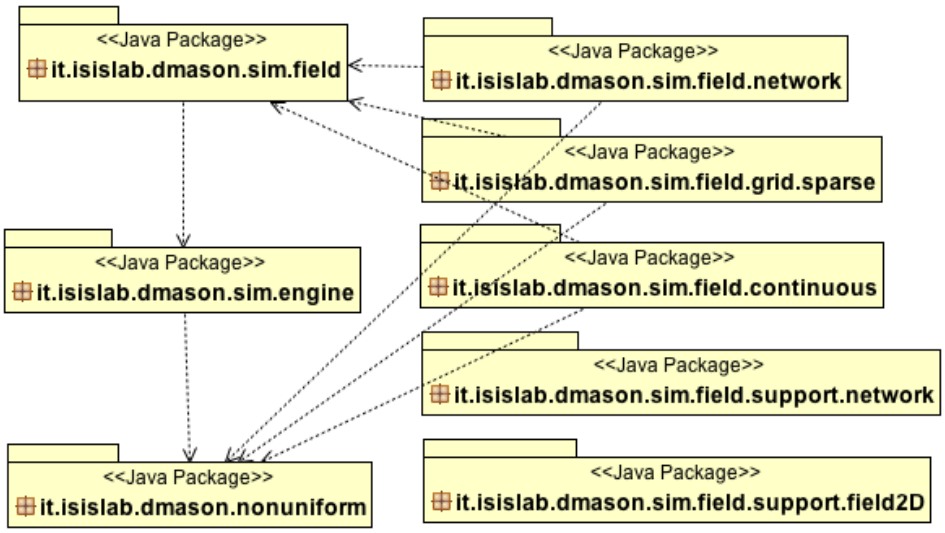

Fig. 2.12.: D-MASON Distributed Simulation Layer Packages

Engine

The *Engine* package, depicted in Figure 2.13, consists of three objects, each one being a distributed version of a correspondent one in MASON. The first, `DistributedState`, represents the state of the simulation in a distributed environment and includes:

- a method to create a reproducible sequence of identifiers to be assigned to agents created by each cell of the simulation;
- a cell identifier;
- a method to retrieve the implementation of the specific field to be used by the simulation;
- a method to add agents in the cell (this method enables the migration of agents between cells).

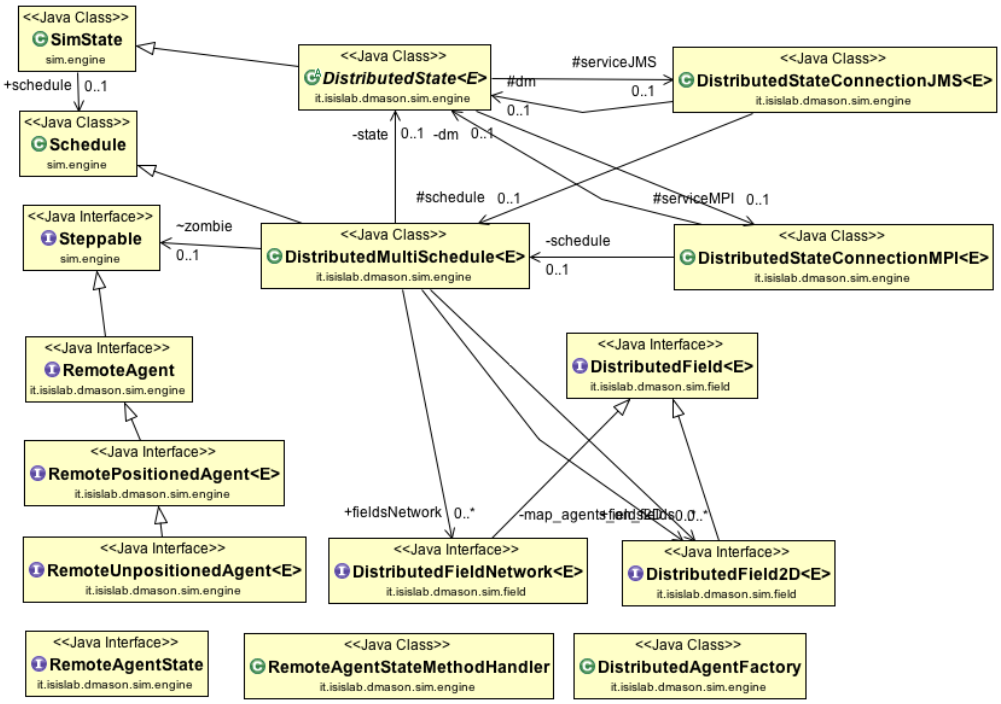

Fig. 2.13.: D-MASON Distributed Simulation Layer Core Classes

The second object, `DistributedMultiSchedule`, represents the time of distributed simulation and allows the self-synchronization between successive simulation steps. This object also adds a "zombie" step-able object to the schedule. This object, which does not perform any action, is required to avoid the end of the simulation when a cell become empty. Indeed, MASON ends the simulation when, during a step, the schedule queue is empty.

The last object is `RemoteAgent`, which represents the abstraction of a distributed object `Steppable`. This object is parametrized according to the kind of distributed field used by the simulation. RemoteAgent provides also a unique identifier for the agent across the system. The unique identifier is required because agents can migrate from one cell to another.

These three objects are the core of D-MASON. Listing 2.1 depicts a basic example of a `DistributedState`. This toy simulation uses a `DContinuousGrid2D` (described in the following), a continuous space field corresponding to the `Continuous2D` field of MASON. The simulation initializes 100 agents for each LP, and sets their positions randomly on the field.

Listing 2.1: DSimulation

```
1 public class DSimulation extends DistributedState<Double2D>
2 {
3     public DContinuousGrid2D sim_field;
4     public DSimulation(GeneralParam params, String prefix)
5     {
```

```
6         super(params,new DistributedMultiSchedule<Double2D>(),prefix,
              params.getConnectionType());
7     }
8     @Override
9     public void start()
10    {
11
12        super.start();
13        try
14        {\\Tua.
15            sim_field =
                  DContinuousGrid2DFactory.createDContinuous2D(10.0/1.5,200,
                  200,this,
16                    super.AOI,TYPE.pos_i,TYPE.pos_j,
                          super.rows,super.columns,MODE,"dfield1",
                          topicPrefix,true);
17            init_connection();
18        } catch (DMasonException e) { e.printStackTrace(); }
19
20        DAgent agent = null;
21        for (int i = 0; i < 100; i++) {
22             agent=new DAgent(this,new Double2D(0,0),
                   this.random.nextInt());
23             agent.setPos(sim_field.getAvailableRandomLocation());
24             sim_field.setObjectLocation(agent, agent.pos);
25             agent.setColor(Color.RED);
26             schedule.scheduleOnce(sim_field);
27        }
28
29    }
30    @Override
31    public DistributedField2D getField()
32    {
33        return sim_field;
34    }
35    @Override
36    public void addToField(RemotePositionedAgent rm, Double2D loc)
37    {
38        sim_field.setObjectLocation(rm,loc);
39
40    }
41    @Override
42    public SimState getState()
43    {
44        return this;
45    }
46 }
```

Listing 2.2 depicts the code for the toy agent. An agent is build by two object: `RemoteDAgent` and its real implementation `DAgent`. `RemoteDAgent` is and abstract class that implements `RemoteUnpositionedAgent` or `RemotePositionedAgent`.

These two objects are a subclass of RemoteAgent and are, respectively, an instance of an agent that has a position in the space (e.g. an agent in a continuous 2D space) and an instance of an agent without any positioning (e.g. an agent in a Network).

This hierarchy is necessary to exploit all MASON features. Indeed, some MASON features, like some type of agents visualization, are obtained by extending a visualization class. On the other hand, in D-MASON the agent class should extend a `RemoteAgent` class, while unfortunately Java does not support multiple inheritance. This reasoning justify the hierarchy described above and ensures the compatibility with all MASON functionalities.

Listing 2.2: DAgent

```
1  //Abstract based agent class
2  public abstract class RemoteDAgent<E> implements Serializable,
       RemotePositionedAgent<E> {
3
4     private static final long serialVersionUID = 1L;
5     public E pos;      // Location of agents
6     public String id; //id remote agent.An id uniquely identifies the
          agent in the distributed-field
7     public RemoteDAgent() {}
8     public RemoteDAgent(DistributedState<E> state){
9        int i=state.nextId();
10       this.id=state.getType().toString()+"-"+i;
11    }
12    @Override
13       public E getPos() { return pos; }
14       @Override
15       public void setPos(E pos) { this.pos = pos; }
16       @Override
17       public String getId() {return id;}
18       @Override
19    public void setId(String id) {this.id = id;}
20    @Override
21    public boolean equals(Object obj) {
22       if (this == obj) return true;
23       if (obj == null) return false;
24       if (getClass() != obj.getClass()) return false;
25       RemoteFlock other = (RemoteFlock) obj;
26       if (id == null) {
27          if (other.id != null) return false;
28       } else if (!id.equals(other.id)) return false;
29       if (pos == null) {
30          if (other.pos != null) return false;
31       } else if (!pos.equals(other.pos)) return false;
```

```
32          return true;
33      }
34  }
35  //Agent class
36  public class DAgent  extends RemoteDAgent<Double2D>{
37      private int val;
38      public DAgent(){}  // Required for D-MASON serialization
39      public DAgent(String id, Double2D location, Integer val) {
40          this.id = id;
41          this.pos = location;
42          this.val = val;
43      }
44      public Bag getNeighbors(DistributedState<Double2D> sm)
45          {
46              return ((DContinuousGrid2D)sm.getField()).
47              getNeighborsExactlyWithinDistance(pos,
48              (10, true);
49          }
50      @Override
51      public void step(SimState state) {
52          Bag b = getNeighbors((DistributedState)state);
53          int max=val;
54          for(Object f:b){
55              DAgent d=(DAgent) f;
56              int dval =d.getVal();
57              if(max < dval) max=dval;
58          }
59          this.val = max;
60      }
61      public int getVal(DistributedMultiSchedule schedule){
62          return val;
63      }
64
65  }
```

Listing 2.3 depicts the example code for executing D-MASON simulation on a local machine. This code considers that an instance of the message broker `Apache ActiveMQ` is running on the local machine (see Section Communication Layer). The test initializes and executes 8 LPs (`DSimulation` object), using a uniform partitioning approach.

Listing 2.3: DTestLocalMachine

```
1  public class DTestLocalMachine {
2      private static int numSteps = 100; //number of step
3      private static int rows = 2; //number of rows
4      private static int columns = 4; //number of columns
5      private static int AOI=10; //AOI
6      private static int CONNECTION_TYPE=ConnectionType.pureActiveMQ;
7      private static String ip="127.0.0.1"; //ip of activemq
```

```
8     private static String port="61616"; //port of activemq
9     private static String topicPrefix=""; //unique string to identify
          topics for this simulation
10    private static int MODE =
          DistributedField2D.UNIFORM_PARTITIONING_MODE;
11
12    public static void main(String[] args)
13    {
14       class worker extends Thread
15       {
16          private DistributedState<?> ds;
17          public worker(DistributedState<?> ds) {
18             this.ds=ds;
19             ds.start();
20          }
21          @Override
22          public void run() {
23             int i=0;
24             while(i!=numSteps)
25             {
26                ds.schedule.step(ds);
27                i++;
28             }
29             System.exit(0);
30          }
31       }
32
33       ArrayList<worker> myWorker = new ArrayList<worker>();
34       for (int i = 0; i < rows; i++) {
35          for (int j = 0; j < columns; j++) {
36
37             GeneralParam genParam = new GeneralParam(null, null, AOI,
                   rows, columns, null, MODE, CONNECTION_TYPE);
38             genParam.setI(i);
39             genParam.setJ(j);
40             genParam.setIp(ip);
41             genParam.setPort(port);
42             ArrayList<EntryParam<String, Object>> simParams=new
                   ArrayList<EntryParam<String,Object>>();
43             DSimulation sim = new DSimulation(genParam,
                   simParams,topicPrefix);
44             worker a = new worker(sim);
45             myWorker.add(a);
46          }
47       }
48       for (worker w : myWorker) w.start();
49    }
50 }
```

Engine: Memory Consistency Mechanism (MCM)

As observed in [Hyb+06], ABMs are very expressive; they implement the *sense-think-act* cycle and offer a real agent-based programming model. Unfortunately, due to the high level of interdependencies between agents, these models [Min+96; Luk+05; Nor+07] are not easy to parallelize.

As said above, D-MASON provides a self-synchronized environment where, before starting each simulation step, each LP exchanges some data with neighbors' LPs in order to obtain all the information required to execute the step (no other information is exchanged during the step execution).

Theoretically, all the agents update their status at step t, considering the status of all neighbor agents at step $t-1$ (synchronous update). Specifically, in synchronous updating, the agent v at the t-th iteration updates its state based on the state of its neighbors at the $(t-1)$-th iteration. Suppose agent v has k neighbors $u_1, u_2, \ldots, u_k$. Let $S_v[t]$ be the state of agent v at the t-th iteration. Hence,

$$S_v[t] = f_{upd}(S_{u_1}[t-1], S_{u_2}[t-1], \ldots, S_{u_k}[t-1]),$$

where f_{upd} computes the state of an agent. Using a sequential environment, the agents are updated asynchronously, that is

$$S_v[t]{=}f_{upd}(S_{u_1}[t], \ldots, S_{u_m}[t], S_{u_{m+1}}[t{-}1], \ldots, S_{u_k}[t{-}1]),$$

where $u_1, u_2, \ldots, u_m$ are the neighbors of v that have already performed their computing in the current iteration, while $u_{m+1}, u_{m+2}, \ldots, u_k$ are the neighbors that have not yet performed the computing. Asynchronous execution exhibits a very strong side effect: the behavior of the model is influenced by the order of agents' executions [LM09]. So in order to meet the reproducibility feature (see Section 2.3.2), it is extremely important to update the agents using a deterministic strategy. Indeed, the asynchronous approach, performed on a sequential environment, adds unnecessary dependencies that harm the efficiency of the system.

Asynchronous execution becomes even harder on a distributed environment that exploits the space partitioning approach. Such systems use some ghost agents (which lay on a different cell, but are close enough to influence the behavior of some cell agents). The problem is that ghost agents are never updated but receive the novel state only at the end of the simulation step, so in order to realize an asynchronous update strategy the system has to be synchronized after each agent update.

Although there are several approaches, like *double buffering*, which enable implementing the synchronous approach even in a sequential environment, it is worth mentioning that these strategies

- are usually not straightforward for the model designer (with limited knowledge of distributed programming);
- have their drawbacks either in terms of increased usage of memory or in terms of efficiency, since they need to update the agents' status even when it has not changed. Such strategies have been conceived for massively parallel architectures (like GPUs) where the cost of a `memcopy` is negligible.

MCM Strategy. D-MASON provides a *memory consistency* mechanism which, during the t-th simulation step, ensures that the read of an agent state will return the state at the end of step $t-1$ when the read is performed from an outside agent while it returns the updated status when the read is performed inside the agent. The idea of the mechanism is quite simple. During the simulation step t, the first time an agent state is updated the preceding state (that is the state at the end of step $t-1$) is saved into an hashmap which maps the agent identifier to its state at the end of step $t-1$. Then all the remaining write operations operate normally. The read operation performed from the outside (that is using the `getter` method), first checks whether the hashmap contains an entry for the corresponding agent. In case of success, the agent state is read from the hashmap, otherwise the state has not changed and can be recovered in the standard way. Once the simulation step is completed for all the agents, the hashmap is cleaned so that a unique state is stored. Using this approach the memory overhead is limited because each agent state is stored at most twice and a copy is generated only when necessary.

MCM Implementation Details. Clearly the mechanism described above is quite hard to implement, especially for model designers with limited experience in object oriented programming. In D-MASON this mechanism is easily affordable for every user because D-MASON solves the problem at framework level. A former implementation of the memory consistency mechanism was based on the Byte Code Generation Library (cglib [cgl16]) to inject, at runtime, the code needed to realize the above mechanism. The cglib is high level API to generate and transform JAVA byte code. It is used by Aspect Oriented Programming, testing, data access frameworks to generate dynamic proxy objects and intercept field access. Unfortunately this former implementation introduces a performance slowdown mainly due to Java Reflection overhead (see Figure 2.14).

Another implementation of the memory consistency mechanism is based on Java Method Handles, which, using Java 8, provides better performance than the Java Reflection. The package `engine` provides the class `RemoteAgentStateMethodHandler` that is an object that enable to access "consistently" the state of an agent. The same package also provides the `StateVariable` object, which binds each state variable name with the corresponding type.

Basically, in order to use the memory consistency mechanism of D-MASON, the model's programmer has to:

- define all access methods to the agent state variables;
- select the variables that define the agent state. In order to do that, each state variable must be declared as part of the agent's state. The agent's state is defined in the agent class by a static list of `StateVariable` (see Listing 2.4);
- define a new static object `RemoteAgentStateMethodHandler`, in order to access the agent's state in the step method of the agent.
  The `RemoteAgentStateMethodHandler` provides two methods `getState(...)` and `setState(...)` to get and set the agents' state (see Listing 2.4).

In order to better explain the coding effort needed to use the D-MASON memory consistency mechanism, in the following, we provide a toy example: The state of each agent is represented by an integer. Agents wanders into a geometric field. During each simulation step, each agent read the state of all its neighbors (i.e., the agents within a certain distance) and updates its value as the maximum among its state and the states of all its neighbors. The following code (Listing 2.4) implements the above logic using the memory consistency mechanism. The same logic without MCM is shown in the Listing 2.2.

First is declared the variables, which compose the agents' state, after that it is declared the `RemoteAgentStateMethodHandler` object to access consistently the agents state (see Listing 2.4). Notice that, in the step method, the access to the state variable `val` is always performed using `getState(...)` and `setState(...)` of the `RemoteAgentStateMethodHandler` object. It is worth mentioning that, although the methods `getVal` and `setVal` are implemented in the standard way by the modeler, they will be used by `RemoteAgentStateMethodHandler` for access "consistently" the agents' state.

Listing 2.4: DAgent

```
1  public class DAgent extends RemoteDAgent<Double2D>{
2     private int val;
3     //Agent state definition
4     static ArrayList<StateVariable> statevariables=new
          ArrayList<StateVariable>();
5     static{
6        statevariables.add(new StateVariable("val",int.class));
7     }
8    final static RemoteAgentStateMethodHandler memory = new
        RemoteAgentStateMethodHandler DAgent.class , statevariables);
9    //end Agent state definition
10    public DAgent(){}
11
12    public DAgent(String id, Double2D location, Integer val) {
13       this.id = id;
```

```
14          this.pos = location;
15          this.val = val;
16      }
17
18      @Override
19      public void step(SimState state) {
20          Bag b = getNeighbors((DistributedState)state);
21          int max=val;
22          for(Object f:b){
23              DAgent d=(DAgent) f;
24              int dval = (int) memory. getState(
                    (DistributedMultiSchedule) st.schedule , d , "val");
25              if(max < dval) max=dval;
26          }
27          memory.setState( (DistributedMultiSchedule) st.schedule, this ,
                "val" , max);
28      }
29
30      public int getVal(DistributedMultiSchedule schedule){
31          return val;
32      }
33
34      public void setVal(DistributedMultiSchedule schedule,int val){
35          this.val=val;
36      }
37  }
```

MCM Performance. Four MCM implementations in Java have been tested:

- **Double Buffering (DB),** uses the well know double buffering strategy implemented ad hoc in the agent.
- **Java Reflection (Reflection),** uses the Byte Code Generation Library (cglib [cgl16]) to inject, at runtime, the code change the effect of access methods exploiting the Java Reflection Proxy to intercept all methods calls.
- **Method Handler,** uses the Method Handler mechanism of Java 8. Two kind of binding have been considered:
  - **Dynamic Lookup (MHDL),** where the methods handler, for each access method, are obtained dynamically;
  - **Static Lookup (MHSL),** where the methods handler are bound in a static context, as show in Listing 2.4.

The figure 2.14 depicts the performance obtained by different memory consistency mechanisms.

The different implementations were tested on a cluster machine of 16 nodes. Each node is a Linux machine configured with 24 cores, 32 GB of RAM and Java Virtual Machine 1.8. The test simulation used was a naive SIR model implementation. Each agents wander on a bi-dimensional field (of size 4000×4000) exchanging

an information among neighboring agents. Each test has been performed with $4, 16, 36, 64$ and 144 LPs and 500000 agents for 100 simulation steps. The overall completion time have been collected and depicted in Figure 2.14. The figure shows that the worst performance is achieved by the Reflection implementation due to the overhead introduced by the Java Reflection mechanism. On the other hand, the MHSL implementation provides almost the same performance of the Double Buffering implementation, which as discussed above, requires strong programming skills and heavily impacts on the usage of memory.

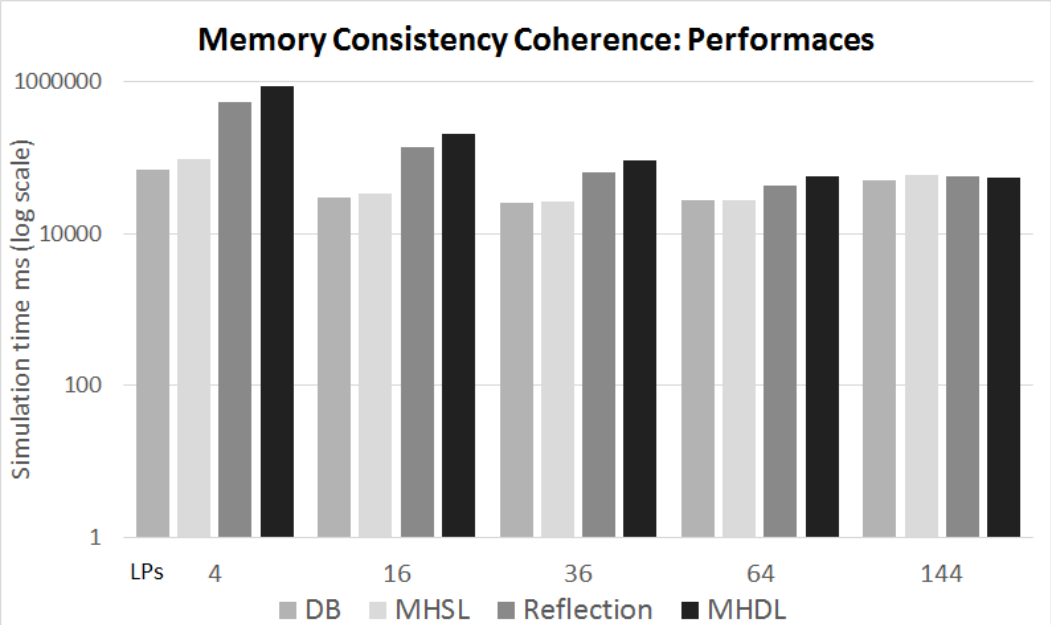

Fig. 2.14.: Memory Consistency Performances: Simulation time obtained running a SIR simulation using four different memory consistency implementations with $4, 16, 36, 64$ and 144 LPs.

Field

The package *Field*, depicted in Figure 2.13, is the real core of D-MASON: indeed this package defines the logic of agents' distribution. This package provides classes in common with all the fields and the distributed versions of several MASON fields.

The new hierarchy of fields in Distributed MASON is based on a Java interface called `DistributedField` that represents an abstraction of a distributed field (cell): all the fields that are meant to be distributed must implements this interface, so that they have to expose functionalities, such as evaluating whether a given position belongs to the current cell or not or a method to generate a random position in the current cell. The interface also exposes a method `synchro()` that allows the local synchronization among cells and hence the update of the simulation (see Figure 2.9).

D-MASON local synchronization is made through the object `UpdateMap` that asynchronously receives messages by neighbors cells. When all the messages, related to the current step, are received, the synchronization is complete and the cell is ready to perform another simulation step. Each update message, represented by the class `DistributedRegion`, is composed by two lists of agents: one represents the agents that are migrating to the cell that receive the message; and one that contains the agents, aka ghost agents, that may fall into the AOI of an agent in the receiving cell. The agents, which are migrated to the cell, need to be scheduled for the next simulation step, while ghost agents will be used to compute the behavior of simulated agent (even if they are not simulated). In order to assure the reproducibility of results, agents are scheduled using a priority queue which does not depend on the order in which messages are received.

D-MASON provides a distributed version for almost all the MASON fields. For instance, the sub-packages `grid` and `continuous` provide two specializations of the 2D fields of MASON, `SparseGrid2D`, `Continuos2D`, `IntGrid2D` and `DoubleGrid2D`, two types of class factory to create the right field according to the type of partitioning (e.g., `DSparseGrid2DXY` and `DContinuosGrid2DXY` for uniform partitioning and `DSparseGridNonUniform` and `DContinuosGridNonUniform` for non-uniform partitioning) and the corresponding abstract classes for the fields, `DSparseGrid2D` e `DContinuousGrid2D`.

As anticipated in the Section 2.3.2, currently, in D-MASON there are two kind of field partitioning: *uniform* and *non-uniform*. Listing 2.1 shown the code of a toy simulation. Line 15 shows the code for instantiating a `DContinuous2D` field using the factory `DContinuousGrid2DFactory` that instantiates a new distributed field using the given constructions parameters. This example uses the method `createDContinuonus2D` which performs a uniform partitioning. On the other hand, the method `createDContinuous2DNonUniform` can be used to perform a non-uniform partitioning.

Field: Distributed Network Field

Unfortunately, the basic field partitioning approach is inappropriate and inefficient when the simulation field is not an Euclidean space (e.g., a Network field). In such a case, the partitioning strategies cannot use space information, but should instead be based on the relationship among the agents, described in forms of a communication graph. D-MASON provides the `DNetwork` field, that is the distributed version of MASON `Network` field. The `DNetwork` field exploits a different partitioning strategy that will be discussed in details within Appendix A.

Why Networks are important in ABM simulation? Networks are everywhere. Complex interactions between different entities play an important role in modeling the

behavior of both society and the natural world. Such networks – which comprise the World Wide Web, metabolic networks, neural networks, communication and collaboration networks, and social networks – are the subject of a growing number of research efforts. Indeed, many interesting phenomena are structured as networks (i.e., sets of entities – aka nodes – joined in pairs by lines – aka edges – representing relations or interactions).

The study of networked phenomena has experienced a particular surge of interest due to the increasing availability of massive data about the static topology of real networks, as well as the dynamic behavior generated by the interactions among network entities. The analysis of real networks topologies has revealed several interesting structural features, like the small-world phenomena, as well as the power-law degree distribution [EK10], which appear in several real networks and can be extremely helpful for the design of artificial networks. On the other hand, understanding the dynamic behavior generated by complex network systems is extremely hard. Networks are often characterized by a dynamic feedback effect which is hard to predict analytically.

DNetwork field. The field partitioning approach, implemented on D-MASON, and described in the previous Section, is devoted to decomposing ABMs based on geometric fields. On the other hand, when agents lie and/or interact on a network [CH05] – where the network can represent social, geographical or even a semantic space – a different approach is needed. The problem is to (dynamically) partition the network into a fixed set of sub-networks in such a way that:

- the components have roughly the same size;
- both the number of connections and the communication volume between nodes belonging to different components are minimized (see Figure 2.15).

This problem is well known in literature as the graph partitioning problem. It has been extensively studied (see [Bad+13] for a comprehensive presentation) and is known to be NP-hard [GJ90]. Being a hard problem, exact solutions are found in reasonable time only for small networks. However the applications of this problem require partitioning much larger networks. For this reason, several heuristic solutions have been proposed in literature and are discussed, in the context of ABM simulations, in the Appendix A.

D-MASON provides a distributed network field, named *DNetwork* (see Figure 2.15), based on METIS [KK98], a graph multilevel k-way partitioning suite, developed in the Karypis lab of University of Minnesota, evaluated for our specific purpose in [Ant+15].

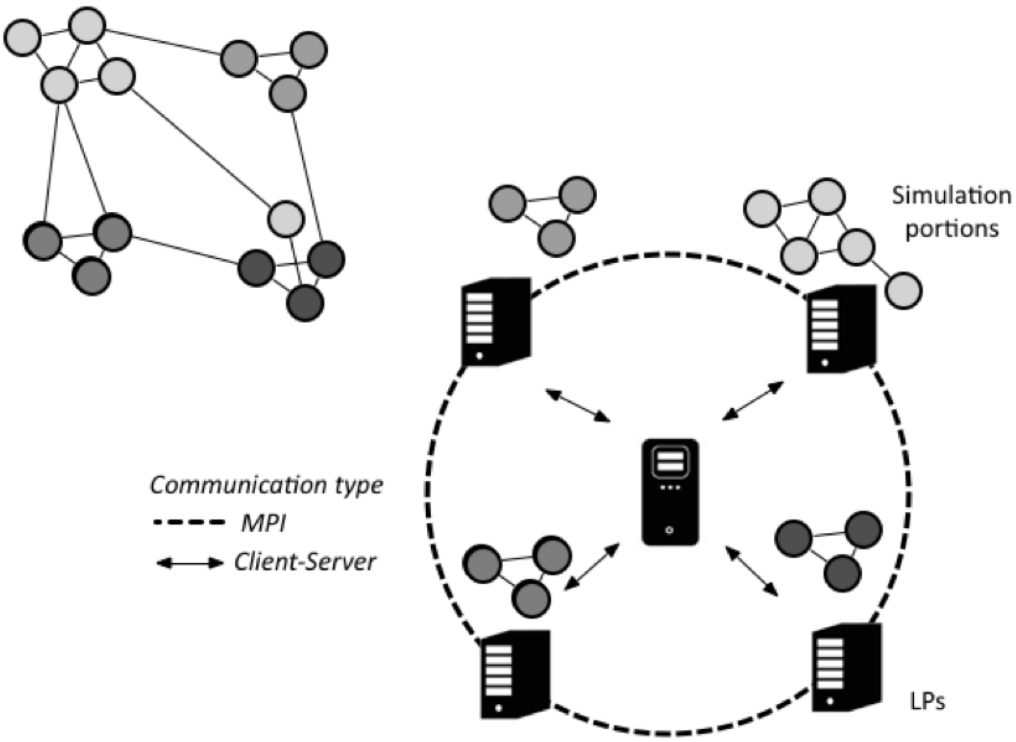

Fig. 2.15.: DNetwork Field D-MASON.

DNetwork field: Usage. The following Listing 2.5 shows the code of the `start` method for an D-MASON simulation on the *DNetwork* field. For visualization purposes, the agents also lie on a `Continuous2D` field.

The method *PartitionManager.getNetworkPartition(graph_path, graph_parts_path)* loads the network and its partitioning (obtained through any k-way partitioning algorithm) and computes the communication overlay. Then the distributed network is generated using the factory `DNetworkFactory`, which takes as parameter the communication overlay structure. Thereafter all the agents, that belong to the part assigned to the current LP, are added to the fields (`DNetwork` and `Continuous2D`) and to the schedule. Finally, the edges among the agents are created. During this phase some ghost agents (i.e., agents that belong to a different part but are connected to at least one agent already in the field) are added to the field (but not scheduled).

Listing 2.5: DNetwork usage

```
1 public DNetwork network;
2 public Continuous2D yard = new Continuous2D(1.0,100,100);
3
4 @Override
5 public void start()
6 {
7     super.start();
8     int commID = (TYPE.pos_i*rows)+TYPE.pos_j;
```

```
9   NetworkPartition parts_data=PartitionManager.getNetworkPartition(
        graph_path, graph_parts_path);
10  network = DNetworkFactory.createDNetworkField(this, super.rows,
        super.columns, TYPE.pos_i, TYPE.pos_j,
        parts_data.getEdges_subscriber_lsit(),"mygraph", topicPrefix);
11  HashMap<Integer, Agent> myAgents = new HashMap<Integer,Agent>();
12  HashMap<Integer, Agent> myGhostAgents = new
        HashMap<Integer,Agent>();
13  SuperVertex my_vertices=parts_data.getParts2SuperGraph().get(
        commID);
14  for(Vertex v: my_vertices.getOriginal_vertex()){
15     Double2D pos=new Double2D(yard.getWidth() *
           random.nextDouble(), yard.getHeight() *
           random.nextDouble());
16     Agent netSource = (Agent) DistributedAgentFactory.newIstance(
           Agent.class, new Class[]{ SimState.class, Integer.class,
           Boolean.class, String.class, Double2D.class}, new
           Object[]{this, my_vertices.getId(), false, v.getId()+"",
           pos}, DVertexState15.class);
17     myAgents.put(v.getId(), netSource);
18     schedule.scheduleOnce(netSource);
19     network.addNode(netSource);
20     yard.setObjectLocation(netSource, pos);
21  }
22  for (Edge e : parts_data.getOriginal_graph().edgeSet()){
23     Vertex source = (Vertex)  parts_data.
           getOriginal_graph().getEdgeSource(e);
24     Vertex target = (Vertex)  parts_data.
           getOriginal_graph().getEdgeTarget(e);
25     Integer sourceComm =  parts_data.getGraph2parts().get(source);
26     Integer targetComm = parts_data.getGraph2parts().get(target);
27     Agent netSource = myAgents.get(source.getId());
28     Agent netTarget = myAgents.get(target.getId());
29     if(netSource==null && netTarget==null) continue;
30     if(netSource==null &&  myGhostAgents.get(source.getId())==null){
31        Double2D pos=new Double2D(yard.getWidth() *
              random.nextDouble(), yard.getHeight() *
              random.nextDouble());
32        netSource = (Agent) DistributedAgentFactory.newIstance(
              Agent.class, new Class[]{ SimState.class, Integer.class,
              Boolean.class, String.class, Double2D.class}, new
              Object[]{this, sourceComm, false, source.getId()+"",
              pos}, DVertexState15.class);
33        myGhostAgents.put(source.getId(), netSource);
34        network.addNode(netSource);
35        yard.setObjectLocation(netSource, pos);
36        }
37     if(netTarget==null &&  myGhostAgents.get(target.getId())==null){
38        Double2D pos=new Double2D(yard.getWidth() *
              random.nextDouble(), yard.getHeight() *
              random.nextDouble());
```

```
39            netTarget = (Agent) DistributedAgentFactory.newIstance(
                  Agent.class, new Class[]{ SimState.class, Integer.class,
                  Boolean.class, String.class, Double2D.class}, new
                  Object[]{this, targetComm, false, target.getId()+"",
                  pos},DVertexState15.class);
40            myGhostAgents.put(target.getId(), netTarget);
41            network.addNode(netTarget);
42            yard.setObjectLocation(netTarget, pos);
43        }
44        network.addEdge(netSource, netTarget, null);
45    }
46    init_connection();
47 }
```

2.4.2 Communication Layer

The package `Util.Connection`, depicted in Figure 2.16, contains the interface *Connection* (see Listing 2.6) which defines the API of the D-MASON Publish/Subscribe pattern used. D-MASON's architecture enables the customization of the communication mechanism via the specialization of the interface *Connection*, so that different communication mechanisms can be used.

Listing 2.6: Connection

```
1  public interface Connection {
2      public boolean setupConnection(Address providerAddr) throws
           Exception;
3
4      public boolean createTopic(String topicName, int numFields) throws
           Exception;
5
6      public boolean publishToTopic(Serializable object, String
           topicName, String key) throws Exception;
7
8      public boolean subscribeToTopic(String topicName) throws Exception;
9
10     public boolean asynchronousReceive(String key);
11
12     public ArrayList<String> getTopicList() throws Exception;
13
14     public boolean unsubscribe(String topicName) throws Exception;
15 }
```

Considering the good results obtained by the first version of D-MASON, which was mainly devoted to heterogeneous clusters of workstations with a limited number of LPs, the focus moved to dedicated installations, such as massively parallel machines or supercomputing centers. These platforms usually offer a large number of homogeneous machines that, on one hand, simplify the issue of balancing the load among LPs, but, on the other hand, the considerable computational power provided by the system weakens the efficiency of the centralized communication server. Indeed, centralized solutions can not scale both in terms of the growth of the computing power, which affects the amount of communication, and in terms of the number LPs, which affects the number of communication channels. For this reasons, a novel decentralized communication mechanism, which realizes a Publish/Subscribe paradigm through a layer based on the MPI standard, was implemented in D-MASON [Cor+14b].

The current version of D-MASON, as anticipated in the Section 2.3.2, provides two kind of communication specialization. A centralized communication, which is used for general purposes architecture (heterogeneous computing or cloud computing), exploits the Java Message Service standard, exposed by the *ConnectionJMS* interface

and implemented in *ConnectionWithActiveMQ*. The Server side is represented by Apabibte che ActiveMQ Server [Apa11]. The decentralized communication, designed mainly for homogeneous computing (such Extreme-Scale computing), is based on the Message Passing Interface (MPI) [MPI16]. The `ConnectionMPI` interface defines this kind of communication which has been implemented in three different versions: `ConnectionMPIWithBcastMPIBYTE`, `ConnectionMPIWithGatherMPIBYTE` and `ConnectionMPIWithParallelMPIBYTE`. These three version are described in details in the Section 2.4.2.

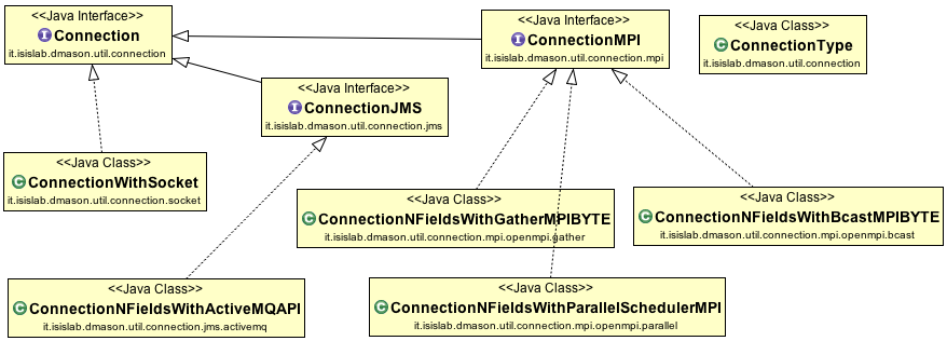

Fig. 2.16.: D-MASON Communication Layer: Core Classes

The functionalities of the layer for homogeneous computing were tested on the OpenMPI [Ope16c] implementation. Using MPI, the overall communication is completely decentralized. Moreover, when the system requires some system management functionalities, D-MASON communication is performed using a hybrid approach: the communication among LPs is handled by the MPI infrastructure (in order to achieve scalability) while the management messages, being asynchronous, operate through the ActiveMQ Server.

Connection Types. In order to manage different communication types, D-MASON provides a class, named `ConnectionTypes`, that exposes the macros for all the available communication strategies (see Listing 2.7). The following strategies are currently available:

1. `pureActiveMQ`: uses *Apache ActiveMQ* as message broker for both the management and LPs' synchronization messages;
2. `pureMPI`, uses only the MPI layer and can be used only when there are no management services (like centralized visualization). Three different MPI communication mechanisms are available [Cor+14b; Cor+14a]:
   - `pureMPIBcast`, exploits a broadcasting mechanism between the LPs groups (i.e., LPs managing adjacent cells);
   - `pureMPIGather`, using the function `MPI Gather` an LP is able to get, in a single step, all the information needed from its neighborhood. This strategy allows to decrease the number of communication rounds but increases the message size.

- `pureMPIParallel`, tries to maximize the degree fo parallelism during the communications between different LPs in order to reduce the number of communication rounds; this approach is based on a randomized graph coloring algorithm. It is highly recommended for simulations having a large number of LPs.

3. `hybridActiveMQMPI`, uses Apache ActiveMQ for the communication between LPs and the system management or the visualization components, while it uses MPI for simulation updates between LPs. Also in this case, it is possible to choose the desired MPI communication mechanism.

Listing 2.7: ConnectionType

```
1  public final class ConnectionType implements Serializable{
2
3      public static final int unitTestJMS = -5;
4
5      public static final int pureMPIBcast = 1;
6      public static final int pureMPIGather = 2;
7      public static final int pureMPIParallel = 3;
8
9      public static final int pureActiveMQ = 0;
10
11     public static final int hybridActiveMQMPIBcast = -1;
12     public static final int hybridActiveMQMPIGather = -2;
13     public static final int hybridActiveMQMPIParallel = -3;
14
15     private static int ConnectionType = pureActiveMQ;
16 }
```

D-MASON Decentralized Communication

D-MASON provides the communication layer for Extreme-Scale computing using MPI. Obviously MPI does not allow directly Publish/Subscribe functionalities so we had to manage them in a different layer according to the Communication interface of D-MASON which exposes some routines to publish and receive messages on specific topics.

The design strategy that enables to use MPI in D-MASON is based on the concept of *synchronization time* that is the time required to exchange all the synchronization messages at the end of a simulation step. The system associates an MPI Process to each D-MASON LP and uses MPI *communication groups* to define the communication among cells. The *synchronization time* is partitioned in communication rounds, each of which is dedicated to a different communication group.

MPI. MPI is a library specification for message-passing, designed for high performance on both massively parallel machines and on workstation clusters. MPI has

emerged as one of the primary programming paradigms for writing efficient parallel applications, it provides point-to-point and collective communications and guarantees portability with all platforms compliant with the MPI Standard. MPI provides several collective operations which are very important because they sustain very high parallel speed-ups for parallel applications[MPI13]. Two solutions for using MPI in D-MASON have been considered: a Java implementation of MPI and a Java binding of MPI.

The need for thread safety. Since the architecture of D-MASON uses a thread for each operation (send or receive) on the same cell, during the choice of the MPI implementation, it is important to consider thread safety implementations.

D-MASON uses a programming model that is a mixture of message passing and multithreading, hence an MPI process has to include multiple threads making MPI calls. An MPI process together with its threads is uniquely identified by the MPI layer, so when an MPI process, handling one or more threads, receives a message, it will be received by all the threads of that process.

Before MPI-2 there was no solution to enable users to write user-level threads, in MPI programs, able to make calls to MPI. The MPI-2 Standard has clearly defined the interaction between MPI and user-level threads in MPI programs. There are four levels of thread safety supported by the standard, that a user must explicitly select according to his/her necessities [GT07].

In the literature there are several Java implementations of the MPI standard [Wei+00; Haf+11; Bak+06]. MPJ Express is an open-source library for message passing compliant with MPI-1.1, offering the same thread-safety guarantees of MPI-2. The comparison between MPJ Express and other MPI implementations is reported in [exp13]. Open MPI Project [Ope13] is an open source MPI-2 Standard implementation that is developed and maintained by a consortium of academic, research and industry partners. Combining the expertise, technologies, resources from all across the *HPC* community, it offers the best MPI library available. The *prerelease 1.9a1r27886* of Open MPI and the current *trunk* on Open MPI'SVN [Ope13] offers *mpiJava*, a Java binding of MPI-1.1 Standard [Bak+99; Car+99].

D-MASON uses *mpiJava* as MPI implementation, because it provides the best performance, even though it offers no guarantee of thread safety in the version *relase 1.9*. The current release 2.0 of *mpiJava* in *OpenMPI* supports full MPI-3.0 Standard, and is declared to be thread safe. Experiments on a communication strategies that uses multiple receiving threads on each LPs is object of future work.

Group Communication. The communication model in D-MASON is potentially $n - to - n$, that means that each node of the network may need to communicate with all others. We chose the Publish/Subscribe paradigm to meet the requirements of flexibility and scalability of the system.

The collective communications offered by MPI allows to make a series of point-to-point communications in one single call. MPI processes can be grouped and managed by an object called *Communicator* [Mes97].

The group communication routines we used are:

- `MPI_Bcast`: sends the same data to all processes in a communicator;
- `MPI_Gather`: collects data from many processes to one single process, the root process, allowing a group to get all needed information in a single step.

Significant research has been carried out in the past for improving collective communication using parallel algorithms based on the message size and optimized for the specific platform. As an example, the function `MPI_Bcast` for small messages uses binomial tree algorithms, for shared memory systems employs pipelining to improve buffer utilization [Mam+]

D-MASON Decentralized Communication: Architecture. The Communication Layer of D-MASON exploits the flexibility of the Publish/Subscribe paradigm to virtualize groups of communication between the agents. In the distributed simulation these groups communicate at the end of each simulation step.

Two states of the computation are present in D-MASON in which all the processes are performing the same action: the connection with the Publish/Subscribe server, because after this phase all workers are connected in the network and the master can see them and spread the computation; the end of each simulation step because at this point of the simulation, each cell has to receive and send updates to start the next step. The purpose of the strategy is to create an MPI infrastructure that abstracts the Publish/Subscribe pattern used in D-MASON, ensuring the decoupling between topics and MPI processes and the scalability of the number of topics by exploiting the collective communication primitives offered by the MPI Standard.

The communication layer using MPI is based on the following set of assumptions:

1. each cell creates its own topic to publish its updates;
2. each cell subscribes to at least one topic; a cell can not start a new simulation step until it receives updates from all the topics it is subscribed to;
3. subscription is static: each cell executes the subscription routines only before starting the simulation;
4. the simulation and communication/synchronization phase do not interleave. The publish is invoked only at the end of the simulation step when all the updates for each field are available.

Each cell is associated with MPI process and a topic is represented with an MPI Group. So in order to create a topic, an MPI process creates an MPI Group and a subscription corresponds to a MPI Group join.

Using the centralized communication, the initialization of the communication can happen before the creation and subscription of topics. On the other hand, using the MPI approach, the creation and subscription of topics has to be performed before any communication.

The assumptions 2 and 3 are exploited in order to perform a global communication phase to exchange information of topics among the MPI processes. This phase is divided in n rounds: in the i-th round the process with *rank* i sends information about the topic created and the topic which it is subscribed to. The other processes receive this information and update their local list of topics. At the end of this phase, each cell of the simulation knows, for each topic, who are the publishers and the subscribers.

Summarizing, using the centralized communication the communication phases are: connection to the server Publish/Subscribe, creation and subscription to topics. In the decentralized/MPI strategy the communication phases are: creation and addition to MPI Groups (that represent D-MASON topics), creation of MPI Communicators.

The main difference between the centralized communication and the MPI one is the synchronization. In the centralized communication, the synchronization is implemented at the framework level using a data structure that indexes, for each step, the updates and acts as barrier, so that each cell remains locked until it receives all the required updates. On the other hand, the MPI strategy takes advantage of the intrinsic synchronization of MPI, because the collective communication primitives are blocking.

Another key difference between centralized communication and decentralized one is that, in the centralized communication, the communication is asynchronous: At the end of a simulation step each cell sends (or publishes) to its topics and creates some message listeners. Each message listener is executed on a separate thread, to receive the updates from the topics to which it is subscribed. In the decentralized communication, on the contrary, the communication is scheduled by the MPI layer, forcing the process in a ordered sequence of communication.

Proposed implementations. Three different implementations, based on the assumptions 2 and 4, have been designed. According to the assumptions 2 and 4, at the end of a simulation step, each cell publishes on its topics; when the last publish routine is invoked it is possible to start the synchronization phase. Since we use the same communication schedule on each cell, we avoid communication deadlocks due to multiple concurrent accesses to the MPI layer.

MPI_Bcast. The first proposed solution uses as collective communication primitive the `MPI_Bcast` function. In this model time is divided in t time-slots, each slot is associated to a D-MASON topic, as in Figure 2.17. The i-th slot is in turn divided in p minislot, each of which represents the *publication turn* of one process publisher

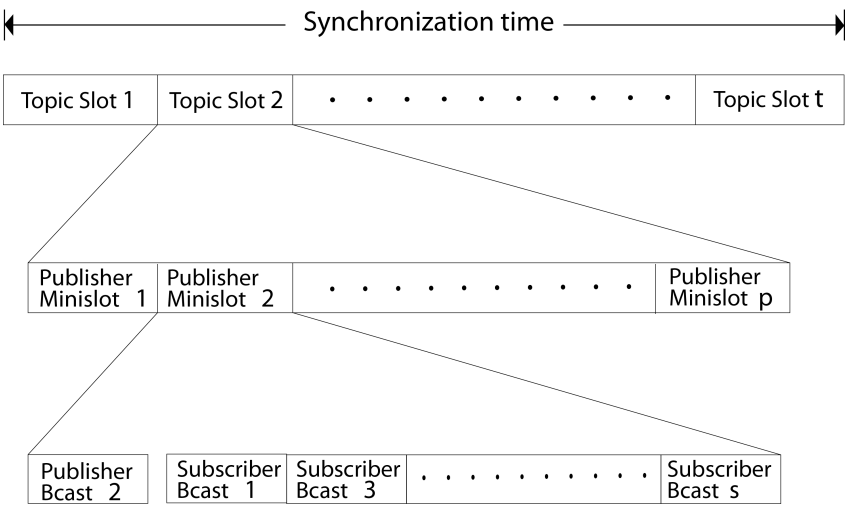

Fig. 2.17.: `MPI_Bcast` approach.

of the topic i. In the j-th minislot, one publisher process invokes `MPI_Bcast` to send updates to subscribers, while the other processes, subscribed to topic i, invoke `MPI_Bcast` to receive updates from the topic i.

MPI_Gather. By observing that, during each simulation step, each cell needs to collect information from its neighborhood, the second implementation uses the `MPI_Gather` function in order to gather all the messages needed in a single time step. This allows to decrease the number of communication phases while increasing the messages size. In this solution, a communication phase is divided into n time-slots, where n is the number of MPI processes (cf. Figure 2.18). During the i-th phase the process i invokes the `MPI_Gather` function in order to receive updates from all its neighborhood.

The Gather strategy is based on two concept:

- *Union Groups.* For each $i = 1, 2, \ldots, n$, the Union Group i contains all MPI processes present in the MPI Group of the topics to which the cell i is subscribed;
- *Communicator Groups.* The Communicator Group i is the Communicator of the Union Group i.

During the phase i:

- The MPI process i executes an `MPI_Gather` in reception.
- All the MPI processes $j \neq i$ in the Union Group of i: create a message for i containing all messages from the topics for which j is the publisher and to which i is subscribed, and invoke an `MPI_Gather` in sending.

Parallel. We performed a deep analysis of *mpiJava* analyzing also several details of its code. In particular, inspecting the source code of the version of *mpiJava* used, available at [Ope13], we observed that in the implementation of `MPI_Bcast` and `MPI_Gather` there is a type check on the data: when it detects that the data to be

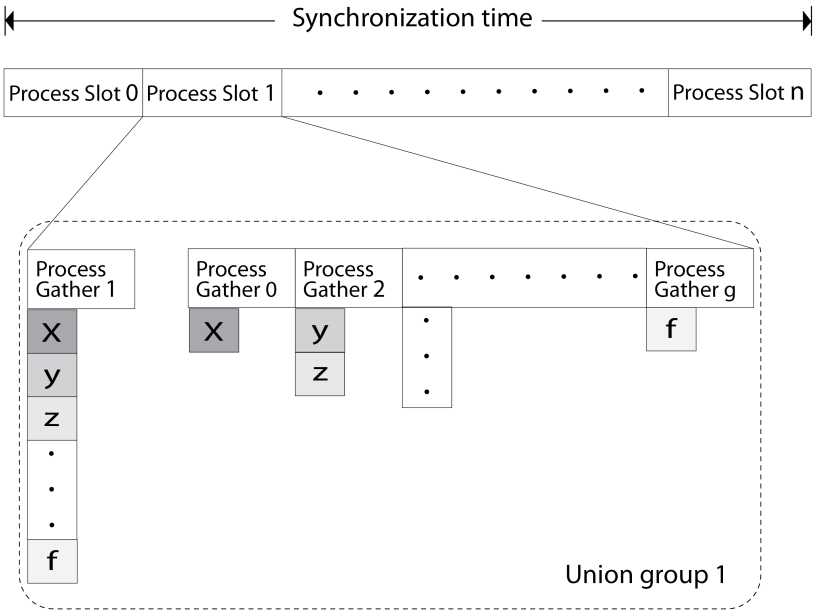

Fig. 2.18.: `MPI_Gather` approach.

sent or received is an *Object* type, it invokes a sequence of `MPI_Send` or `MPI_Receive` to the members of the MPI Group (without exploiting any sort of parallelism). We then found out that in the MPI Strategy, when a cell invokes a routine for collective communication, it executes `MPI_Send` and `MPI_Receive` sequentially to or from all the members of the group.

This explains why the first two strategies use native Java Serialization of the objects, and send and receive arrays of bytes. The problem is that the serialization/deserialization of objects can be computational expensive and for this reasoning an ad-hoc strategy, named *parallel*, was developed without using any collective communication routines in order to increase the degree of parallelism during the communication.

The novel strategy is depicted in Figure 2.19. The problem to be addressed is the following. At the end of each step there are c communications to be executed (communications are represented by a pair $\langle$sender, receiver$\rangle$) during the synchronization phase. Each process can invoke one function (send or receive) at a time, so while it is receiving it can not send anything.

The synchronization phase is partitioned, as shown in Figure 2.20, in r rounds where for each round $i = 1, 2, \ldots, r$ contains c_i disjoint pairs, that can communicate at the same time (that is, each process appears only once as a sender or a receiver).

Clearly the goal is to minimize the size of r. If one maps this problem on a graph where there is a node for each process and a directed edge between two processes q and p if, during the synchronization phase, q has to send a message to p, the problem above become the well-known *Edge coloring* problem, which is known to be NP-hard [MG92]. An edge coloring of a graph is a minimum assignment of colors (rourds in

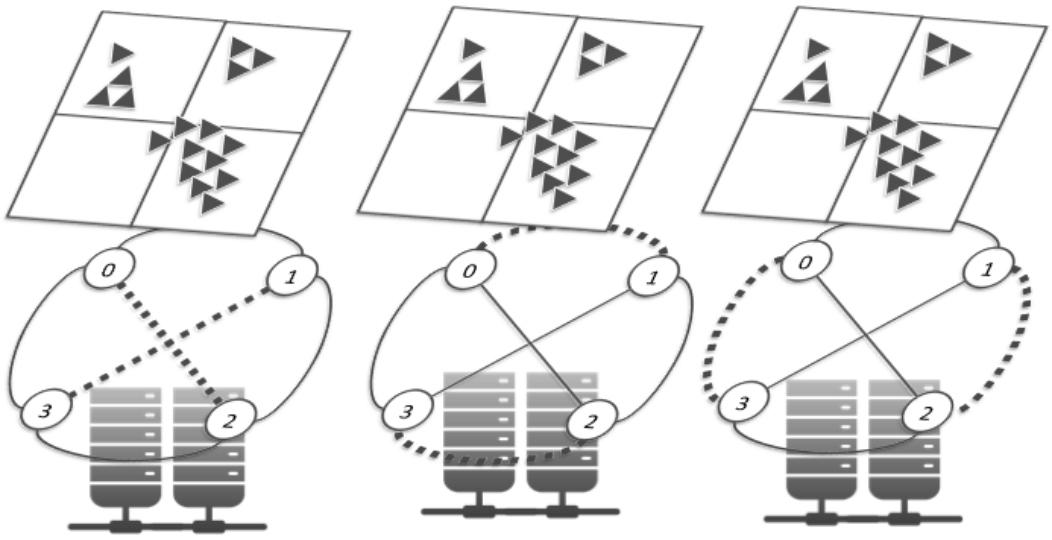

Fig. 2.19.: Possibles simultaneous communications using 4 LP and uniform partitioning mode.

our case) to the edges of the graph so that no adjacent edges have the same color. The number of colors needed to edge color a simple graph is at least Δ where Δ is the maximum degree of the graph. Edge coloring has applications in scheduling problems and in frequency assignment for fiber optic networks. In the literature there are several greedy strategies that construct coloring that use at most $\Delta + 1$ colors. The Parallel strategy uses the following simple randomized strategy. Let E be the edges set and C be the set of colors (rounds). Consider a random permutation of the edges in E. For each uncolored edge $e \in E$, check whether there is a color in C which, once assigned to e, avoids conflicts. If no colors are available, the strategy generates a new color for e, and adds it to C. This greedy algorithm runs, in the worst case, in $\mathcal{O}(|E|^2)$, and uses, $\Delta + 1$ colors on average.

MPI strategies comparison The tree different MPI strategies and the centralized communication were compared running two simulations: *Flockers* and *Ant foraging*.

Flockers is a model stated in 1987 [Rey87] by Craig Reynolds that simulates the behaviour of a flock. In this model "*flockers*" moves according to three simple rules: *separation*, they steer to avoid crowding local flockers, *alignment*, they steer toward the average heading of local flockers, *cohesion*, they steer to move toward the average position (center of mass) of local flockers.

Ant Foraging simulates the foraging process of ant colonies. At the start of the simulation ants are in the nest and no ant knows where the food source is. The food discovery happens through an adaptive and iterative process: ants forager random walk on several paths, lying pheromone traces along the visited path, that is used by future ants to evaluate the quality of the path, and once they find the food

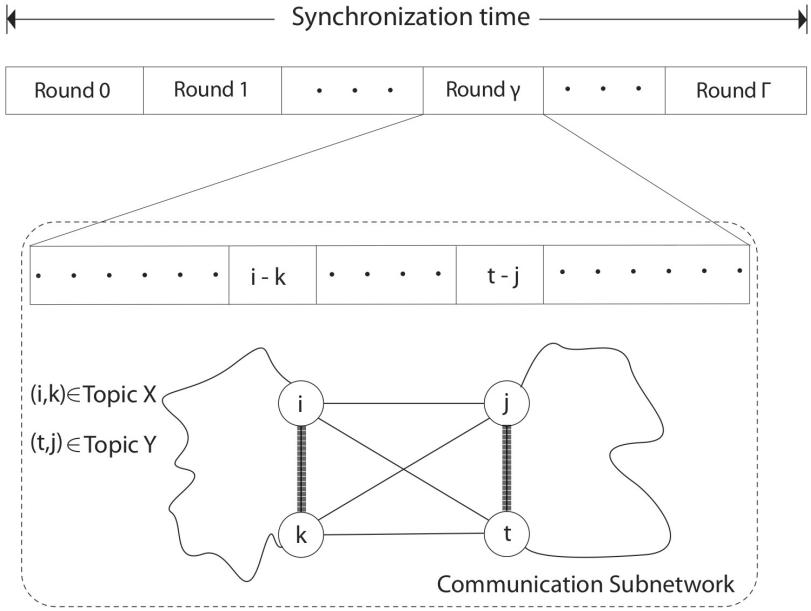

Fig. 2.20.: `Parallel` approach.

source they come back to the nest reinforcing the good path with pheromone traces in a stigmatic fashion. This kind of simulation is characterized by a non-uniform distribution of agents on the field: when the ants find the food source, then they will move only on the shortest path between the nest and the food source.

Simulations were run using 4 LPs, for 100 simulation steps (Flockers) and until the first ant reaches the food (Ants). Several configurations were considered where the parameters:

- The AOI (that influence the amount of communications) ranges from 10 to 50.
- The number of agents (A) range from $500,000$ to $5,000,000$.

The size of the simulation field has been changed in a way that keeps constant the field's density $density \stackrel{\text{def}}{=} \frac{A}{w \times h}$, where w and h denote respectively the width and the height of the field. The density has been chosed according to the original density of the sequential version of the simulations available in MASON. For instance on *Flockers* we start using a 2D field of dimension $7,500 \times 7,500$ with $500,000$ agents up to a 2D field of dimension $23,800 \times 23,800$ with $5,000,000$ agents. The tests have been performed on 4 machines configured ad follow:

- CPU Intel Core i7 $2.53Ghz$;
- Memory $8GB$;
- Operating System Ubuntu 11.04;
- Oracle JDK $1.6.35$;
- ActiveMQ $5.51v$ (for JMS communication);

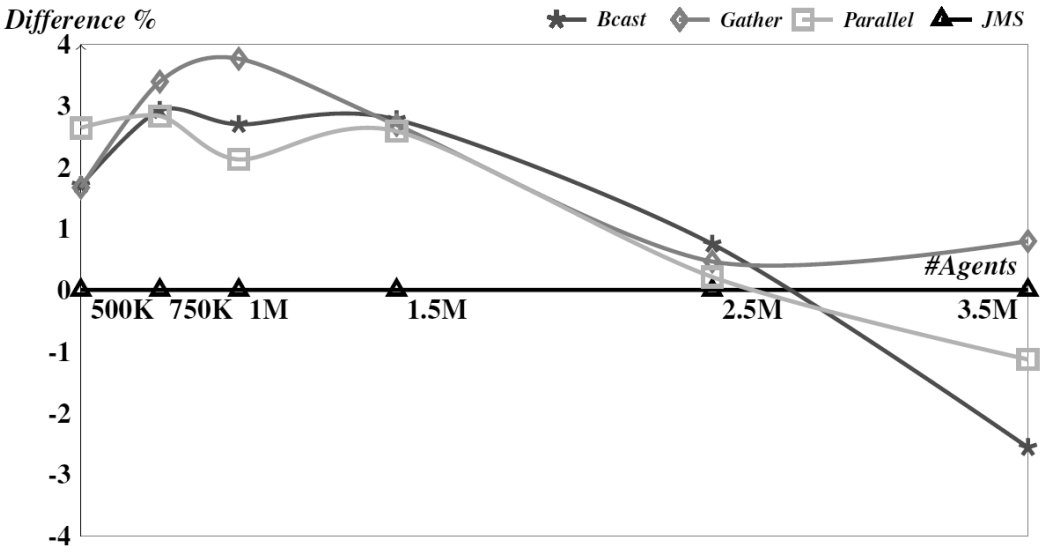

Fig. 2.21.: Performance comparison among JMS Strategy, Bcast, Gather, Parallel. The X axis represents the number of agents while the y axis represents the time difference expressed in percentage compared to the JMS Strategy (lower is better)

- OpenMPI $1.9a1r27886$ nightly snapshot of $01/21/2013$.

The following strategies were tested: JMS (centralized) and MPI (Bcast, Gather, Parallel). Figure 2.21 depicts the comparison between the four strategies, JMS, Bcast, Gather and Parallel, on *Flockers* with $AOI = 10$. The JMS performance are placed on the X axis as a benchmarks for the other strategies. In general, the MPI strategies tend to be better on massive simulations up to a certain value where the MPI Strategies show an additional load due to the Garbage Collection and an heavy disk I/O burst.

It is worth to mention that in our small test scenario, with only 4 workers, the benefit of the Parallel solution with respect to the other solutions do not appear yet. On the other hand in bigger simulations, with a high number of cells (at least 16) it is expected significant enhancements of the performance.

Indeed using n MPI processes with a neighborhood size (degree) Δ, the random algorithm uses at most $\Delta + 1$ communications rounds. In a 2×2 grid in which each cell has 8 neighbours (topic subscriptions), there are 8×4 communication rounds using the Bcast and at most $(8 + 1) \times 2$ communication rounds (we multiply by 2 because each edge of the communication graph is bidirectional) using the Parallel: the gain in percentage is less than the 50% of rounds. In a 4×4 grid in which each cell has 8 neighbors, there are $8 \times 16 = 128$ communication rounds using the Bcast and again at most $(8 + 1) \cdot 2$ communication rounds using the Parallel, so the gain in percentage is greater than the 85% of rounds.

The section 2.6.2 will present a deeper analysis of communication strategies discussing D-MASON scalability with a increasing number of LPs both on an HPC system.

2.4.3 System Management Layer

The System Management Layer was introduced to improve D-MASON usability and engagement. The main purpose of the System Management Layer is to provide a better user experience to scientist that are using D-MASON for their experiments. The management of an application in distributed system is a challenging issue in terms of either efficiency, effectiveness and usability that may determinate the success (or failure) of an application.

As mentioned in Section 2.3.1, D-MASON is based on the Master/Worker paradigm. The D-MASON Master application is used for the discovery of workers (the machines that provides the computational power), the bootstrap and the management of simulations in a simple and efficient way. By exploiting the underlying communication layer it is possible to discover the workers with their hardware and software features. Using this information it is possible to balance the workload between workers, i.e., the number of LPs to be assigned to each worker.

Two different versions of D-MASON System Management were designed. The former version was a standalone Java application, while the last version is a Web based application that better meet the users needs.

The first version of the System Mangement is available in the `experimentals.util.management` package, and is composed by two fundamental classes:

- *JMasterUI* that provides a GUI to discover workers, collect information on their hardware (e.g. processor type, memory, etc ...) and to setup the simulation. Once the user has chosen which simulation to perform, it allows setting up the following parameters: type of field partitioning, number of cells, AOI range, field width, field height and number of agents. Thereafter, it is possible, for each specific worker, to intantiate a certain number of LPs according to its capability. When all the parameters are set, it is possible to start and interact with the entire distributed simulation (play, pause and stop features);
- *WorkerWithGui*, represents the application executed worker-side that interacts with the *JMasterUI* to receive the parts (i.e., cells or subnetworks) to be simulated as well as messages (i.e., commands) for playing, pausing or stopping the simulation.

D-MASON Web System Management

The first version of D-MASON system management had two disadvantages: first, it was not fully decoupled from the simulation part. Hence, adding new features often requires complex interventions with a considerable waste of time. Moreover, the system was designed for local interactions (that is assuming that both the simulation

and the management applications are reachable on several IP ports). Unfortunately, this is not always the case, both NAT and firewall services may result in unreachable ports. For the reasoning above, a fully decoupled system management services easily available via web services was designed and developed.

Design. According to the usability requirement, the Web System Management includes a lightweight web server into its architecture, in this way it is possible to deploy D-MASON more easily on each Java distributed environment. After a deep analysis of the open web servers available for Java, Jetty [Jet13] web server was adopted. In order to develop an efficient, pleasant and engaging interface, the Web System Mangement design was based on Google material design [Goo16c], the guidelines provided by Google for the development of design interfaces. The interface is based on the Polymer library [Goo16b], which has been designed to create components for the modern web, following the material design guideline.

Architecture. The novel web server components has been encapsulated into the D-MASON Master application, and is available on `/resources/systemmanagement` folder and `experimentals.systemmanagement` package, which now comprises two communication components:

- ActiveMQ, for centralized communication (see Section 2.4.2) between D-MASON applications (either master-worker or worker-worker);
- Jetty, for communication between the user and the master application (via web interface).

When the user starts the Master application, both the ActiveMQ and the Jetty server will run on the host. In particular the Jetty server is reachable on a TCP port (default is 8080) and the user can access the management console via browser. Using this approach the user can manage and monitor its simulation, considering that the port 8080 of the Master node is reachable on the Internet.

It is worth to mention that the load of the Jetty Server will not harm the performance of the system. This is true especially when the number of users is small and the user interaction is limited. Indeed, the load of the Jetty server is only due to the activity of discovering and monitoring of LPs. In any case, when this load increases (i.e., a huge number of users continuously interacting with the master and/or a large number of LPs to be monitored) the master node can be configured to use an external ActiveMQ communication server in order to separate the communication and monitoring effort.

A dedicated hand-shaking mechanism enables the negotiation between the Master application and the available workers. When a worker joins the system, it communicates how many slots (LPs) it can afford. As soon as the master realizes that he has

enough LPs to start the simulation, the system enables the user to interact with the desired simulation.

The Web System Management provides four main views, selectable by a control panel (see Figure 2.22):

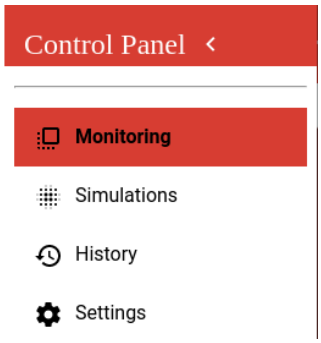

Fig. 2.22.: Master control panel.

1. *Monitoring*, enables the user to monitor the resources available on all connected workers (see Figure 2.23). Using such information, the user is able to choose appropriately the workers to be engaged for future simulations. The system

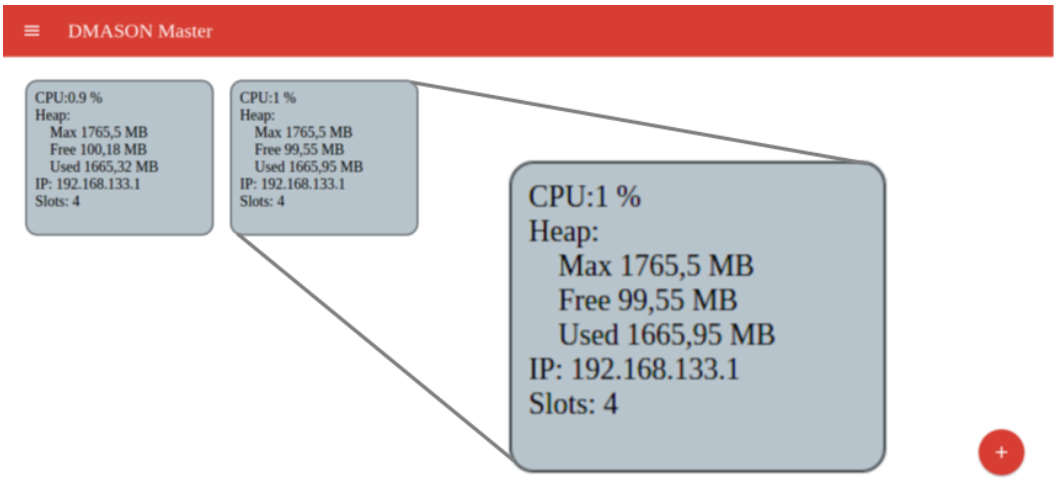

Fig. 2.23.: Workers seen from Master.

management provides a library of preloaded simulation but at the same time, it is possible to upload a novel simulation as a jar file. Once a simulation has been chosen, the user selects the simulation's parameters and submit them to the selected workers (see Figure 2.24).

2. *Simulations*, enables the user to monitor the running simulations (see Figure 2.25). The simulations view shows the list of all the simulations running on the system; for each simulation, using the *Simulation Controller* (see Figure 2.26 (left)), the user can start, pause or stop the execution until the end of the simulation. In order to monitor the evolution of a simulation, a logging

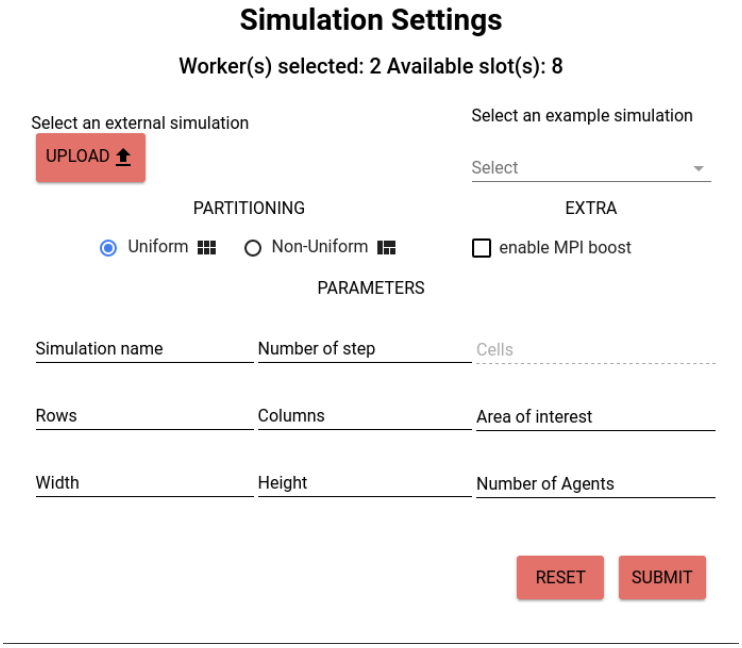

Fig. 2.24.: Simulation Controller.

mechanism has been implemented. All the log files are available at run-time on the Simulation Info panel (see Figure 2.26 (right)).

Fig. 2.25.: Simulations view.

3. *History*, enables the user to visualize the performed simulations (see Figure 2.25). The history page allows also downloading log files of the simulations.
4. Settings, enables the user to change system configurations, for instance the IP and PORT number for the JMS server.

The Appendix C.2 describes other D-MASON feature that help to execute D-MASON on a cluster environment and/or a cloud infrastructure.

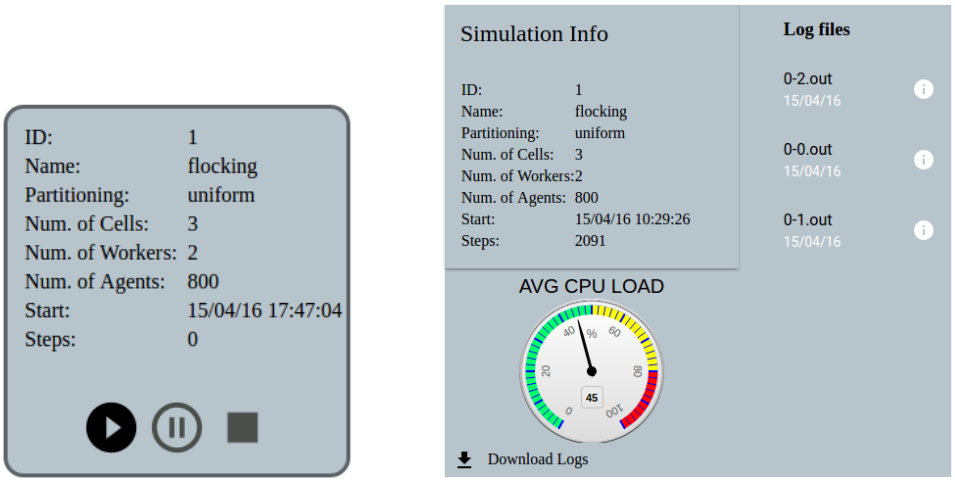

Fig. 2.26.: Simulation Controller (left) and Simulation Info (right)

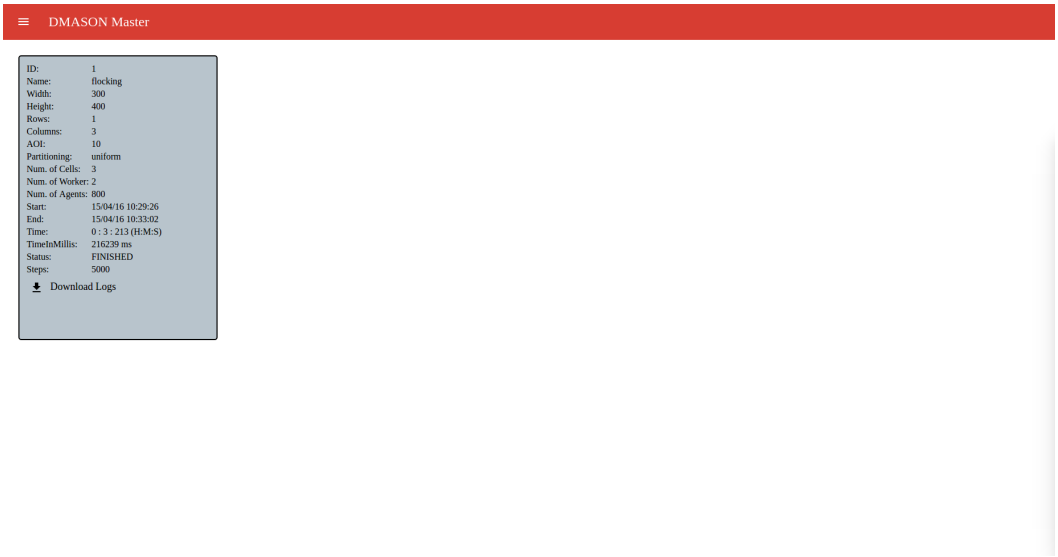

Fig. 2.27.: History view.

2.4.4 Visualization Layer

The Visualization Layer was introduced in D-MASON to face to the usability requirement. The idea of the Visualization Layer is to provide an efficient architecture to visualize massive simulation. The visualization of a massive simulation is an complex problem due to the high numbers of agents to be visualized. A straightforward strategy is to send all agents to a single worker in the system, which visualize each of them. This strategy is clearly not scalable and may degrade the overall simulation performances. Sending all agents may cause both a communication overhead and computation overhead, each LPs must send in a message all its agents. Furthermore, the node that visualize the whole simulations must unpack all messages and iterate over all agents in order to visualize them. This is not feasible for a scalable visualization of massive simulation. Moreover considering the memory limit of a single node, it may not be possible to put all the agents in a single node.

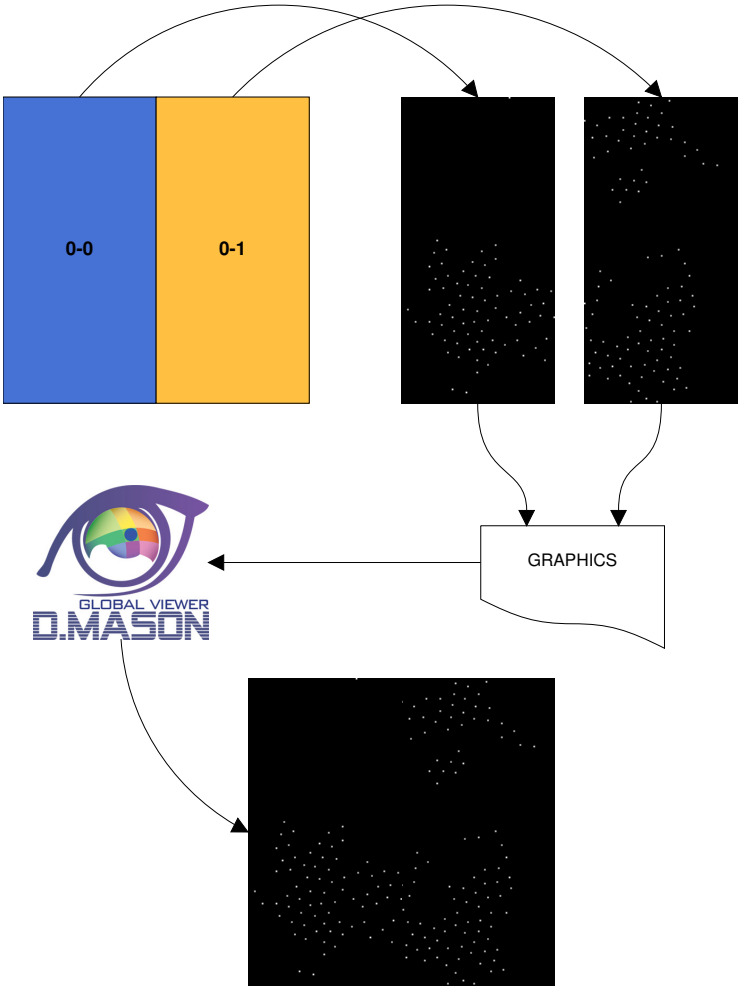

Fig. 2.28.: Visualization Strategy

The initial version of the visualization layer was a standalone Java application, while in the last version of D-MASON, the visualization layer is included in the Web System Management Layer functionality, although the two layer are yet independent. The solutions adopted in the first implementation, and described in the next Section, have been taken similarly in the new version.

2.4.5 Visualization Strategy

The global visualization in D-MASON is inspired by data compressing algorithms. Each LPs send its agents in a compressed way to a system node that is responsable for visualize them. The visualization strategy asssumes that each LP is simulating a rectangular cell. For each simulation step, each LP creates an image (compressing the agents) which represents a snapshot and summarizes some data (for example agents' positioning). Each image is sent to a node (global visualization node), which places the image in the corresponding position of a canvas representing the whole field, without any further elaboration.

With more details, each LP i simulates a cell of a bi-dimensional space having top-left coordinates (x_i, y_i) and size $w_i \times h_i$. Each simulation step is preceded by a synchronization phase, during which each LP builds a bitmap image depicting agents' information on that cells. The bitmap image is packed into a `RemoteSnap` object together with cell ID, top-left coordinates and step number. The `RemoteSnap` object is, then, published to a management topic named `GLOBALS`. The visualization node subscribes to `GLOBALS` so that it will receive `RemoteSnaps` from every LP at every step and puts them in a priority queue ordered according to the step number (see Figure 2.28). The visualization node waits for all the `RemoteSnap` objects of the step being inspected and paints the whole image using the received bitmaps, the top-left coordinates and cell sizes. It is important to store top-left coordinates and cell sizes within `RemoteSnaps` at every step, because the load-balancing mechanism of D-MASON can expand or reduce LP's area of concern. All these operations are implemented at framework level, so the whole process will work transparently provided that agents' information are defined.

The strategy allows to change the level of details of the visualization, according to the size of cells, the level of details needed, the network speed and the computational power available on both, workers and the collector. Strategies like image compression/decompression can be implemented in order to make the system more efficient.

Global Viewer Standalone Application

The visualization strategy was implemented, initially, in a standalone Java application available in the package `experimentals.util.visualization`. The visualization feature is given by two main components: `Global Viewer` e `Zoom App`. The former allows the user to visualize the whole simulation, while the second allows to execute the MASON agents visualization of a selected cell.

Zoom App. The compressed visualization provided by the Global Viewer is not enough to ensure the usability and effectiveness requirements of D-MASON. Assuming that it is not possible to view detailed visualization, of whole simulation field, on a single node, D-MASON provides an additional mechanism to view the detailed visualization of the agents of one cell at time. The idea consists to enable the visualization of a single LP from the Global Viewer, by clicking on the corresponding space in the view, that results in the activation the MASON visualization, which the other hand may be seen as a *zoom* operation.

The zoom application is synchronized with the simulation, due to the massive number of agents to be visualized (but for smaller simulation it is also possible to execute the zoom in asynchronous way). After the activation of the zoom, for a particular LP, the LP packs and sends its agents, at end of each simulation step, on a

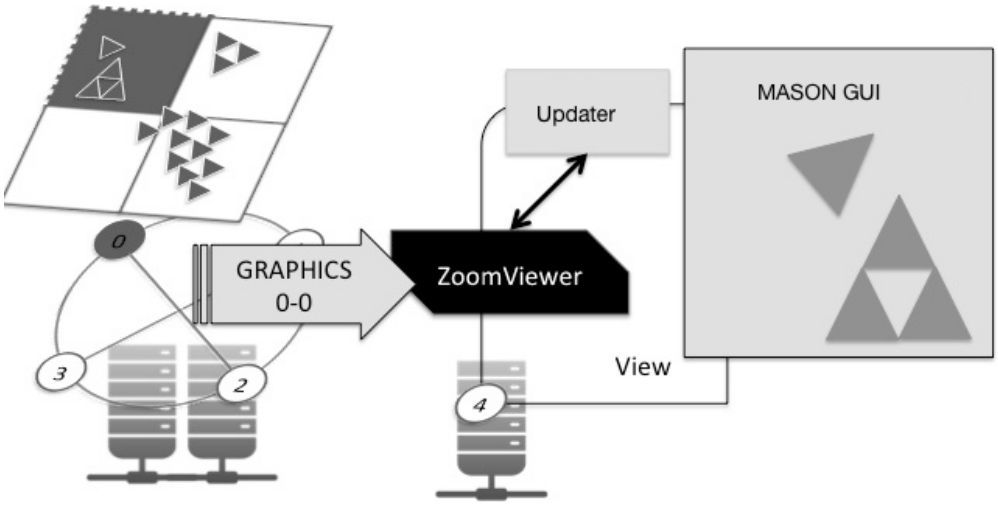

Fig. 2.29.: Zoom App application architecture

topic named `GRAPHICS-ID_LP`. The zoom application is listening on this topic and uses the MASON components to visualize the agents.

The zoom application exploits the MASONvisualization. Basically, a zoom application is an original MASON visualization, without agents' simulation. In this case the fake simulation schedules an agent that extends the `Updater` class. The `Updater` object is responsible for receiving the agents, at the end of each simulation step, and visualizing them using the MASON gui library. Each LP in their `SimState` initializes an object of the class `ZoomViewer`. The `ZoomViewer` object provides all functionalities needed to the zoom application (e.g., the `Updater` object uses this object to receive the agents). The zoom application architecture is shown in Figure 2.29.

Global Viewer Web Application

The last version of the System Management Layer, described in the Section 2.4.3, embed the global visualization. Figure 2.30 shows the global visualization from the Visualization page of the Web System Management.

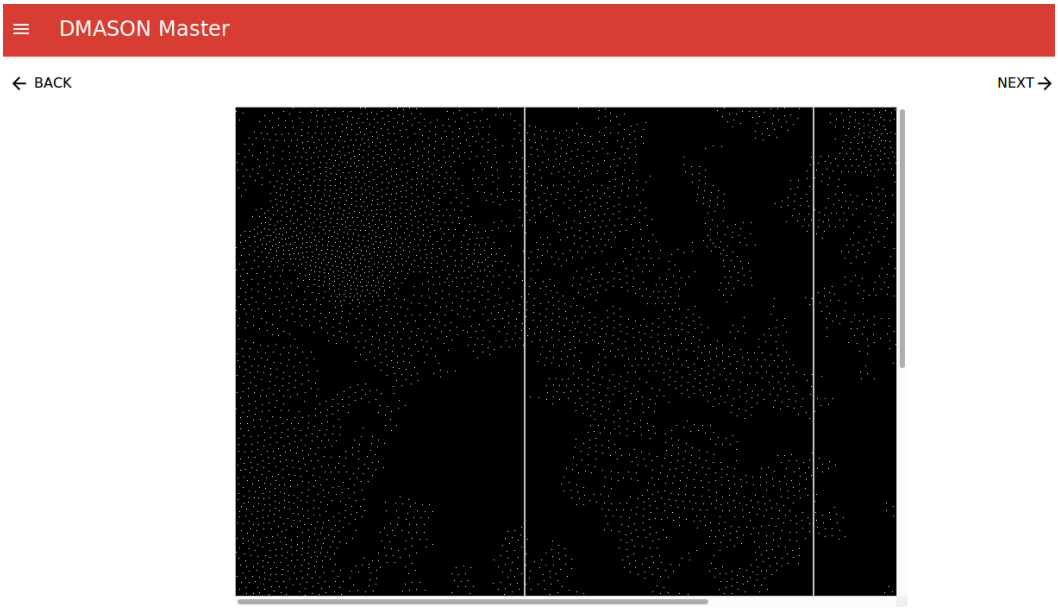

Fig. 2.30.: Global Viewer Web application

2.5 D-MASON on the Cloud

D-MASON was designed for scalable distributed Agent-Based simulations, the current trend in massive computation is to use Cloud Computing infrastructures. D-MASON was tested and experimented on the Amazon Web Services (AWS) cloud infrastructure. In the following sections are described the architecture used, in order to execute D-MASON on the cloud, and a complete analysis and cost evaluation of the experiments. The idea was to realize a SIMulation-as-a-Service (SIMaaS) environment.

2.5.1 The Cloud Infrastructure: Amazon Web Services

Amazon Web Services (AWS) is a scalable and highly reliable cloud infrastructure for deploying applications on demand. The main idea is to let the user building its services with minimal support and administration costs. AWS provides different services on the cloud. Amazon Elastic Compute Cloud (Amazon EC2) provides resizable computing capacity in the cloud. In terms of abstraction layers, the Amazon EC2 is an instance of the Infrastructure as a Service (IaaS) model, where the Amazon infrastructure is seen as a complete virtual environment which allows to execute different instances of virtual machines. Specifically, Amazon allows bundling operating system, application software and configuration settings into an Amazon Machine Image (AMI). Each user can configure and deploy a cluster of machines using a specific AMI instance to run distributed simulations. Advanced users may also create their own AMIs and publish them on the Amazon Marketplace Web Service (Amazon MWS).

In terms of business model, Amazon offers three different purchasing mechanisms: On-Demand Instances, Reserved Instances and Spot Instances. On-Demand Instances have fixed price (per hour) and enable using the resources immediately. With Reserved Instances, it is possible to reserve the utilization of some instances for a predefined period (from 1 to 3 years) with lower payment. Finally, when the timing is not crucial, with Spot Instances, it is also possible to bid for unused resources in order to reduce drastically the costs.

2.5.2 Cluster-computing toolkit: StarCluster

The main issue a user needs to solve in order to use an IaaS service, to run a distributed application, is the configuration and management of each machine. Even using a dedicated AMI, which bundle the basic software components, there are still several parameters that have to be configured separately on each machine. Moreover, the management of the machines is usually time-consuming and requires repetitive tasks that need to be executed for each instance and therefore should be automate to avoid human mistake. To face this issue, a cluster-computing toolkit, StarCluster

[Sta16], released under the LGPL license, has been deployed to configure and manage Amazon EC2 instances. StarCluster enables users to easily setup a cluster computing environment in the cloud, suited for distributed and parallel computing applications and systems.

StarCluster is useful to configure the network of the cluster, create user accounts, enable password-less connections sharing the SSH password between the cluster's nodes, setup NFS shares and the queuing system for the jobs. StarCluster is also customizable via plug-ins, which enable users to configure the cluster with their specific configuration. Plug-ins are written in Python exploiting StarCluster API to interact with the nodes. The API supports the execution of commands, copy of files, and other OS-level operations on the nodes. StarCluster supports also the use of Spot instances allowing the user to run on-demand experiments in easy way and at affordable prices.

2.5.3 Architecture

D-MASON on the cloud has been realized with the purpose to provide a SIMulation-as-a-Service (SIMaaS) environment. The architecture of the system is depicted in Figure 2.31. D-MASON on the cloud is based on a modular approach, which comprises three levels: The Infrastructure is given by Amazon EC2 which provides a wide portfolio of instance types [Ama16] designed to be adopted for different use cases. Instance types vary by CPU performances, memory, storage (size and performance), and networking capacity. The user is free to select an AWS cell according to prices and availability or resources. Starting with a free available Amazon AMI (ami-52a0c53b), which includes a minimal software stack for distributed and parallel computing [Sta16], an AMI, specifically configured for executing D-MASON on the cloud, was realized. The D-MASON AMI, public available on Amazon Infrastructure, provides also Java 8, Maven. On top of that, we developed a StarCluster plug-in, which exploits all the functionality provided

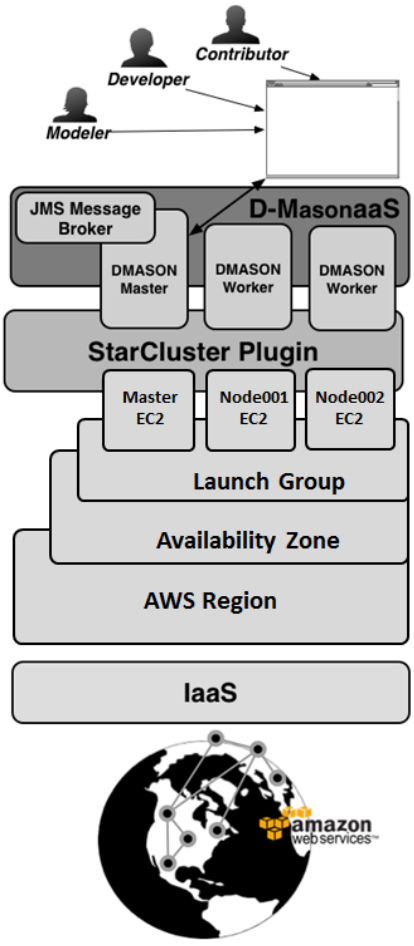

Fig. 2.31.: D-MASON on the Cloud: Architecture.

by StarCluster in order to create automatically a runnable D-MASON environment based on the D-MASON AMI. With more details, the StarCluster plug-in:

- configure the cluster network environment (users account, hostnames setting, SSH key share, NFS setup);
- install and configure the D-MASON environment;
- appoint one of the machines as a Master node.

The master node runs the D-MASON Master application, the JMS message broker (ActiveMQ) and the web system management server (see Section 2.4.3). The other machines run the D-MASON Worker applications, which communicate using the JMS message broker running on the Master node. Each D-MASON Worker application provides a simulation slot for each core available on the machine. The StarCluster D-MASON Plugin is freely available on GitHub D-MASON source code repository [Repce].

The D-MASON tier did not require any particular change: the engine of the system will be executed on the cloud environment and the management is performed thanks to the Web system management interface described 2.4.3.

2.6 D-MASON Performances

Several benchmarks have been performed in order to evaluate the scalability of D-MASON using several partitioning (see Section 2.6.1) and communication (see Section 2.6.2) strategies. An overall evaluation of D-MASON performance is provided in Section 2.6.3. Eventually in Section 2.6.4 we analyze the computational and economic efficiency of running D-MASON on the Amazon Web Services EC2 instances.

2.6.1 Scalability of field partitioning strategies

This analysis aims to evaluate the scalability of the different field partitioning strategies adopted in D-MASON and described in Section 2.3.2. The simulation used is a variant of the *Boids Reynolds* model [Rey87], in which the agents move over a 2-dimensional euclidean field. The agent speed is limited to a fixed range, in order to keep roughly constant the initial distribution of agents on the field.

The simulation experiments compare two field partitioning strategies:

- **Uniform partitioning** (henceforth **Square**), which partition a field in k cells (number of LPs), using a $\sqrt{k} \times \sqrt{k}$ matrix (all the cells have the same dimensions $\left(\left(w/\sqrt{k}\right) \times \left(h/\sqrt{k}\right)\right)$, where w and h are the dimensions of the field).
- **Non-uniform partitioning** (henceforth **Tree**), the *Quad-Tree* based partitioning. The *Tree* partitioning strategy is implemented on top of D-MASON using a variant of the *Quad-tree* data structure. The Java code is available on a public Git repository[1].

The strategies have been tested on two distributions of agents:

- **QU** (Quasi-Uniform distribution), the agents are distributed in 8 groups, 3 groups of 25% of the total agents and 5 of 5% of the total agents.
- **NU** (Non-Uniform distribution), the agents are distributed in giant group on the left-bottom of the field.

This experiments compare the performances of 2 partitioning strategies (Square, Tree) on 2 agents configurations (QU, NU) with 5 degree of granularity $k \in \{4, 9, 16, 25, 36\}$ varying the number of agents in the field in 2 ways, so as to analyze both the weak and the strong scalability. Simulations with granularity k have been executed using k LPs. Overall $2 \times 2 \times 5 \times 2 = 40$ simulation test settings were performed.

The performance of each test is measured in terms of the number of simulation steps performed in a time span of 120 seconds. Since each configuration involves some

[1] D-MASON GitHub repository – Quad Tree Java implementation – https://goo.gl/9JAupK

randomization, we executed each test setting 10 times. The results were compared using means of simulation steps performed (the observed variance was negligible).

The simulation was performed using the centralized communication of D-MASON (see Section 2.4.2), which uses the Java Message Broker *Apache ActiveMQ* $5.5.1$ for the communication between the simulation LPs [Cor+14a].

In the following are described the performance results in terms of weak and strong scalability.

Field partitioning: Weak scalability

In the weak scalability test is explored the ability of the partitioning strategies to scale using a fixed amount of computation for each logical processor (see Section 1.4.1). In order to do that, the number of agents is proportional to k (i.e., $A = 28000 \times k$) while the dimensions of the field are set in order to keep the agents' density constant (i.e., $density = \left(\frac{w \times h}{A}\right) \approx 100$, where w and h denotes the weight and the height of the field) for each weak scalability test.

Twenty configurations varying the value of $k \in \{4, 9, 16, 25, 36\}$, the partitioning strategy (Square and Tree) and the agents distribution (QU and NU) were tested. The Figure 2.32 depicts the weak scalability results as the the number of simulation steps performed in a time span of 120 seconds (Y-axis), for each value of $k \in \{4, 9, 16, 25, 36\}$ (X-axis) and for each *partitioning strategy-agents distribution* (series).

The Tree strategy gives the best performances for both the agents distribution. Furthermore, we notice that the performance trends are affected by the granularity of the decomposition, which impacts on the communications overhead. Using the Square strategy, the amount of communication overhead is proportional to k (8 communication channels for each cell). The tree strategy generates more channels (see table 2.4.2). Hence, with small values of k, the gap is sensible. Increasing k, the improvement tends to decrease as the communication overhead increases, especially using a centralized communication approach.

Field partitioning: Strong scalability

The strong scalability test explores the ability of the partitioning strategies to scale using a fixed amount of computation (see Section 1.4.1). In our case the amount of computation consists of $1M$ agents moving on a 2-dimensional field of size 10000×10000. The Figure 2.33 depicts the strong scalability results as the the number of simulation steps performed in a time span of 120 seconds (Y-axis), for

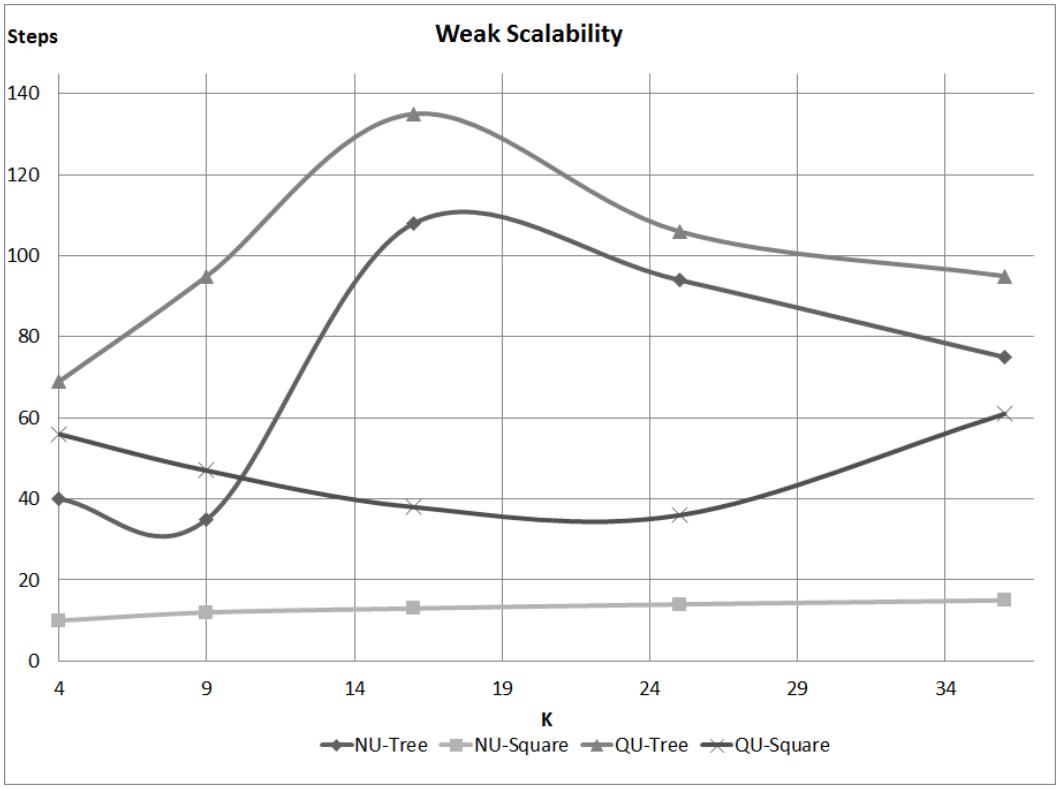

Fig. 2.32.: Field Partitioning Strategies: Weak Scalability

each value of $k \in \{4, 9, 16, 25, 36\}$ (X-axis) and for each *partitioning strategy-agents distribution* (series).

The Tree strategy gives the best performances for both the agents distribution. The improvement ranges from $\times 2$ for the QU agents distribution to $\times 30$ for the NU agents distribution. The figure shows also that the Tree strategy is able to counterbalance the non-uniform agents distribution. Indeed, especially for $k = 16$ and $k = 25$, the performance of the Tree strategy is not affected by agents distribution.

2.6.2 Scalability of the Communication layer

This analysis aims to evaluate the scalability of the different communication strategies adopted in D-MASON and described in Section 2.4.2.

Different experiments have been carried on several configurations obtained varying: number of agents (A), communication scheme (S), number of cells (P), AOI radius and fields dimensions. Such parameters determine a ratio between the communication and computation requirements.

Setting and goals of the Experiments. These tests were performed on a cluster of eight nodes, each equipped as follows:

- CPUs: 2 x Intel(R) Xeon(R) CPU E5-2680 @ 2.70GHz (#core 16, #threads 32)
- RAM: 256 GB

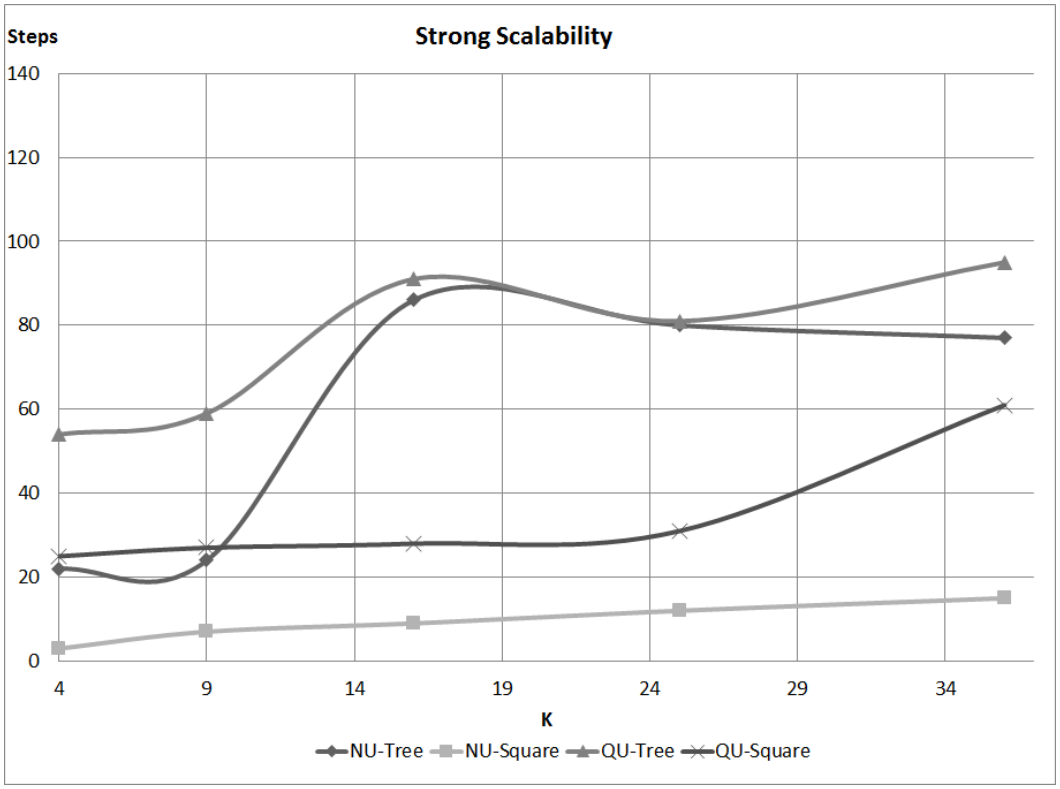

Fig. 2.33.: Field Partitioning Strategies Strong Scalability

- Network: adapters Intel Corporation I350 Gigabit

Considering the high computational power of each node, the tests were able to run several (up to 90) LPs on each node. Simulations have been conducted on a scenario consisting of seven machines for computation and one for managing the simulations and running the ActiveMQ server when needed. As in the previous test, the tests was executed using the simulation *Flockers* (see Section 2.4.2). Among the three decentralized discussed and preliminary evaluated in Section 2.4.2, we decided to analyze only the *Parallel MPI* strategy, which have been shown to be extremely efficient on simulation involving a large number of LPs.

Two experiments were conducted:

1. *Communication scalability*: this test aims to evaluate the scalability of the communication layer in terms of the number of LPs. As the number of LPs increases, the communication requirements become crucial in the efficiency. On the other hand, on very large simulations the ability to run a large number of LPs is essential in order to partition the overall computation without exceeding the physical limits of each LP in the system;
2. *Computation scalability*: this test aims to evaluate the scalability of the communication layer in terms of the number of simulated agents. In this case an increase of the number of agents corresponds to an increase of the computa-

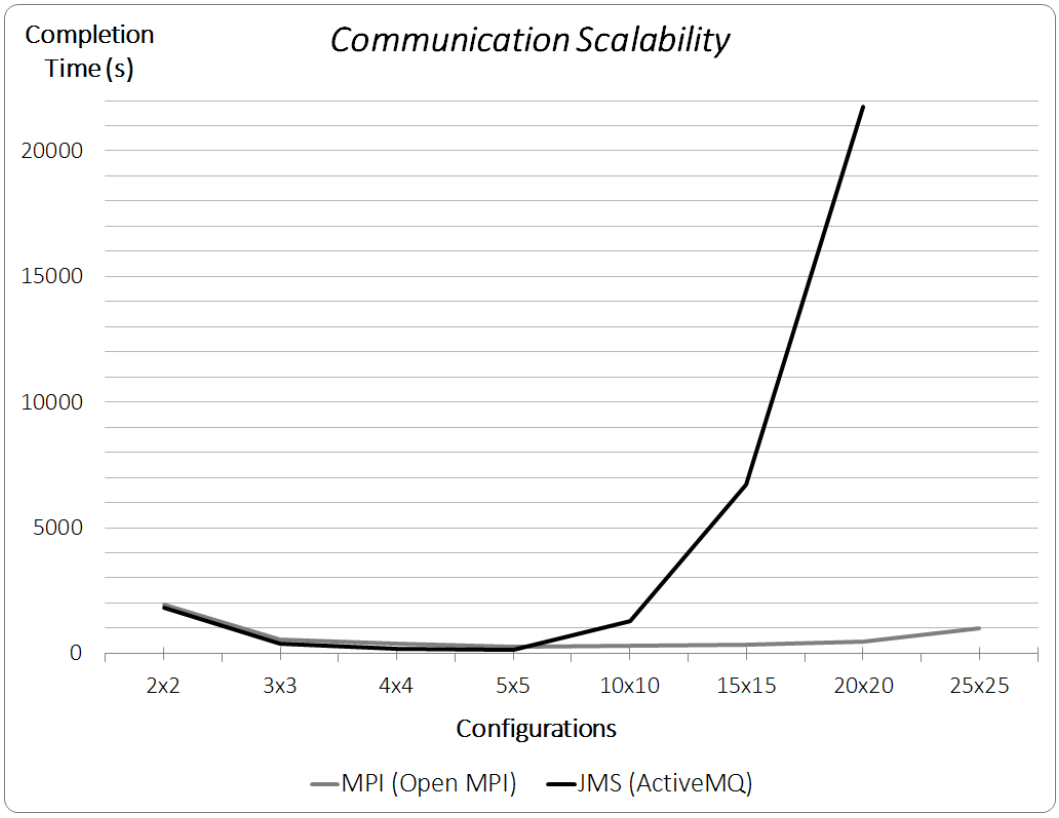

Fig. 2.34.: Communication scalability.

tional power required, and consequently to a reduction of the communication / computation ratio.

Communication scalability test. For this experiment, the field size ($10,000 \times 10,000$), the number of agents (1 million) and the AOI (10), were fixed. This experiment is defined by 16 test settings, each characterized by: the field partitioning configuration (number of rows and columns), which determines also the number of Logical Processes (Number of LPs = $[R]ows \times [C]olums$) and the communication scheme (decentralized or centralized). A couple (P, S) identifies each test setting where

- $P \in \{2 \times 2, 3 \times 3, 4 \times 4, 5 \times 5, 10 \times 10, 15 \times 15, 20 \times 20, 25 \times 25\}$ is the field partitioning configuration.
- $S \in \{ActiveMQ$ (centralized), MPI (decentralized)$\}$ is the communication scheme.

The two communication schemes (decentralized or centralized) were compared by running the simulation *Flockers* for $3,000$ simulation steps. Each simulation has been executed several times in order to check for any fluctuations in the results but the observed variance was negligible.

Figure 2.34 presents the results. The $X-$axis indicates the value of P (left to right the number of LPs is increasing), while the $Y-$axis indicates the overall execution

time in seconds. Notice that there is a point missing because the test setting $(25 \times 25, ActiveMQ)$ crashes after few steps (the centralized ActiveMQ server was not able to manage the communication generated by 625 LPs.)

When the number of LPs is small, the advantage of the decentralized communication does not appear because the message broker is much efficient comparing to the coarse grain synchronization requirement of the decentralized one. By increasing the number of LPs, the efficiency of the centralized message broker gets down dramatically and the simulation performance does exhibit the benefits of using the decentralized communication. This trend is due to the fact that by increasing the LPs number there are much more messages in the system and the effort needed to have a synchronizing mechanism in the decentralized communication approach is hidden by the time taken by the message broker to deliver all the messages.

Computation scalability test. For this experiment, the density of the field (100) and the AOI range (10), were fixed. This experiment is defined by 72 test settings, each characterized by: the field partitioning configuration, the communication scheme and the number of agents. Each test setting is identified by a triple (P, S, A) where

- $P \in \{10 \times 10, 15 \times 15, 20 \times 20\}$ is the field partitioning configuration.
- $S \in \{ActiveMQ$ (centralized), MPI (decentralized)$\}$ is the communication scheme.
- $A \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048\} \times 1,000,000$ (M) is the number of agents.

Six configurations were compared, each one characterized by a field partitioning configuration and a communication scheme, by running the simulation *Flockers* for $3,000$ simulation steps.

Figure 2.35 presents the results. The $X-$axis indicates the number of agents A, while the $Y-$axis indicates the overall execution time in seconds. The test starts with a field size of $10,000 \times 10,000$ and one million of agents, these values are scaled up proportionally in such a way to keep a fixed density along the overall test.

The figure 2.35 shows that for each field configuration the decentralized approach performs better than the centralized one up to a certain number of agents (i.e., $64M$ for 10×10 configuration) that is when the computational requirement are significantly higher than the communication one. However, the figure shows also that, if this is the case, then the system deserves a finer field partitioning. Indeed, by increasing the number of LPs (i.e., moving from 10×10 to 15×15).

Moreover increasing the number of LPs requires more communication, which increases the ratio communication / computation and consequently shifts the "*cross-point*" ($1024M$ for 15×15 configuration). It is worth mentioning that in the last field configuration (20×20), the cross point has not been reached because the centralized

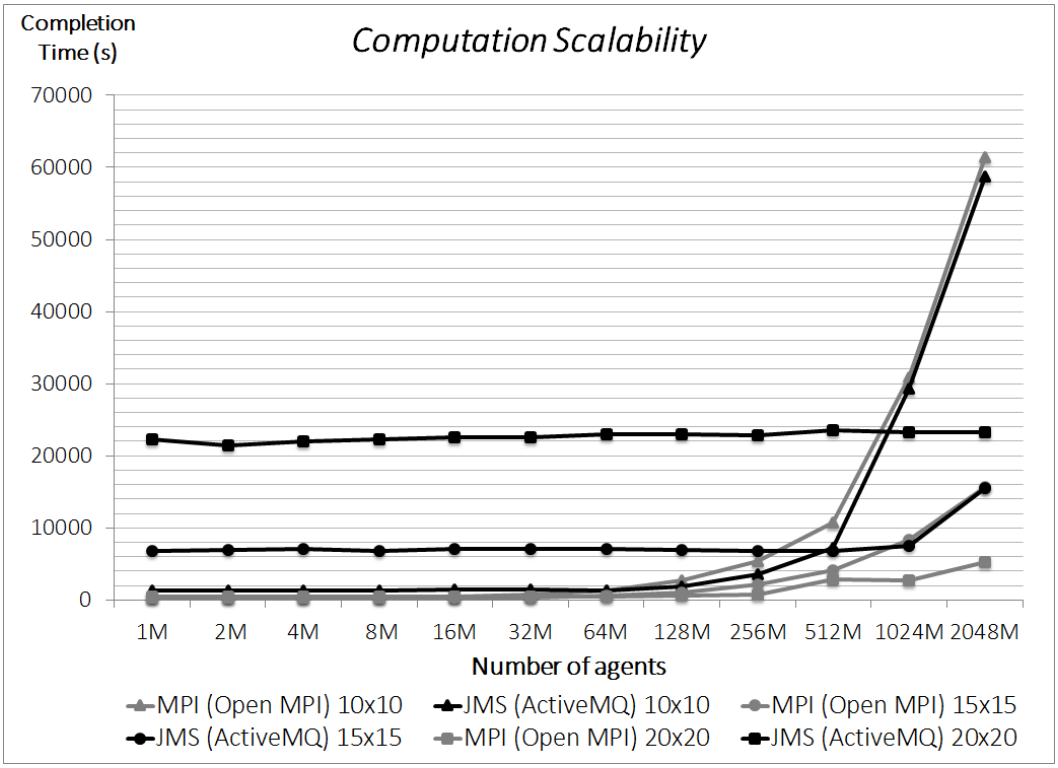

Fig. 2.35.: Computation scalability.

server was not able to manage the communication generated by more than $2048M$ agents.

2.6.3 Beyond the Limits of Sequential Computation

This analysis aims to evaluate the scalability D-MASON to exploit the computational power of D-MASON on a larger infrastructure: a cluster of 15 nodes, each equipped as follows:

- CPUs: 2 x Intel(R) Xeon(R) CPU E5-2430 v2 @ 2.50GHz (#core 12, #threads 24)
- RAM: 32 GB
- Network: adapters Intel Corporation I350 Gigabit
- Software: Ubuntu $14.04.3$ LTS, Oracle Java Virtual Machine 1.8.

The two communication schemes (centralized and decentralized) were compared by running the *Flockers* simulation for 15 minutes. Each simulation has been executed several times in order to check any fluctuations in the results. However, no significant changes were observed in the results.

Weak Scalability

This test aims to evaluate the weak scalability of D-MASON varying the number of LPs. The amount of computation for each LP consists of around 90000 agents. Several tests were performed varying the number of LPs from 4 (360000 Agents) up to 225 (20M Agents).

Figure 2.36 presents the results. The $X-$axis indicates the number of logical processors, while the $Y-$axis indicates the number of steps performed within a time span of 15 minutes. The system scales pretty well, the overall performance degrades gracefully. Moreover, with this configuration the centralized approach seems to perform better than the decentralized one.

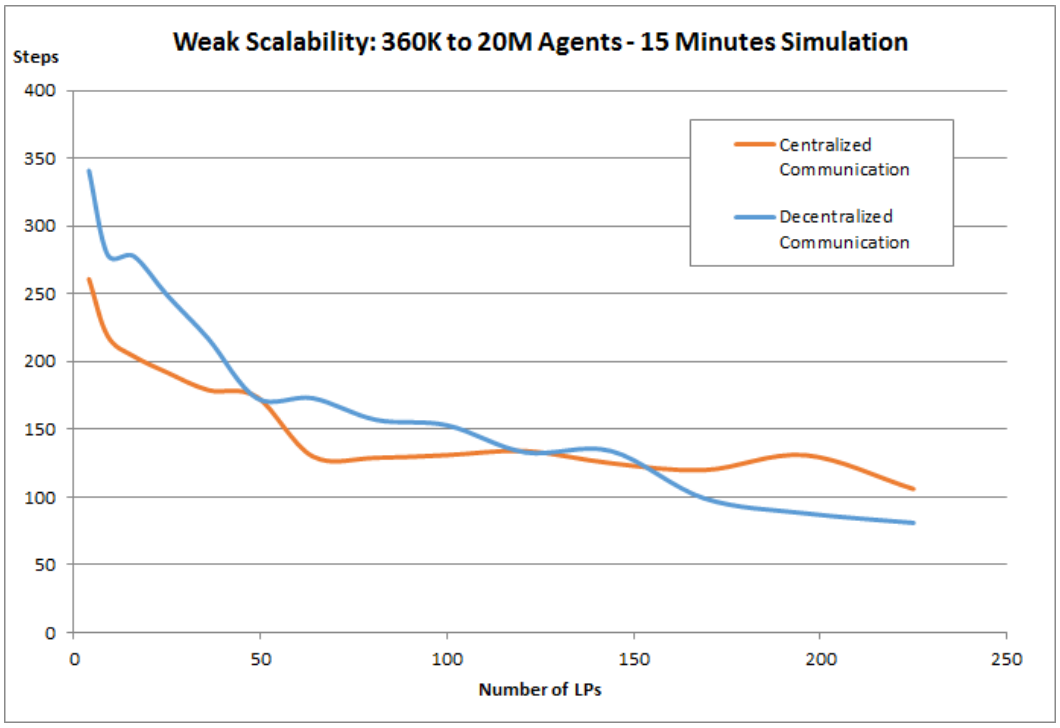

Fig. 2.36.: D-MASON Weak Scalability

Strong Scalability

This test aims to evaluate the strong scalability of D-MASON fixing the overall amount of computation ($20M$ agents) and varying the number of LPs (from 4 to 225).

Figure 2.37 presents the results. The $X-$axis indicates the number of logical processors, while the $Y-$axis indicates the number of steps performed within a time span of 15 minutes. The speedup provided is always better than the 50% of the ideal speedup. Again, with this configuration the centralized approach seems to

perform better than the decentralized one. Hence both these tests suggests that the ActiveMQ server is much more efficient on this more performing machine and a further communication-intensive experiment has been designed and performed.

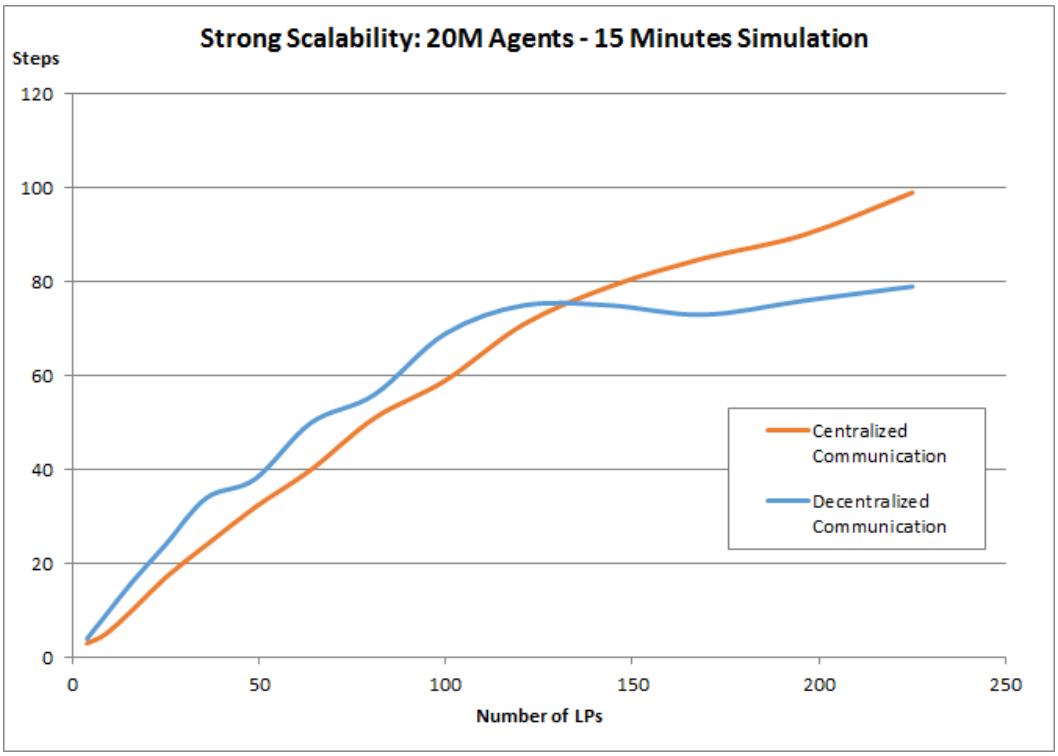

Fig. 2.37.: D-MASON Strong Scalability

A Massive simulation case study

This test aims to evaluate the limits of D-MASON for a particular cluster machine. This test is similar to the weak scalability test described above but here the amount of computation for each LP consists of 450000 agents. Tests were performed varying the number of LPs from 4 (18000000 Agents) up to 225 (100M Agents). The results depicted in figure 2.38 show the limit of centralized approach. The two communication strategy provide similar trends up to 125 LPs. After that the performance of the centralized communication sensibly drops.

2.6.4 Scalability and Cost evaluation on a cloud infrastructure

This benchmarks have been performed on the Amazon AWS cloud infrastructure in order to evaluate the efficiency/cost tradeoff between D-MASON SIMaaS and D-MASON on an HPC infrastructure.

All the tests have been performed on *Flockers*. Boids/Agents have been simulated on a 2D geometric field having size 6400×6400. For each test we executed a

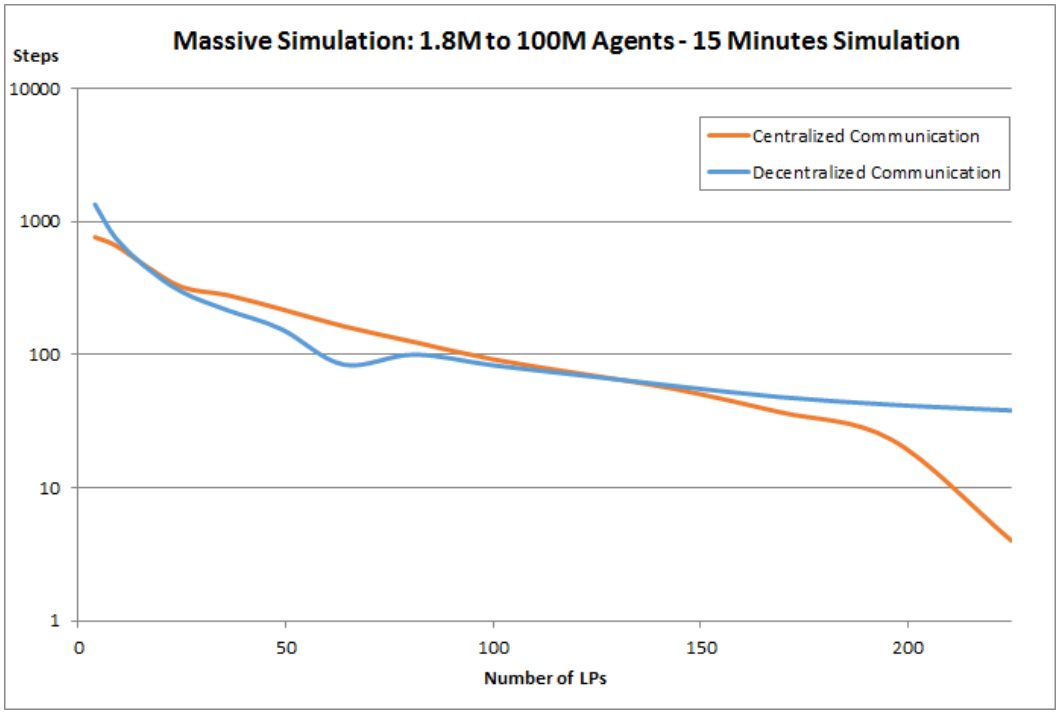

Fig. 2.38.: D-MASON Scalability beyond the limits of sequential computation

reproducible simulation with 1M agents for 15 minutes. At the end of the simulation, the number of simulation steps performed was collected. The web-based system management, described in Section 2.4.3 was used, to start and stop the simulation and to collect the log files.

Five space partitioning strategies (2×2, 2×4, 3×4, 4×4, 4×5) have been considered. All the simulations have been performed with a number of LPs (cores) equal to the number of cells described by the partitioning strategy on four infrastructures (either cloud or HPC). Specifically we tested two cloud instances available on Amazon EC2:

c3.large, processor Intel Xeon E5-2680 v2 (Ivy Bridge) with 2 vCPU, 3.75GB of memory and 2 x 16GB SSD storage (cost $0.105 /h — or 0.019/h for spot at the low price range);

c3.xlarge, processor Intel Xeon E5-2680 v2 (Ivy Bridge) with 4 vCPU, 7.5GB of memory and 2 x 40GB SSD storage. (cost $0.210 /h — or 0.039/h for spot at the low price range).

In order to compare the results against a dedicated on–site environment, we performed the same tests on an HPC cluster. The HPC cluster consists of 16 nodes – each one equipped with 2 $\times$ Intel(R) Xeon(R) CPU E5-2430 with 12 vCPU, 16GB of memory and 1TB HDD storage – interconnected through a Gigabit Ethernet. Each node is running Ubuntu 14.04 operating system with latest updates. The (per node) cost of the considered HPC environment is reported in Table 2.2.

Cost factor	Value	Calculated cost
Hardware purchase	$6500	
Amortization - number of months	36	
Monthly server hardware cost	$200	
Average number of hours in month	730	
Server usage %	50%	
Average number of effective hours in month	365h	
Hardware cost for effective hour		0.49
Power consumption full load	500W	
Power consumption stand by	200W	
Power management unit (PMU)	2.5	
Server usage %	50%	
Average hourly consumption	350×2.5 = 875W	
Electricity price per KWh	$0.13	
Electricity cost for effective hour		$0.11
Rack space	$30 / month	
UPS	$20 / month	
Internet connection	$20 / month	
Collocation effective hour		$0.19
Human hardware maintenance	$200 / server×month	
Managing per effective hour		$0.55
Total effective costs per server hour		$1.34
Number of CPUs	16	
Total effective costs per CPU		$0.08

Tab. 2.2.: Cost calculation for in-house hosting of a single server with 8 Xeon 2-cores processors.

Two different HPC configurations were considered. In the former one, named **HPC1**, all the LPs are executed using a single node, while in the latter, named **HPC***, we executed exactly 2 LPs for each machine. Hence in this last configuration the system uses up to 10 nodes.

Four instances were tested (**c3.large, c3.xlarge, HPC1, HPC***) with 5 partitioning configuration (20 tests overall). Notice that all the tests have been executed on a reproducible deterministic simulation using the same JVM (version 1.8.0_72). Each tests was executed 10 times. The results are compared using means of simulation steps performed (we observed a minimum variance in the cloud instance results, while on the HPC instances the variance was negligible). Results about performance and costs are reported in Table 2.3.

Analyzing the results from Table 2.3: D-MASON on the cloud provide a good degree of scalability with very affordable prices. The **HPC*** instance provides the best performance. This result was expected and we believe that it is mainly due to the quality of the dedicated interconnection network. It should be highlighted, however, that the **HPC*** configuration is considerably more expensive. On the other hand the cloud instances are much cheaper than the **HPC** ones. Moreover, both the cloud

Instance type	# of Instances	# of LPs	Partitioning	Performed steps in 15 min (Avg)	Overall cost	Overall cost EC2 Spot	Cost (x Step) $/1000
c3.large	2	4	2×2	110	$0.210/h	$0.038/h	0.48
c3.large	4	8	2×4	271	$0.420/h	$0.076/h	0.39
c3.large	6	12	3×4	408	$0.630/h	$0.114/h	0.39
c3.large	8	16	4×4	601	$0.840/h	$0.152/h	0.35
c3.large	10	20	4×5	846	$1.05/h	$0.19/h	0.31
c3.xlarge	1	4	2×2	139	$0.210/h	$0.038/h	0.38
c3.xlarge	2	8	2×4	325	$0.420/h	$0.076/h	0.32
c3.xlarge	3	12	3×4	555	$0.630/h	$0.114/h	0.28
c3.xlarge	4	16	4×4	598	$0.840/h	$0.152/h	0.35
c3.xlarge	5	20	4×5	955	$1.05/h	$0.19/h	0.27
HPC1	1	4	2×2	245	$1.34/h	N/A	1.37
HPC1	1	8	2×4	336	$1.34/h	N/A	1
HPC1	1	12	3×4	375	$1.34/h	N/A	0.89
HPC1	1	16	4×4	387	$1.34/h	N/A	0.87
HPC1	1	20	4×5	389	$1.34/h	N/A	0.86
HPC*	2	4	2×2	326	$2.68/h	N/A	2.05
HPC*	4	8	2×4	651	$5.36/h	N/A	2.06
HPC*	6	12	3×4	966	$8.04/h	N/A	2.08
HPC*	8	16	4×4	1293	$10.72/h	N/A	2.07
HPC*	10	20	4×5	1591	$13.4/h	N/A	2.11

Tab. 2.3.: Performance and Costs comparison.

instances scale better than the **HPC1**, which have comparable costs. Finally, in order to measure the trade-off between performances and cost, we computed the cost (per step) of each test setting (see last column of Table 2.3). The results show that the cloud instances are much cheaper than dedicated instances. Figure 2.39 summarizes the results shown in the table 2.3. The number of LPs appear on the X-axis, the number of steps performed in 15 min (avg) appear along Y-axis and the instance configuration are reported as series. The radii of the bubbles are proportional with costs ($\times$ step).

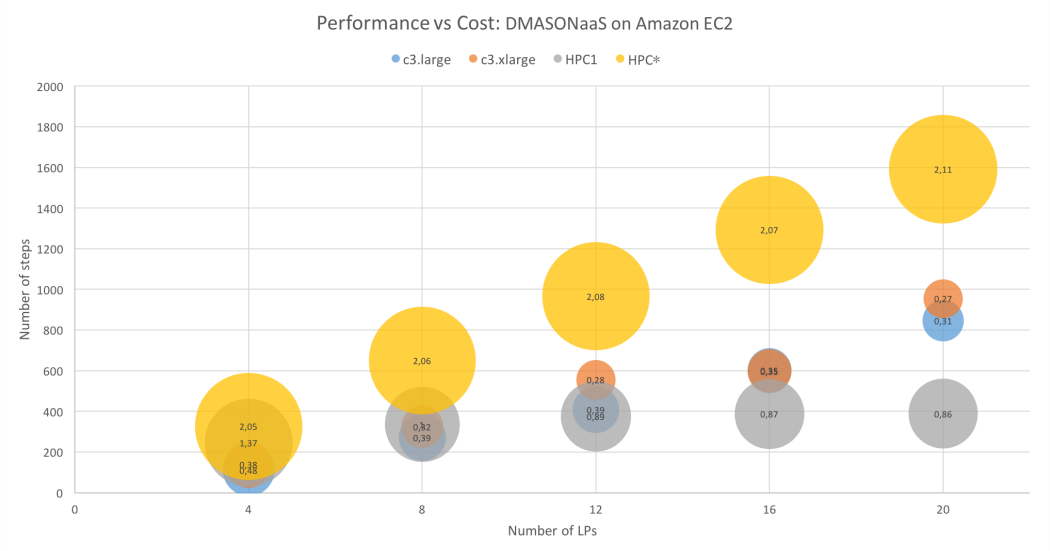

Fig. 2.39.: D-MASON performances on the Cloud and HPC system

3 SWIFT/T Parallel Language and JVM scripting

> *I know how to make 4 horses pull a cart – I don't know how to make 1024 chickens do it.*
>
> **— Enrico Clementi**

3.1 Swift/T: High-Performance Dataflow Computing

The Swift programming language [Wil+09] is a programming scripting for large computational power machines. Swift has been proposed to enabling scientific programmers to easily exploit parallel computing resources in their problems. The first implementation of Swift was called Swift/K because it is based on a system called Karajan grid workflow engine, Karajan and its libraries exploit diverse schedulers (PBS, Condor, etc.) and data transfer technologies in order to orchestrate the computation. Swift/K was designed for coordination of large-scale distributed computations over supercomputing systems. However, Swift/K allows to dispatch at most 500–1000 tasks per second.

To address more demanding parallel applications, a high-performance implementation of Swift, that parallelizes and distributes the executions across several nodes, was designed . Swift/T [Com] is the last implementation of the Swift language.The Swift script is translated into an MPI program, to better exploit the underlying system. The Swift/T syntax and semantics are derived from the Swift, and aims to obtain an high-scalable fine-grained task parallelism. For more details, see the Swift/T website.

Swift/T provides an attractive feature, which makes it of particular interest in the field of Computational Science, that is the ability to invoke code fragments written in other languages including C, C++, Fortran, Python, R, Tcl, Julia, Qt Script, as well as the ability to invoke binary programs. This is achieved by using special function called *leaf function*.

This chapter discusses the design and the development of a new Swift/T feature: the ability to invoke Java Virtual Machine (JVM) based interpreted languages, like Clojure, Groovy, Javascript, Scala etc. This feature is becoming more and more attractive from the Computational Science point of view, due to the high number of open-source scientific programming libraries. Furthermore, many vendors of

supercomputing systems provide in their systems the ability to execute JVM based languages, such as Cray Inc.

The next sections summarize the syntax and basic semantics of the Swift/T language.

3.2 Swift/T Background

This Section summarizes the syntax, the programming model and the basic semantics of the Swift/T language. Then in Section 3.3, the support for interpreted external languages will be extended to (JVM) based interpreted languages. Finally, the support for interpreted external languages is described.

3.2.1 Syntax

The Swift language uses C-like syntax and conventional data types such as `int`, `float`, and `string`. It also has typical control constructs such as `if`, `for`, and `foreach`. Swift code can be encapsulated into functions, which can be called recursively. As shown in 3.1, Swift can perform typical arithmetic and string processing tasks quite naturally. Swift also has a `file` type, that allows dataflow processing on files.

Listing 3.1: Sample Swift syntax

```
1 add(int v1, int v2) {
2   printf("v1+v2=%i", v1+v2);
3 }
4 int x1 = 2;
5 int x2 = toint("2");
6 add(x1, x2);
```

3.2.2 External execution

Swift is primarily designed to call into external user code, such as simulations or analysis routines implemented in various languages. Like many other systems, Swift/T supports calls into the shell. However, this is not efficient at large scale, therefore Swift/T also supports calls into native code libraries directly.

Listing 3.2: Swift used as a Makefile

```
1 app (file o) gcc(file c, string optz) {
2   "gcc" "-c" "-o" o optz c;
3 }
4 app (file x)F(file o) {
5   "gcc" "-o" x o;
6 }
7 file c             = input("f.c");
```

```
8 file o<"f.o">   = gcc(c, "-O3");
9 file x<"f">     = link(o);
```

An example use of Swift for shell tasks is shown in 3.2. This example demonstrates a fragment of a software build mechanism. The user defines two `app` functions, which compile and link a C language file. Swift `app` functions differ from other Swift functions because they operate primarily on variables of type `file`.

Other forms of external execution in Swift/T allow the user to call into native code (C/C++/Fortran) directly by constructing a package with SWIG. Such libraries can be assembled with dynamic or static linking. In the static case, the Swift script and the native code libraries are bundled into a single executable, with minimal system dependencies (for the most efficient loading on a large-scale machine).

3.2.3 Concurrency

Listing 3.3: Code Overview Swift/T Concurrency

```
1  int X = 100, Y = 100;
2  int A[][];
3  int B[];
4  foreach x in [0:X-1] {
5    foreach y in [0:Y-1] {
6      if (check(x, y)) {
7        A[x][y] = g(f(x), f(y));
8      } else {
9        A[x][y] = 0;
10     }
11   }
12   B[x] = sum(A[x]);
13 }
```

The key purpose of Swift/T is to provide an easily to manage environment to perform multiple concurrent executions. This is accomplished in Swift/T through the use of *data flow* instead of *control flow* [EK13]. Adopting the *data flow* model Swift/T does not have an instruction pointer. The execution of each task is triggered by the data availability. This results in an implicit parallel programming model. Listing 3.3 shows an example of Swift/T code, while the Figure 3.1 depicts the corresponding computational workflow.

3.2.4 Features for large-scale computation

Swift/T provides multiple features to support the needs of workflow applications, including support for locations and MPI tasks. The underlying call from Swift/T to external code (via the shell, a script, or native code) is called a *leaf function* as its execution is opaque to Swift/T. These features are accessed by Swift/T *annotations*

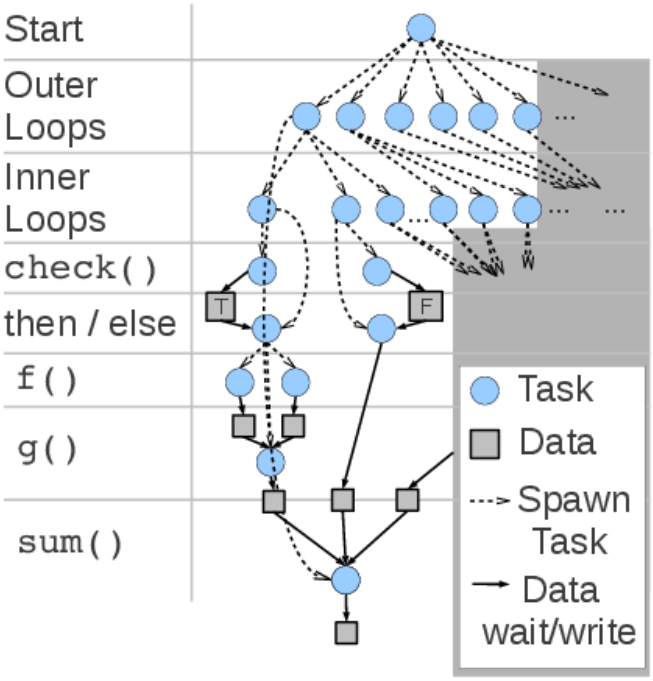

Fig. 3.1.: Diagram Overview Swift/T Concurrency.

that are applied to the leaf function invocation. A generic annotation takes the form

$$type\ result\ =\ @\textbf{\textit{key}}{=}value\ f(params);$$

where the key and the value denote the annotation type.

Task locations. Task locations allow the developer to specify the location of task execution in the system. Locations are optional; by default, Swift/T places the next task in a location determined by the tasks load balancer. Locations can be used to direct computation to part of the system for multiple reasons [Dur+16]. In a data-intensive application, tasks can be sent to the location containing the data to be processed. In a workflow with *resident tasks* [Ozi+15], certain processes retain state from task to task, and can be queried by sending a task to that process.

A `location` object `L` in Swift/T is a data structure containing an MPI rank `R` and optionally other location-aware scheduling constraint information. The MPI rank is the target of the location, it is simply a rank integer in the overall Swift/T run, representing a single process. The code in 3.4 shows the Swift steps for looking up a hostname, constructing a `location` object, and sending a task there.

Listing 3.4: Use of location in Swift.

```
1 R = hostmapOneWorkerRank("node1");
2 L = location(R);
```

```
3 y = @location=L f(x);
```

3.2.5 Parallel tasks

Swift/T offers the ability to construct workflows containing large numbers of MPI tasks that run on variably-sized communicators [Woz+13]. These tasks obtain a communicator of a programmatically-determined size, specified with the `@par` annotation, and (optionally) destroy it on completion. Swift/T obtains the given number of worker processes and constructs these communicators using the `MPI_Comm_create_group()` function in MPI 3. Parallel tasks can be configured to run on contiguous processes, or on any available processes. The `@par` annotation can be combined with other annotations such as `@location` (see Listing 3.5).

Listing 3.5: Swift parallel tasks

```
1 foreach i in [0:9] {
2     @par=i simulate(i);
3 }
```

3.2.6 Support for interpreted languages

Swift/T also provides high-level, easy to use interfaces for Python, R, Julia and Tcl, allowing the developer to pass a string of code into the language interpreter for execution (via its C or C++ interface). These interpreters are optionally linked to the Swift/T runtime when it is built. This allows the user to tightly integrate Swift/T logic with calls to the interpreters, as the interpreter does not have to be launched as a separate program for each call. This is a significant performance benefit on very large scale supercomputers, enabling to make millions of calls to the interpreter per second [Com].

Python Support

Python is a widely used high level programming languages used in several scientific fields. During the years the Python community has developed several libraries as well as code fragments for a wide range of problems.

Listing 3.6: Python example python-f.sh

```
1 export PYTHONPATH=$PWD
2 swift-t -p python-f.swift
```

For this reason many users ask for tools that enable to access Python from the top level of the scientific workflow; and optionally call down from the interpreted level

into native code, to gain high-performance operations for numerical methods or event-based simulation. A popular example of this model is `Numpy`, which provides a high-level interface for interaction, with high-performance, vendor-optimized `BLAS`, `LAPACK`, and/or `ATLAS` numerical libraries underneath.

A basic example of Python usage from Swift/T is shown in Listings 3.7, 3.8 and 3.6. In this three codes fragment, a short module is defined in `F.py` (Listing 3.7) which provides an addition function named `f()`.

Listing 3.7: Python example F.py

```
1 def f(x, y):
2     return str(x+y)
```

A call to this function from Swift/T is shown in `python-f.swift` (Listing 3.8) lines 3-5. The string containing the python code is populated with the Pythonic `%` operator, which fills in values for `x` and `y` at the conversion specifiers `%i`. The Python function `F.f()` receives these values, adds them, and returns the result as a string. Swift/T receives the result in `z` and reports it with the Swift/T builtin `trace()` function.

Listing 3.8: Python example python-f.swift

```
1 import python;
2 x = 2; y = 3;
3 z = python("import F",
4   "F.f(%i,%i)"
5   % (x,y));
6 trace(z);
```

Thus, data can easily be passed to and from Python with Pythonic conventions; only a string formatting is required. To execute, the user simply sets `PYTHONPATH` (Listing 3.6) so that the Python interpreter can find module `F`, and runs `swift-t`.

R Support

The R support in Swift/T is similar to the Python support. An example use case is shown in Listing 3.9. This script is devoted to run a collection of simulations in parallel, then send result values to R for statistical processing. The first section (lines 1-4) simply imports requisite Swift packages. The second section (lines 6-10) defines the external simulation program, which is implemented as a call to the `bash` shell random number generator, seeded with the simulation number `i`. The output goes to temporary file `o`. The third section (lines 11-15) calls the simulation a number of times, reading the output number from disk and storing it in the array `results`. The fourth section (lines 16-19) computes the mean of `results` via R. It joins the results

into an R vector, constructed with the R function `c()`, then it uses the R function `mean()`, and returns the mean as a string `mean` that is printed by Swift.

Listing 3.9: R example stats.swift

```
1  import io;
2  import string;
3  import files;
4  import R;
5
6  app (file o) simulation(int i) {
7    "bash" "-c"
8      ("RANDOM=%i; echo $RANDOM" % i)
9      @stdout=o;
10 }
11 string results[];
12 foreach i in [0:9] {
13   f = simulation(i);
14   results[i] = read(f);
15 }
16 A = join(results, ",");
17 code = "m = mean(c(%s)))" % A;
18 mean = R(code, "toString(m)");
19 printf(mean);
```

3.3 Support for JVM interpreted languages

As described in Section 3.2.6, Swift/T provides a simple mechanism to invoke Python and R code. Similar strategy can be uses to invoke Julia, Tcl and C code. This Section describes a novel contribution to Swift/T that allows to invoke JVM interpreted languages. Whit more details, Section 3.3.1 will present the architecture and the implementation of a `C-JVM` engine that enable to invoke different *Java Virtual Machine* (JVM) interpreted languages from C code. Section 3.3.2 describes how to use `C-JVM` engine within Swift/T for supporting JVM interpreted languages.

3.3.1 C-JVM interpreted languages engine

The C-JVM engine is published on [Ca] public repository. It aims to provide an easily C API to invoke JVM interpreted languages. An interpreted language is a programming language for which most of its implementations execute instructions directly, without previously compiling a program into machine-language instructions. Many interpreted languages are first compiled to Java bytecode and after executed directly on a JVM. The most famous JVM interpreted languages are:

- *Clojure* ([Lan]), is a dialect of the Lisp programming language and particularly devoted to functional programming;

- *Groovy* ([Jp]), is an object-oriented programming language for the Java platform, but is possible to use it as a scripting language, similar to Python;
- *Scala* ([lan]), is a general-purpose programming language. Scala has full support for functional programming and a strong static type system. Scala supports the scripting;
- *JavaScript* ([Jav]), is a high-level, dynamic, untyped, and interpreted programming language mostly used in Web production.

The C-JVM engine supports up to now all the above languages, but the support for others languages and frameworks, like Apache Spark Library are currently ongoing works. The architecture of the C-JVM engine follows.

C-JVM engine: Architecture

The C-JVM engine is composed by two functional blocks: a C library (named `C-JVM-c`) that provides the ability to invoke the considered interpreted languages, and a Java library (named `C-JVM-j`) that concretely implements the evaluation of the code.

The C-JVM-c uses the Java *Java Native Interface* (JNI) API to initialize a new JVM and invoke Java codes to evaluate a string, containing the code of a interpreted language, by using the C-JVM-j. JNI is a programming framework that enables Java code to call and to be called by native applications such as C, C++ and assembly. The Listing 3.10 depicts the C code of the C-JVM-c interface. The reader will observe that the functions exported are those for evaluating a string of code written in each of the supported interpreted languages.

Listing 3.10: C-JVM-c interface

```
#ifndef SWIFT_JVM_H_   /* Include guard */

#define SWIFT_JVM_H_

//extern char path_java_code[]="classes";
/* Evaluate Clojure Code and return a char array of the stdio*/
char * clojure(char *code);
/* Evaluate Groovy Code and return a char array of the stdio*/
char * groovy(char *code);
/* Evaluate Scala Code and return a char array of the stdio*/
char * scala(char *code);
/* Evaluate JavaScricpt Code and return a char array of the stdio*/
char * javascript(char *code);

#endif //SWIFT_JVM_H_
```

The code in Listing 3.11 shows the implementation of C-JVM-c. The C code uses JNI to call static Java methods, provided by C-JVM-j. These methods enable to evaluate strings of code. Two type of evaluation are supported: one will be used when the code is supposed to provide an output (as a string) and one will be used when no output is expected. For instance, lines $40-46$ evaluates a string of Groovy code that is supposed to provide an output. The first step shown in line 42 is the JVM initialization. Then, two C-JVM-j methods are invoked, the first method set the engine of the JVM interpreter as Groovy, while the second invokes the method `eval` that returns the output string, given by the evaluation of the Groovy code.

Listing 3.11: C-JVM Engine

```

#include "swift-jvm.h"
/* -- Includes Definitions */
/*hide*/

/* -- Macro Definitions */
/*hide*/

JNIEnv * env;
JavaVM * jvm;

static inline void pdebug(const char* fmt, const char* s)
{
  printf(fmt, s);
  fflush(stdout);
}

char * createClassPathString(char *jars_dir)
{
/*hide*/
```

```
21 }
22 void call_java_static_method(char *java_class_name,char
       *method_name,char *arg)
23 {
24  /*hide*/
25 }
26
27 char * call_java_static_char_method(char *java_class_name,char
       *method_name,char *sengine,char *scode)
28 {
29  /*hide*/
30 }
31
32 static int init_jvm() {
33 /*hide*/
34 }
35 void destroy_jvm()
36 {
37   /*hide*/
38 }
39 /* Evaluate Groovy Code and returns a char array of the stdio*/
40 char * groovy(char *code)
41 {
42     if(jvm == NULL) init_jvm();
43   call_java_static_method(
         "it/isislab/swift/interfaces/SwiftJVMScriptingEngine",
         "setEngine", "groovy");
44   char* tor = call_java_static_char_method
         ("it/isislab/swift/interfaces/SwiftJVMScriptingEngine", "eval",
         "groovy",code);
45   return tor;
46 }
47 /* Evaluate Clojure Code and returns a char array of the stdio*/
48 char * clojure(char *code)
49 {
50   if(jvm == NULL) init_jvm();
51   char * tor=call_java_static_char_method
         ("it/isislab/swift/interfaces/SwiftJVMScriptingEngine", "eval",
         "clojure",code);
52   return tor;
53 }
54 /* Evaluate Scala Code and returns a char array of the stdio*/
55 char * scala(char *code)
56 {
57   if(jvm == NULL) init_jvm();
58   char * tor=call_java_static_char_method
         ("it/isislab/swift/interfaces/SwiftJVMScriptingEngine", "eval",
         "scala",code);
59   return tor;
60 }
61 /* Evaluate JavaScricpt Code and returns a char array of the stdio*/
```

```
62 char * javascript(char *code)
63 {
64   if(jvm == NULL) init_jvm();
65   char * tor=call_java_static_char_method
       ("it/isislab/swift/interfaces/SwiftJVMScriptingEngine", "eval",
       "javascript",code);
66   return tor;
67 }
```

The listing 3.12 depicts the usage of the C-JVM-c library from C code, for evaluating a string of Groovy code.

Listing 3.12: C-JVM Engine Test (Groovy)

```
1 #include "swift-jvm.h"
2 #include <assert.h>
3 #include <string.h>
4 #include <stdio.h>
5 int main(void)
6 {
7     char *groovyoutput=groovy("import java.security.MessageDigest\n
          def hash(text){ MessageDigest.getInstance(\"SHA-512\").
          digest(text.getBytes(\"UTF-8\")).encodeBase64()
          .toString()}\n println hash(UUID.randomUUID().toString())");
8     printf("Groovy: ok, Output: %s \n", groovyoutput);
9
10 }
```

The C-JVM-j is a Java library developed using Maven. C-JVM-j is composed by six modules: *swift-jvm-build*, *swift-clojure*, *swift-groovy*, *swift-scala* and *swift-javascript*, *swift-interfaces*. The swift-jvm-build module defines the build of the library. The swift-*(language-name) modules define the Java classes for evaluating the code of a particular interpreted languages. Finally, the swift-interfaces module is the hearth of C-JVM-j as it provides the external library functionalities. In the following a detailed description of the module swift-interfaces will be provided.

Swift-interfaces module. The swift-interfaces module is composed by three Java classes:

- `SwiftJVMScriptingInterface` (see Listing 3.13), a Java interface to define new language engine class. It is composed by two methods: `init` and `eval`. The `init` method allows to initialize the language interpreter and the `eval` method that evaluates the given string of code and returns an `Object`;

  Listing 3.13: C-JVM-j SwiftJVMScriptingInterface.java

  ```
  1 package it.isislab.swift.interfaces;
  2 import javax.script.ScriptException;
  ```

```
3 public interface  SwiftJVMScriptingInterface {
4     public void init() throws ScriptException;
5     public Object eval(String code) throws ScriptException;
6 }
```

- `SwiftJVMScriptingEngine` (see Listing 3.14), a Java Plain Old Java Object (POJO) class that allows to operate in two different way. The first way is to set up the language (using the `setEngine` method) and call each time the method `String eval(String code)`, directly without setting the language. The second way uses the method `String eval(String engine_name_given, String code)` that initializes the corresponding interpreter and evaluates the code given as parameter;

Listing 3.14: C-JVM-j SwiftJVMScriptingEngine.java

```
1  package it.isislab.swift.interfaces;
2  import java.io.StringWriter;
3  import javax.script.ScriptContext;
4  import javax.script.ScriptEngine;
5  import javax.script.ScriptEngineManager;
6  import javax.script.ScriptException;
7  import it.isislab.swift.scala.ScalaScriptEngine;
8  import it.isislab.swiftlang.swfit_clojure.ClojureScriptEngine;
9  public class SwiftJVMScriptingEngine {
10    public static ScriptEngine engine;
11    public static String engine_name;
12    public static void setEngine(String engine_name_given)
13    {
14       engine_name=engine_name_given;
15       try {
16          switch (engine_name) {
17          case SwiftJVMScriptingEngineNames.CLOJURE:
18             engine = new ClojureScriptEngine();
19             break;
20          case SwiftJVMScriptingEngineNames.GROOVY:
21             engine = new ScriptEngineManager() .getEngineByName
                   (SwiftJVMScriptingEngineNames.GROOVY);
22
23             break;
24          case SwiftJVMScriptingEngineNames.SCALA:
25             engine = new ScalaScriptEngine();
26             break;
27          case SwiftJVMScriptingEngineNames.JAVASCRIPT:
28             engine = new ScriptEngineManager(). getEngineByName(
                   SwiftJVMScriptingEngineNames.JAVASCRIPT);
29             break;
30          default:
31             break;
32          }
33       } catch (ScriptException e) {
```

```
34            e.printStackTrace();
35        }
36    }
37    public static String eval(String code)
38    {
39        Object output=null;
40        try {
41            switch (engine_name) {
42            case SwiftJVMScriptingEngineNames.CLOJURE:
43                output=(engine.eval(code,engine.
                      getContext())).toString();
44                return output!=null?output.toString():"";
45            case SwiftJVMScriptingEngineNames.GROOVY:
46            case SwiftJVMScriptingEngineNames.JAVASCRIPT:
47                StringWriter writer = new StringWriter();
48                ScriptContext context = engine.getContext();
49                context.setWriter(writer);
50                engine.eval(code);
51                output = writer.toString();
52                return output!=null?(String)output:"";
53            case SwiftJVMScriptingEngineNames.SCALA:
54                output=engine.eval(code);
55                return output!=null?output.toString():"";
56
57            default:
58                return null;
59            }
60        } catch (ScriptException e) {
61            e.printStackTrace();
62            return null;
63        }
64
65    }
66    public static String eval(String engine_name_given,String
          code)
67    {
68        Object output=null;
69        StringWriter writer;
70        ScriptContext context;
71        engine_name=engine_name_given;
72        try {
73            switch (engine_name) {
74            case SwiftJVMScriptingEngineNames.CLOJURE:
75                engine = new ClojureScriptEngine();
76                output=(engine.eval(code,engine.
                      getContext())).toString();
77                return output!=null?output.toString():"";
78            case SwiftJVMScriptingEngineNames.GROOVY:
79                engine = new ScriptEngineManager().
                      getEngineByName(SwiftJVMScriptingEngineNames.GROOVY);
80                writer = new StringWriter();
```

```
81                context = engine.getContext();
82                context.setWriter(writer);
83                engine.eval(code);
84                output = writer.toString();
85               return output!=null?(String)output:"";
86            case SwiftJVMScriptingEngineNames.SCALA:
87               engine = new ScalaScriptEngine();
88               output=engine.eval(code);
89               return output!=null?output.toString():"";
90            case SwiftJVMScriptingEngineNames.JAVASCRIPT:
91               engine = new ScriptEngineManager() .getEngineByName(
                     SwiftJVMScriptingEngineNames. JAVASCRIPT);
92                writer = new StringWriter();
93                context = engine.getContext();
94                context.setWriter(writer);
95                engine.eval(code);
96                output = writer.toString();
97               return output!=null?(String)output:"";
98            default:
99               return null;
100           }
101       } catch (ScriptException e) {
102          e.printStackTrace();
103          return null;
104       }
105   }
106 }
```

- `SwiftJVMScriptingEngineNames` (see Listing 3.15), a Java class that exports the available languages names.

Listing 3.15: C-JVM-j SwiftJVMScriptingEngineNames.java

```
1 package it.isislab.swift.interfaces;
2 public class SwiftJVMScriptingEngineNames {
3    static final String CLOJURE = "clojure";
4    static final String GROOVY = "groovy";
5    static final String SCALA = "scala";
6    static final String JAVASCRIPT = "javascript";
7 }
```

Usage

The C-JVM requires different tools for building the library: Java Development Kit (JDK) version major/equal to 1.7, Maven 3, gcc 4.2, autoconf (GNU Autoconf) 2.69 and automake (GNU automake) 1.14. The commands in the Listing 3.16 allows to build the C-JVM (assuming that the script is executed from the root of the project).

Listing 3.16: Build C-JVM

```
1 ./bootstrap
2 ./configure
3 make
4 #change this with additional jar folder libraries
5 export SWIFT_JVM_USER_LIB= swift-jvm/swift-jvm-build/
      target/swift-jvm-build-0.0.1-bin/ swift-jvm/classes/
6 #change this with JVM home
7 export LD_LIBRARY_PATH= /usr/lib/jvm/ java-8-oracle/jre/lib/
      amd64/server
```

3.3.2 C-JVM and Swift/T

This section describes the software integration of C-JVM in Swift/T language. The Swift/T code is available on the public repository [Com] which provides also a guide [Gui16] on how to contribute to the project (allowing the developers to add new language features). Starting from the guidelines of Swift/T project, two principal entities are developed. A C back-end that exports the functions implementation in Tcl and a Swift function for each interpreted languages.

Listing 3.17: Swift JVM tcl-jvm.h in `turbine/code/src/tcl/jvm/`

```
1 #ifndef TCL_JVM_H
2 #define TCL_JVM_H
3 void tcl_jvm_init(Tcl_Interp* interp);
4 #endif
```

Listing 3.17 and 3.18 refer to the implementation of the C back-end that exploits the C-JVM library (see Section 3.3.1) to evaluate code in Groovy, Clojure, Scala and JavaScript. For instance, the function `Clojure_Eval_Cmd` (see Listing 3.18, lines $14-31$) allows to evaluate a string code of Groovy language. The C-JVM $clojure$ function is called (line 24) to evaluate the string code parameter (recoverd at line 19). Finally, at lines $110-113$, the four functions $clojure$, $groovy$, $scala$ and $javascript$ are exported to the Tcl environment (the C code is converted in Tcl package). Therefore, these functions can be used to define new Swift/T functions, shown in the Listing 3.19. The Listing 3.20 depicts the instructions to build the TCL module.

Listing 3.18: Swift JVM tcl-jvm.c in turbine/code/src/tcl/jvm/

```
#include "config.h"
#if HAVE_JVM_SCRIPT==1
#include "swift-t-jvm/src/swift-jvm.h"
#endif
#include <stdio.h>
#include <tcl.h>
#include <string.h>
#include <list.h>
#include "src/util/debug.h"
#include "src/tcl/util.h"
#include "tcl-jvm.h"
#if HAVE_JVM_SCRIPT==1
static int
Clojure_Eval_Cmd(ClientData cdata, Tcl_Interp *interp,
                 int objc, Tcl_Obj* const objv[])
{
  TCL_ARGS(3);
  // A chunk of Clojure code
  char* code = Tcl_GetString(objv[1]);
  // A chunk of Clojure code that returns a value
  char* expr = Tcl_GetString(objv[2]);
  clojure(code);
  // The string result from Clojure: Default is empty string
  char* s = clojure(expr);
  TCL_CONDITION(s != NULL, "clojure code failed: %s", code);
  Tcl_Obj* result = Tcl_NewStringObj(s, strlen(s));
  if (strlen(s)>0)
    free(s);
  Tcl_SetObjResult(interp, result);
  return TCL_OK;
}
static int
Groovy_Eval_Cmd(ClientData cdata, Tcl_Interp *interp,
           int objc, Tcl_Obj* const objv[])
{
  TCL_ARGS(2);
  // A chunk of Groovy code:
  char* code = Tcl_GetString(objv[1]);
  // The string result from Groovy: Default is empty string
  char* s = groovy(code);
  TCL_CONDITION(s != NULL, "groovy code failed: %s", code);
  Tcl_Obj* result = Tcl_NewStringObj(s, strlen(s));
  if (strlen(s)>0)
    free(s);
  Tcl_SetObjResult(interp, result);
  return TCL_OK;
}
static int
JavaScript_Eval_Cmd(ClientData cdata, Tcl_Interp *interp,
                    int objc, Tcl_Obj* const objv[])
```

```
51 {
52   TCL_ARGS(2);
53   // A chunk of JavaScript code:
54   char* code = Tcl_GetString(objv[1]);
55   // The string result from JavaScript: Default is empty string
56   char* s = javascript(code);
57   TCL_CONDITION(s != NULL, "javascript code failed: %s", code);
58   Tcl_Obj* result = Tcl_NewStringObj(s, strlen(s));
59   if (strlen(s)>0)
60     free(s);
61   Tcl_SetObjResult(interp, result);
62   return TCL_OK;
63 }
64 static int
65 Scala_Eval_Cmd(ClientData cdata, Tcl_Interp *interp,
66                int objc, Tcl_Obj* const objv[])
67 {
68   TCL_ARGS(2);
69   // A chunk of Scala code:
70   char* code = Tcl_GetString(objv[1]);
71   // The string result from Scala: Default is empty string
72   char* s = scala(code);
73   TCL_CONDITION(s != NULL, "scala code failed: %s", code);
74   Tcl_Obj* result = Tcl_NewStringObj(s, strlen(s));
75   if (strlen(s)>0)
76     free(s);
77   Tcl_SetObjResult(interp, result);
78   return TCL_OK;
79 }
80 #else // JVM SCRIPT disabled
81 /*
82     HIDE THIS CODE
83     It returns for each command:
84    `Turbine not compiled with JVM scripting support'
85 */
86 #endif
87 /**
88    Shorten object creation lines.  jvm:: namespace is prepended
89  */
90 #define COMMAND(tcl_function, c_function) \
91     Tcl_CreateObjCommand(interp, "jvm::" tcl_function, c_function, \
92                          NULL, NULL);
93 /**
94    Called when Tcl loads this extension
95  */
96 int DLLEXPORT
97 Tcljvm_Init(Tcl_Interp *interp)
98 {
99   if (Tcl_InitStubs(interp, TCL_VERSION, 0) == NULL)
100     return TCL_ERROR;
101
```

```
102   if (Tcl_PkgProvide(interp, "jvm", "0.1") == TCL_ERROR)
103     return TCL_ERROR;
104
105   return TCL_OK;
106 }
107 void
108 tcl_jvm_init(Tcl_Interp* interp)
109 {
110   COMMAND("clojure",    Clojure_Eval_Cmd);
111   COMMAND("groovy",     Groovy_Eval_Cmd);
112   COMMAND("javascript", JavaScript_Eval_Cmd);
113   COMMAND("scala",      Scala_Eval_Cmd);
114 }
```

Listing 3.19: Swift JVM Export jvm.swift in `turbine/code/src/export`

```
1  @dispatch=WORKER
2    (string output) clojure(string code, string expr)
3      "turbine" "0.1.0"
4      [ "set <<output>> [ jvm::clojure <<code>> <<expr>> ]" ];
5
6  @dispatch=WORKER
7    (string output) groovy(string code)
8      "turbine" "0.1.0"
9      [ "set <<output>> [ jvm::groovy <<code>> ]" ];
10
11 @dispatch=WORKER
12 (string output) javascript(string code) "turbine" "0.1.0"
13     [ "set <<output>> [ jvm::javascript <<code>> ]" ];
14
15 @dispatch=WORKER
16   (string output) scala(string code)
17     "turbine" "0.1.0"
18     [ "set <<output>> [ jvm::scala <<code>> ]" ];
```

Listing 3.20: Swift JVM module.mk.in

```
1 # MODULE TCL-JVM
2 DIR := src/tcl/jvm
3 TCL_JVM_SRC := $(DIR)/tcl-jvm.c
```

The final operation to be performed, in order to include C-JVM in Swift/T, is to include in the build process of Swift/T the new support for JVM interpreted languages. Therefore, the configure.ac and the Makefile.in of Swift/T Turbine engine have been updated. These changes are shown, respectively, in the Listing 3.21 and 3.22. This allows to configure the building of the Turbine core enabling the support for JVM interpreted languages (by using the parameter `-enable-jvm-scripting`). Furthermore, the configuration allows to specify the home path of JVM and the path for externals Java libraries.

Listing 3.21: Swift JVM Turbine configure.ac

```
...
...
# JVM scripting support: Disabled by default
HAVE_JVM_SCRIPT=0
USE_JVM_SCRIPT_HOME=0
AC_ARG_ENABLE(jvm-scripting,
    AS_HELP_STRING([--enable-jvm-scripting],
                   [Enable calling JVM scripting languages]),
    [
      HAVE_JVM_SCRIPT=1
      USE_JVM_SCRIPT_HOME=swift-t-jvm
    ])
AC_ARG_WITH(jvm-scripting,
   AS_HELP_STRING([--with-jvm-scripting],
                   [Use this JVM scripting plugin home directory]),
  [
      HAVE_JVM_SCRIPT=1
      USE_JVM_SCRIPT_HOME=${withval}
  ])
if (( ${HAVE_JVM_SCRIPT} ))
then
    AC_CHECK_FILE(${USE_JVM_SCRIPT_HOME}/src/swift-jvm.h, [],
                  [AC_MSG_ERROR([Could not find JVM scripting
                      header!])])
    AC_MSG_RESULT([JVM scripting enabled])
else
    AC_MSG_RESULT([JVM scripting disabled])
fi

AC_DEFINE_UNQUOTED([HAVE_JVM_SCRIPT],$HAVE_JVM_SCRIPT,[Enables JVM
    scripting])
AC_SUBST(HAVE_JVM_SCRIPT)
AC_SUBST(USE_JVM_SCRIPT_HOME)

#JVM HOME
AC_SUBST(JVMHOME,"/usr/lib/jvm/java-8-oracle")
AC_ARG_WITH([jvm-home],
             [AS_HELP_STRING([--with-jvm-home],
                [Set up the jvm home directory (default:
                    /usr/lib/jvm/java-8-oracle)])],
             [AC_SUBST(JVMHOME,$withval)],
             )

#JVM SWIFT-T LIBs
AC_SUBST(JVMLIB,$(pwd)"/swift-jvm/ swift-jvm-build/target/
    swift-jvm-build-0.0.1-bin/swift-jvm/classes")
AC_ARG_WITH([swift-jvm-engine-lib],
             [AS_HELP_STRING([--with-swift-jvm-engine-lib],
                [Set up the swift jvm engine lib (default: classes)])],
             [AC_SUBST(JVMLIB,$withval)],
```

```
47                 )
48 # End of JVM scripting configuration
49 ...
50 ...
```

Listing 3.22: Swift JVM Turbine Makefile.in

```
1  HAVE_JVM_SCRIPT     = @HAVE_JVM_SCRIPT@
2  USE_JVM_SCRIPT_HOME= @USE_JVM_SCRIPT_HOME@
3  ...
4  ...
5  # LIBS: links to ADLB, c-utils, MPE, and MPI
6  ...
7  ...
8  ifeq ($(HAVE_JVM_SCRIPT),1)
9    SWIFTTJVM_LIB = $(USE_JVM_SCRIPT_HOME)/src
10   LIBS += -L$(SWIFTTJVM_LIB)/.libs -lswifttjvm
11 endif
12 ...
13 ...
14 ### INCLUDES
15 ...
16 ...
17 include src/tcl/jvm/module.mk
18 ...
19 ...
20 TURBINE_SRC += $(JVM_SCRIPT_SRC)
21 TURBINE_SRC += $(TCL_JVM_SRC)
22 ...
23 ...
```

Listing 3.23 provides an example of code, which explains the usage of the JVM interpreted languages support in Swift/T.

Listing 3.23: Swift JVM test

```
1  import jvm;
2
3  s1 = groovy("println \"GROOVY WORKS\"");
4  trace(s1);
5
6  s2 = javascript("print(\"JAVASCRIPT WORKS\");");
7  trace(s2);
8
9  s3 = scala("println(\"SCALA WORKS\")");
10 trace(s3);
11
12 s4 = clojure("\"CLOJURE SETUP\"", "\"CLOJURE WORKS\"");
13 trace(s4);
```

Simulation Optimization

4

> *We assume that you are here with a computer simulation of a complicated physical model that includes several input parameters . . . the methods need too much computation in high dimensions – more computation that you have available.*
>
> **— Paul G. Constantine**
> (*Active Subspaces: Emerging Ideas for Dimension Reduction in Parameter Studies,* ***2015***)

4.1 Introduction

Complex system simulation are continuously gaining relevance in business and academic fields as powerful experimental tools for research and management, in particular for *Computational Science*. Simulations are mainly used to analyze behaviours that are too complex to be studied analytically, or too risky/expensive to be tested experimentally [Law07; TS04]. The representation of such complex systems results in a mathematical model comprising several parameters. Hence, there arises a need for tuning a simulation model, that is finding optimal parameter values which maximize the effectiveness of the model. Considering the multi-dimensionality of the parameter space, finding out the optimal parameters configuration is not an easy undertaking and requires extensive computing power. Simulations Optimization (SO) [TS04; He+10] and Model Exploration (ME) is used to refer to the techniques studied for ascertaining the parameters of the model that minimize (or maximize) given criteria (one or many), which can only be computed by performing a simulation run. This work consideres the SO process as general case of ME, where the ME is guided by some optimization algorithms.

This work is mainly focused on Agent-based models (ABMs) where the simulation is based on a large set of independent agents, interacting with each other through simple rules, generating a complex collective behaviour. What makes ABMs particularly interesting is that they allow the reproduction of complex and significant aspects of real phenomena by defining a small set of simple rules regulating how agents interact in social structures and how information is spread from agent to agent. ABMs have been successfully applied in several fields such as biology, sociology, economics, military and infrastructures – for a review of ABM applications see [MN05]. The

computer science community has responded to the need for tools and platforms that can help the development and testing of new models in each specific field by providing libraries and frameworks that speed up and make easier the tasks of developing and testing simulations. Some examples are NetLogo [TW04], MASON [Luk+04; Luk+05] and Repast [Nor+07], described in Section Agent-Based Simulation: State of Art.

It should be noted that although ABMs are governed by simple rules, interactions between agents generate network effects that lead to a high degree of complex behaviour [MN05] where it is quite hard to discern any relation between changes in variables and changes in the resulting global behaviour. In particular, the shape of the objective functions is irregular: there are large areas where changes in the parameters do not affect the final behaviour but at the same time a small change, like the butterfly effect, may provide a significant shift within a complex simulation [CH05]. To make matters even more complicated, as a consequence of the stochastic character of the simulation, a static surface does not even exist but has to be approximated by multiple simulation runs [DK14; Law07].

Moreover, a SO strategy has to cope with a high-dimensional search space and therefore has to handle a corresponding number of heterogeneous variables. Some variables configure the behavioural model of the agents while others constitute the global environment. Hence, a brute-force enumeration of all possible solutions is not feasible from a resource perspective.

In summary, complex simulations, and ABM in particular, are powerful tools for modeling aspects of real systems. On the other hand, due to the the high dimensionality of the search space, the heterogeneity of parameters, the irregular shape and the stochastic nature of the objective evaluation function, the tuning of such systems is extremely demanding from the computational point of view. This raises the need for tools, which exploit the computing power of parallel systems to improve the effectiveness and the efficiency of SO strategies. The crucial characteristics of such tools are: *zero configuration*, *ease of use*, *programmability* and *efficiency*. Zero Configuration and easiness of use are required because both the design and the use of SO strategies are performed by domain experts who seldom are computer scientists and have limited knowledge of managing modern parallel infrastructures. Programmability is mandatory because different models usually requires different SO strategies. Finally, the system must be efficient in order to be able to exploit the computing power provided by extreme scale architectures.

This Chapter discusses two framework for SO process, respectively, primarily designed for Cloud infrastructure and HPC systems.

The first frameworks is *SOF: Zero Configuration Simulation Optimization Framework on the Cloud* (discussed in the Section 4.2) and it was designed to run SO process in

the cloud. SOF is based on the Apache Hadoop infrastructure [Tur13]. The second framework is *EMEWS, Extreme-scale Model Exploration with Swift/T* (discussed in the Section 4.3) designed at Argonne National Laboratory (USA). EMEWS as SOF allows to perform SO process in distributed memory architectures. Both the framework have been designed for and tested on ABS. In particular EMEWS was tested using the ABS simulation toolkit Repast. Initially, EMEWS was not able to execute out of the box simulations written in MASON and NetLogo [TW04]. This dissertation presents the novel functionalities of EMEWS that enable to execute MASON and NetLogo simulations in EMEWS.

4.1.1 Model Exploration and Simulation Optimization

A Simulation is an attempt to reproduce the behaviour of a real-life process or system over time. A system is understood as a collection interacting on entities [Ack71]. [Law07] defines a simulation as "numerically exercising the model for the inputs in question to see how they affect the output measures of performance". Hence, the goal of a simulation is to experiment, observe, understand, infer and answer "what if" questions about a complex system described with a model. Simulations can either be used to design a novel system or for predicting the effect of changes in an existing system [CL10]. The main advantage of simulation is that it can be used to explore certain behaviour without causing disruption in the actual system.

The simulation modeler usually needs to execute a large number of simulations in order to find the optimal configuration of input parameters (that is the configuration which allows them to imitate the desired system). This process is named parameter space exploration (PSE), parameter sweep or Model Exploration (ME). Simulation results are evaluated using an objective (evaluating) function which associates a score with each simulation performed with a given set of parameters. As the number of the parameters of a model increases, the parameter space to be explored expands exponentially and it becomes unfeasible to handle the parameter space exploration process – which comprises parameters selection, simulations run and output evaluations – manually. Moreover, the feedback obtained from the simulation of previous configurations can be used to select future configurations to be simulated and evaluated. This cyclic process: a) choice of initial configurations, b) execution, c) evaluation and d) selection of new candidates, is referred to as Simulation Optimization (SO) process.

Simulation Optimization Problem definition

The simulation optimization (SO) problem [CL10; TS04; Amm+11; Nel10] can be presented as

$$\min_{x \in D} \Gamma(x),$$

where $D \subseteq \Theta$ is the feasible decision space, Θ is the whole parameters space, $x \in D$ is an 1-dimensional vector having size δ representing a single configuration (δ is the number of parameters of the simulation), and $\Gamma(x)$ is a function being estimated with simulation. The feasible decision space D can be either discrete or continuous. A user library implements a particular SO algorithm and the selection of points in decision space is usually handled within algorithms. Hence, it is not necessary to assume either a continuous or discrete decision space in advance. This work is not focused on introducing or evaluating new ways to explore the parameters space. Instead, is focused on a support tool to perform the parameters space exploration using an SO approach.

Generally, the problem has a single objective (i.e., $\Gamma(x) \in R$), however multi-objective optimization problems ($\Gamma(x) \in R^n$) can be also considered. For the remainder of the Chapter, single-objective optimization problems are considered; however, the proposed methodology can easily be applied to multi-objective optimization in a similar fashion, as described in [BM05]. The stochastic nature of simulation means that the output of a simulation run is not deterministic and we calculate an expected value for it as $\mathbb{E}[\Phi(x, \epsilon)]$, where $\Phi(x, \epsilon)$ is the result of a stochastic simulation run on configuration x and a random feed ϵ. Finally we calculate $\Gamma(x) = f(\mathbb{E}[\Phi(x, \epsilon)])$, where $f(\cdot)$ is a function that evaluates the result of a simulation and calculates a single rank value. For instance, in [CL10], the value of $\mathbb{E}[\cdot]$ is estimated as a mean result of $r \geq 1$ simulation runs.

4.1.2 State of Art for ABM

Capabilities of existing ABM tools. Many ABM toolkits allow users to define a parameter space and then enable automated iteration through that space. The parameter space is defined in a toolkit-specific format specifying each parameter in terms of a range, and a step value, or as a list of elements. The toolkit takes this *a priori* determined parameter space as input and executes the required simulation runs. The model exploration advanced capabilities of the most popular ABM toolkits are:

- *Repast Simphony*, [MN+13] is the Java based toolkit of the Repast Suite. Given a parameter space as input, Repast Simphony's batch run functionality can divide that space into discrete sets of parameter values and execute simulations over those discrete sets in parallel. The simulations can be run on a local machine, on remote machines accessible through secure shell (ssh), in the cloud (e.g., Amazon EC2) or on some combination of the three. Using an InstanceRunner interface, Repast Simphony models can be launched by other control applications such as a bash, Portable Batch System (PBS), or Swift scripts. For example, [Ozi+14] describes how the InstanceRunner can be used

with Swift to perform an adaptive parameter sweep using simulated annealing, and the InstanceRunner is used in three use cases (see Section 4.3.2).

- *NetLogo* [TW04] has both GUI (BehaviorSpace [Beh]) and command line based batch run capabilities. [Sto11] developed the Behavior Search tool which provides an easy to use interface for running an included set of heuristic model exploration techniques, e.g., simulated annealing, genetic algorithm, on NetLogo models.
- *MASON* simulation library [Luk+05] offers a set of capabilities for creating ABMs. Its modularity allows MASON models to be called either via the command line or as libraries from Java-based programs for model exploration purposes.
- *AnyLogic* is a proprietary multi-method simulation toolkit. AnyLogic comes with the ability to carry out *Experiments*, including optimization, calibration, and user-defined custom experiments using the AnyLogic engine Java API. (Note that many of the experiment capabilities are available only for AnyLogic Professional and University Researcher editions.)

*None of the ABM toolkits on their own offer the capabilities or scope, in terms of flexible, simple integration of external model exploration tools and performance on massively parallel computing resources, that **SOF** and **EMEWS** framework aim to provide.*

Model exploration libraries and frameworks. In the following, the existing model exploration (ME) or in general simulation optimization (SO) libraries and frameworks are briefly discussed. While most can be used as standalone ME tools, some of the these libraries can also be used as ME modules within the presented frameworks. Most of the following software falls under the metaheuristics umbrella. For an overview of metaheuristics see [Luk13], for reviews of more metaheuristics frameworks see [Par+11] and for parallel metaheuristics frameworks see [Alb+13].

- OpenMOLE [Reu+13], provides an execution platform that distributes simulation experiments on high performance computing environments using a domain–specific language (DSL) that is an extension of the Scala programming language.
- Model Exploration ModulE (MEME) [Gul+11], is based on virtual hosts specially prepared for simulation experiments, deployed on EC2 (the Amazon Elastic Cloud).
- OptTek [opt] offers proprietary tools for metaheuristic optimization capabilities and the ability to wrap custom objective functions. OptTek's optimization engine is also directly integrated into a number of ABM and simulation tools (e.g., AnyLogic, Simio). While parallel optimization capabilities are available, the focus of OptTek products appears to be on desktop applications.
- ECJ [ECJ] is an open source (AFL v3) research system for evolutionary computation. ECJ can be used for developing evolutionary algorithms and general

integration of simulation code on massively parallel systems. ECJ is able to integrate Java based simulation code (e.g., written with MASON or Repast Simphony) and the evolutionary algorithms can be parallelized in various ways,

- ParadisEO [Par], published under the CeCILL license, and MALLBA, published under a non-commercial license, are framework for metaheuristics. They include extensive metaheuristics capabilities, including parallel metaheuristics methods. Similarly to ECJ, the general integration of external codes is not the focus.
- Dakota [Dak] combines a number of optimization, design of experiment and uncertainty quantification libraries developed by Sandia National Laboratories (e.g., DDACE, HOPSPACK), in addition to other external libraries. Dakota can be used on machines from desktops to massively parallel computers.

4.2 Simulation Optimization and exploration Framework on the cloud

Observing the computation workflow needed for an SO process, it is clear that looks like an instance of the well known bag-of-tasks application [Adl+03], i.e., an application made of a collection of independent tasks, to be scheduled on a master–worker platform. Nevertheless, a mechanism for the spread of the task and the collection of results is required.

Hence this framework was designed exploiting the assumption that SO processes can be easily deployed by exploiting the MapReduce (MP) programming model. Moreover, an SO process potentially requires several optimization loops in which a large amount of data is generated. The amount of inputs and outputs generated in a SO process, that must be managed in a distributed storage environment, is usually quite large. The MapReduce paradigm and Apache Hadoop will be briefly described in the following.

MapReduce Paradigm Overview. MP [DG08] is a programming model, proposed by Google, for processing large data sets exploiting parallel/distributed computations on a set of loosely coupled machines. MP is based on two principal functions named *Map* and *Reduce*, commonly used in functional programming languages such as Lisp. Each function takes an input pair expressed as key/value to compute some transformation on it. The Map function produces a sets of intermediate results while the Reduce function merges the intermediate results in a new set of key/value. Historically, MP has been used for indexing and calculating PageRank, but since its creation the research community adopted the programming model for several purposes, in particular when the amount of computation is large and the whole computation can be easily decomposed into smaller independent tasks.

Apache Hadoop. Apache Hadoop is an open-source alternative to the Google technologies: *Google File System* [Ghe+03] and *MapReduce* [DG08]. Hadoop is the top-level of many subprojects comprising *Hadoop Distributed File System* (HDFS) and *MapReduce*.

HDFS is a distributed filesystem that enables storage of a huge dataset across a distributed system. HDFS is designed to accommodate the following requirements [Shv+10]: *Large Data Sets*, *Simple Coherency Model*, *Moving Computation is Cheaper than Moving Data* (it attempts to assign a computation to a node that maintains the data instead of move the data around the nodes), *Portability Across Heterogeneous Hardware and Software Platforms* and *Hardware Failure*.

Hadoop defines a specification for the Map and Reduce functions, the developers must provide the input/output specific and the implementations of Map and Reduce

functions, often referred as mappers and reducers. Then the framework manages all the functionality needed to run an MP application as job execution, parallelization, and coordination. A typical MP program, on Hadoop, starts on a single node that launches and manages the execution of the entire distributed program on the distributed system. Then several components operate at different stages:

- *Splitter,* handle the single data source providing input pairs (key/value) to mappers.
- *Mapper,* process a key/value pair to generate a set of intermediate key/value pairs.
- *Combiner,* also called "Local Reducer" (optional). It can help cutting down the amount of data exchanged between Mappers and Reducers.
- *Partitioner,* also called the "Shuffle Operation". It ensures that records with the same key will be assigned to the same Reducer.
- *Reducer,* gathers the results of the computation and concludes the job giving outputs the new set of key/value, typically stored in the HDFS.

4.2.1 Architecture

This Section presents the *Simulation exploration and Optimization Framework on the cloud* (SOF), a framework that allows us to run and collect results for two kinds of optimization scenarios: parameter space exploration or model exploration (PSE or ME) and simulation optimization (SO).

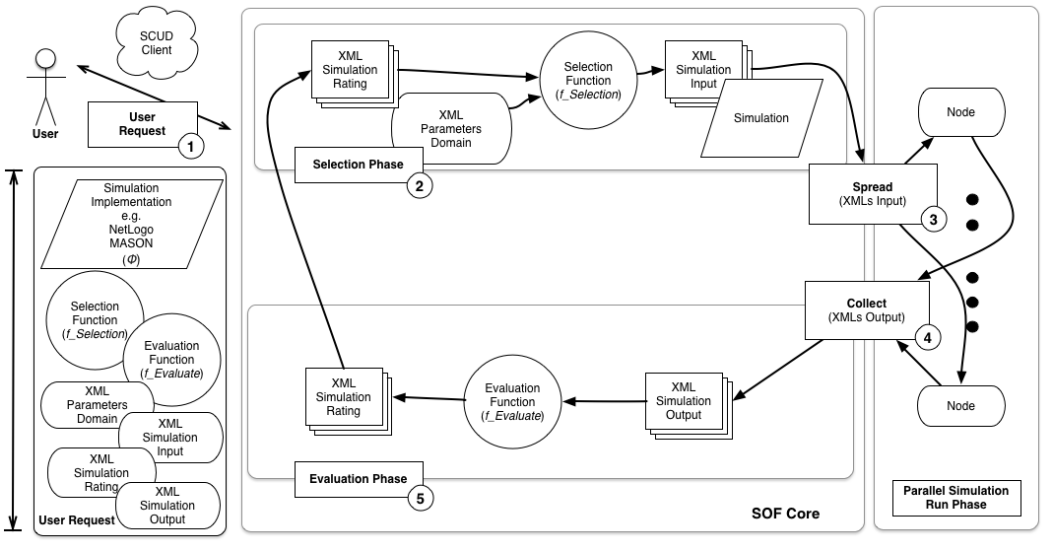

Fig. 4.1.: SOF Work Cycle.

Figure 4.1 depicts the SOF work cycle which comprises three phases: selection, parallel simulations and evaluation. SOF provides a set of functionality that allows developers to construct their own simulation optimization strategy. The framework was designed under the following objectives:

- *zero configuration:* the framework neither requires the installation nor the configuration of any additional software, only Hadoop and a SSH access to the hosting platform are required;
- *ease of use:* the tool is transparent to the user, since the user is totally unaware that system operates on a distributed environment;
- *programmability:* both the simulation implementation and the simulation optimization functionalities can be implemented using different simulation toolkits (MASON, NetLogo, etc.) and/or by exploiting different programming languages, provided that the hosting platform supports them;
- *scalability:* by executing several independent tasks (simulations) concurrently, the framework adequately exploits the resources available on the hosting platforms.

SOF uses a particular simulation optimization scenario described in the following. Two algorithms – inspired by evolutionary algorithms [Cas06] – allow to define the scenarios achievable in SOF. The following symbols are used in the algorithms description:

- Θ, parameters space;
- $D \subseteq \Theta$, feasible decision space;
- $\mathbb{X} \subseteq D$ is a set of configurations from the feasible decision space D, $\mathbb{X} = \{x_1, x_2, \ldots : x_i \in D\}$;
- r denotes the number of simulation run;
- $\Phi(x, \epsilon)$ denotes the results of a stochastic simulation run on configuration x and a random feed ϵ;
- $\mathbb{E}[\cdots]$ denotes the expected results of a set of stochastic simulation run;
- $\mathbb{Y}$ is the set of expected simulation results corresponding to the configuration in $\mathbb{X}$.
- t is the current optimization loop;
- $\mathbb{\Gamma}$ contains the ranking values associated to the configurations in $\mathbb{X}$;

Algorithm 4.2.1: `PSE()`

INPUT: $\mathbb{X}, \Phi(\cdot,\cdot)$
OUTPUT: $\mathbb{Y}$

parallel
- **for each** $x_i \in \mathbb{X}$
  - **do**
    - **for** $j \leftarrow 1$ **to** r
      - **do** $Z_j \leftarrow \Phi(x_i, \epsilon_j)$
    - $Y_i \leftarrow \mathbb{E}[Z_1, Z_2, \ldots, Z_r]$
- $\mathbb{Y} = \{Y_1, Y_2, \ldots\}$

PSE Algorithm (PSE). The PSE or ME scenario describes a generic process of simulation optimization where a fixed set of configuration $\mathbb{X}$ is executed and all the

corresponding results are collected. The algorithm 4.2.1 performs this task and also handles stochastic simulations, which require several execution runs (with different random seed) and the estimate of the expected values. The algorithm 4.2.1 performs r simulations for each point $x_i \in \mathbb{X}$ and collects the simulation results in a set $\mathbb{Y}$.

Algorithm 4.2.2: SO()

INPUT: $D, \Phi(\cdot,\cdot), f\_Selection(\cdot,\cdot,\cdot), f\_Evaluate(\cdot)$
OUTPUT: $\{\mathbb{Y}_1, \mathbb{Y}_2, \ldots\}, \{\mathbb{\Gamma}_1, \mathbb{\Gamma}_1, \ldots\}$
$t = 1$
while $((\mathbb{X}_t = f\_Selection(D, \{\mathbb{X}_1, ..., \mathbb{X}_{t-1}\}, \{\mathbb{\Gamma}_1, ..., \mathbb{\Gamma}_{t-1}\})! = \emptyset)$
- **parallel**
  - **for each** $x_i \in \mathbb{X}_t$
    - **do**
      - **for** $j \leftarrow 1$ **to** r
        - **do** $Z_j \leftarrow \Phi(x_i, \epsilon_j)$
      - $Y_i \leftarrow \mathbb{E}[Z_1, Z_2, \ldots, Z_r]$
  - $\mathbb{Y}_t = \{Y_1, Y_2, \ldots\}$
  - **for each** $Y_i \in \mathbb{Y}_t$
    - **do** $\Gamma_i \leftarrow f\_Evaluate(Y_i)$
  - $\mathbb{\Gamma}_t = \{\Gamma_1, \Gamma_2, \ldots\}$
- $t = t + 1$

SO Algorithm (SO). The simulation optimization is a general case of PSE and corresponds to the execution of several loops of the PSE algorithm. For each optimization loop t, the set of configurations to be executed and evaluated, $\mathbb{X}_t$, depends on the results obtained from the previous loops. The algorithm 4.2.2 describes the SO process. Each optimization loop uses the function $f\_Selection(\cdot,\cdot,\cdot)$ to generate a novel set of configurations. At the end of each loop, the function $f\_Evaluate(\cdot)$ computes the ranking values associated with each configuration in $\mathbb{X}_t$. The SO algorithm ends when the selection function returns an empty set.

A *contributor* who implements a SO package needs to provide a functional mechanism for the definition of the feasible decision space D and the implementation of both the $f\_Selection$ and $f\_Evaluate$ functions.

Then the modeler wishing to use an implementation of SO developed for SOF must provide:

- the definition of D according to the mechanism provided by the SO package;
- a stochastic simulation model $\Phi(x, \epsilon)$;
- all the parameters required by the $f\_Selection(\ldots)$ and $f\_Evaluate(\ldots)$ functions.

4.2.2 Working Cycle

The SOF architecture has been designed according to the *Work Cycle* shown in Figure 4.1 and the algorithms shown in section 4.2.1. The framework is divided into three functional blocks: *the User Front-end* (Figure 4.1, left); *the SOF core* which act as a controller (Figure 4.1, middle); *the computational resources* (Figure 4.1, right).

The User front-end is implemented as a web or a standalone application through which the user provides the inputs to the system: *Simulation Implementation*, *Selection Function*, *Evaluation Function*. In order to ensure flexibility, an XML schema for the description of Domain (*Parameters Domain*), Input (*Simulation Input*), Output (*Simulation Output*) and Rating (*Simulation Rating*) has been included in the request to the system. The application level of SOF provides a tool to easily generate the needed XML files. The execution of the system is described by the loop shown in Figure 4.1. We summarize it in the following key phases:

1. ***User Request***. The user submits the *Simulation Implementation*, the *Selection Function* and the *Evaluation Function* written using any language supported by the cloud environment. Then s/he defines the Parameters Domain, the Simulation Input, Output and Rating format in XML using the SOF XML schema.
2. ***Selection***. The system processes the request using the *Selection Function* and generates a set of parameters according to the XML schema defined by the user.
3. ***Spread***. The generated XML inputs are dynamically assigned to the computational resources. We notice that our system delegates to the distributed computing environment (Hadoop in our case) both scheduling and load balancing of tasks (simulations).
4. ***Collect***. When all the simulations run terminated, the computation state is synchronized and the outputs are collected according to the XML schema defined by the user, through a set of messages exchanged between the computational resources and the system.
5. ***Evaluation Phase***. The system applies the evaluation function on the collected outputs and generates the rating (again in the desired XML format).

After the evaluation phase, the system goes back to the selection phase, which, also using the evaluation results obtained during the preceding steps, generates a new set of XML inputs. Obviously, the selection function also includes a stopping rule which allows to end the SO process.

During the spread phases, the framework executes a large number of simulations in order to achieve the results of a PSE or a SO scenario. The challenge is "How to elaborate a large number of inputs, on a distributed system, in order to ensure fault tolerance and good performance, even for different SO processes running

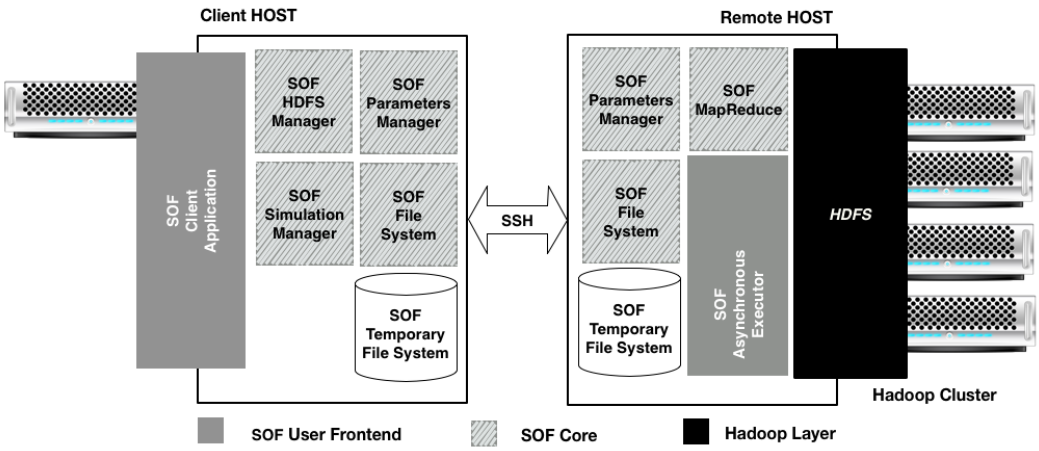

Fig. 4.2.: SOF - Hadoop Architecture

concurrently?". SOF exploits the Apache Hadoop framework to answer to this question.

Apache Hadoop, briefly described in Section 4.2, provides some tools for managing MapReduce applications and the HDFS File System. It also provides a set of Java libraries for writing MapReduce applications. According to the language used by the Simulation Implementation, it will be possible to run the MapReduce application in several ways. For instance, when the implementation is written in Java (e.g, MASON, Repast Simphony) is it possible to write a MapReduce application that initializes the simulation at code level by using mechanisms like Java Reflection. Other frameworks, like Netlogo, provide a Java library for executing simulations from a Java application. Eventually, in the case of generic implementations, the setting of simulation parameters is performed using the Java Runtime to set the input as command line arguments of the executable.

4.2.3 Software Layers

As shown in Figure 4.2, the system workflow presents two main entities: the SOF client and the remote host (on which is installed Apache Hadoop). The SOF architecture is divided into three main software components:

- the SOF front-end (client side), which is the SOF application for running and managing the simulation on the Hadoop infrastructure;
- the Hadoop layer (remote side), which comprises software and libraries provided from Hadoop infrastructure;
- the SOF core, composed of six functional blocks, that are used on both the client and the remote side.

The core. The main objective of the SOF core is to ensure the flexibility in terms of the ability to use any Hadoop installation on-the-fly without requiring a specific

configuration of Hadoop infrastructure or a particular software installation on the remote host. The SOF core uses the Secure SHell (SSH) protocol for the communication between client and remote host to ensure the highest flexibility level and a secure consolidated communication mechanism. With more details, the SOF core comprises:

- *Parameters Manager:* defines the XML schema of the *Parameters Domain Definition*, *Simulation Input Definition*, *Simulation Output Definition* and *Simulation Rating Definition*. It also provides routines for creating, managing and verifying the XML files.
- *File System:* defines the structure of the *SOF Environment*, that is the directories hierarchy on the HDFS, Remote and Client hosts. This block exposes routines to get the paths of a simulation, a simulation loop or for temporary files and folders on both the remote and client hosts.
- *HDFS Manager:* is responsible for monitoring and creating files on the HDFS.
- *MapReduce (SOF process) and Asynchronous Executor (SOF-RUNNER):* allow execution of the SO algorithm on a Hadoop environment.
- *Simulation Manager:* is the fundamental block in the SOF architecture and provides the routines for executing and monitoring simulations. This block uses SSH to invoke an asynchronous execution of the SOF-RUNNER. When an SO process is started, the remote process ID is stored in the XML simulation descriptor file on the HDFS. In this way it is always possible to monitor the SO process on the remote machine and it is also possible to stop/restart or abort the SO process.

The interactions with Hadoop. SOF was designed under the assumption that the remote host is a Unix machine. Therefore, the interactions between client, remote host and Hadoop system are made using SSH and Unix commands. An important contribution of SOF is that presents a novel approach to managing SO processes by embedding them in the MapReduce paradigm. SOF is focused on ABM simulations, hence considers three types of simulation frameworks: MASON, NetLogo and a generic. The first two are the most relevant ABM frameworks in the ABM community; the last refers to any application executable on the computation host.

Following are described in detail the interaction between the SOF core and Hadoop. The main events in the system are:

- *User Login:* After the user login on the Remote machine, the system automatically builds a new *SOF Environment* on the Remote machine and the HDFS and copies two programs onto the remote machine: SOF and SOF-RUNNER. SOF is the MapReduce application specialized for execution, on Hadoop, of MASON, NetLogo or a generic simulation framework. SOF-RUNNER is the SOF process manager, this process is responsible for executing the PSE or SO algorithm

exploiting the SOF MapReduce application. The Simulation Environment allows the storage of all request and output files for the simulation process. The structure of the Simulation environment is defined by the SOF *File System;*

- *Simulation Creation:* The user prepares the simulation environment exploiting the features provided by the SOF frontend. Then subsequently, all simulation files: *Simulation Implementation*, *Selection Function*, *Evaluation Function*, *Parameters Domain Definition*, *Simulation Input Definition*, *Simulation Output Definition* and *Simulation Rating Definition* are copied onto the HDFS using a structure defined by the SOF File System;
- *Simulation Submission:* The SOF Core provides a routine to run a new process that launches the SOF-RUNNER via SSH on a particular simulation. The SOF-RUNNER executes the PSE or SO algorithm exploiting the *Selection Function* and the SOF MapReduce application on the Hadoop infrastructure, for parallel executions of the *Simulation Implementation* and the Evaluation Phase.

Due to the asynchronous nature of the system and decoupling from the Hadoop infrastructure, all states of the processes are visible only by reading the state of the *SOF Environment*, which comprises the Simulation Environment of all the SO processes in the system. On the HDFS, the SOF Environment contains the state of the simulation and the state of the optimization loop for any SO process. On the Remote machine the SOF Environment stores the state of the SOF-RUNNER process: in this way, it is also possible to stop/restart or abort any SO process. SOF has been designed for the concurrent optimization of different simulations performed by one or many users. In order to avoid the concurrency issues, SOF uses a separated Simulation Environment with a unique identifier for each SO process.

The MapReduce embedding. The computation schema of an SO process, as mentioned above, looks like the well known paradigm MapReduce. In particular, we consider the parameter space Θ as a dataset of Key/Value, where the Keys are the input IDs and the Values are a feasible set of values for the simulation parameters. MP-SOF application consists of two main transformations of the inputs: transform the input $x \in \Theta$ in the simulation output $\Phi(x)$ and evaluate the simulation output $f(\Phi(x))$. The first transformation requires the execution of the simulation on the desired set of parameters, while the second transformation evaluates the output using an evaluating function ($f\_Evaluate(\cdot)$) according the SO adopted scenario (see Section 4.2.1).

The SOF-RUNNER, depicted in Fig. 4.3, performs the following steps:

1. First, it executes the selection function using the Java Runtime. The selection function takes three arguments: the path to the input sets already executed, the path to the rating corresponding to the input sets executed and the path where the function creates the novel input set.

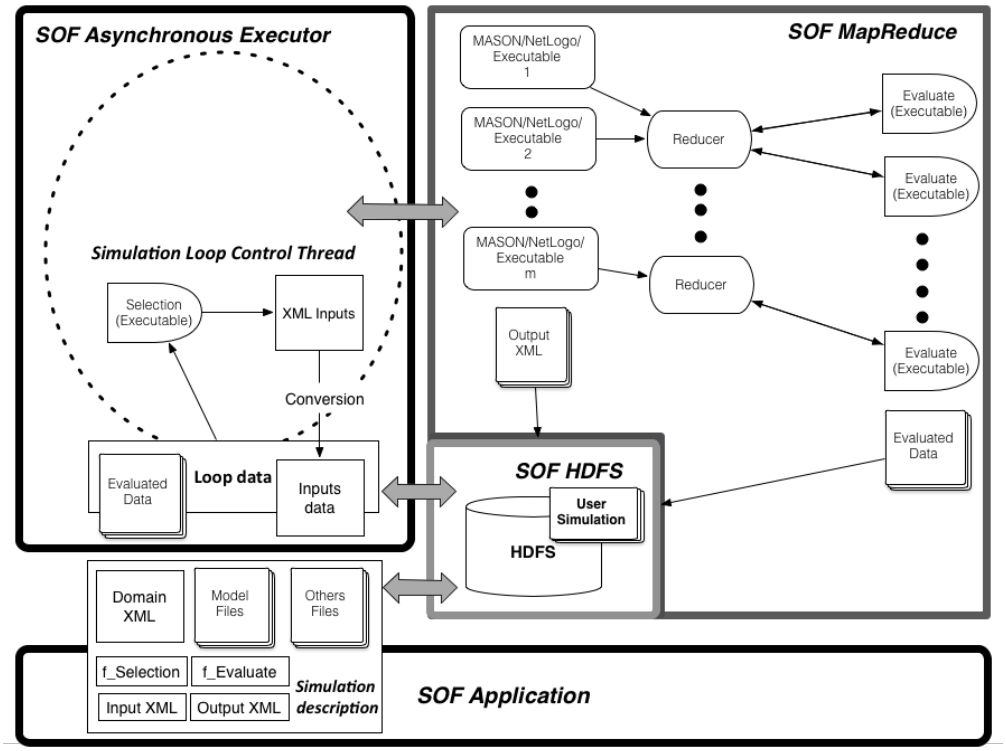

Fig. 4.3.: Simulation Optimization embedding in MP.

2. When the selection function ends, the SOF-RUNNER transforms the input set from XML format to a standard format for Hadoop MapReduce application and copies it on the HDFS (this is not strictly necessary because Hadoop in the latest version support also XML input but, to ensure compatibility with a larger number of Hadoop cluster, we preferred to use a standard format).
3. Then it launches the SOF MapReduce application. The MapReduce application (SOF) consists of two main routines *map* and *reduce* as described in the section 4.2:
   a) the map routine corresponds to executing the simulation and generating an output XML file, which represents the final state of the executed simulation;
   b) the reduce executes the evaluation function, using the Java Runtime. The evaluation function takes two arguments: the path of the output XML files and the path where the function creates the rating XML file.
4. When the evaluation function ends the reducer puts the rating XML file on the HDFS.

The mapper routine should be specialized according to the specific simulation framework used. For MASON and NetLogo, the system can automatically read the final state of the simulation and generate the output XML. In order to ensure this kind of functionality, the system requires the definition of the XML format for the input and the output files. Such files do not contain any values for the parameters

but are needed just to inform the system about the names of the parameters to be initialized at the beginning of the simulation and to be returned at the end. In the case of a generic simulation framework, the modeler of the simulation is responsible for generating the output files using a specific XML format in a predefined output folder. *It is worth mentioning that, although the system is designed for Hadoop, by changing the SOF application, it is still possible to use the system on other environments for distributed computing [Adl+03].*

4.2.4 Evaluation

This Section describes the benchmarks used to evaluate SOF *scalability* and give the results obtained on an Hadoop cluster machine.

The Benchmark Datasets. We have tested a simple SO process test on NetLogo Fire model [Wil97]. This model simulates the spread of a fire through a forest. It shows that the fire's chance of reaching the right edge of the forest depends critically on the density of trees. This is an example of a common feature of complex systems, the presence of a non-linear threshold or critical parameter. In particular, at 59% density, the fire has a $50/50$ chance of reaching the right edge. The Fire model has also been used to validate the OpenMOLE platform [Reu+13].

Since we are evaluating the performance of the framework, the SO process is based on an empty $f\_Evaluate(\cdot)$ function while the $f\_Selection(\cdot,\cdot,\cdot)$ function generates a set of n configurations for the first 10 loops and an empty set, at the end of the 10^{th} loop, so that the SO process terminates. Each configuration consists of the density parameter and a seed for the random generator. All the simulations perform 1000 simulation steps.

The Simulation Environment. Simulations have been performed on an Hadoop cluster of four nodes, each equipped as follows:

- Hardware:
  - CPUs: 2 x Intel(R) Xeon(R) CPU E5-2680 @ 2.70GHz (#core 16, #threads 32)
  - RAM: 256 GB
  - Network: adapters Intel Corporation I350 Gigabit
- Software:
  - Ubuntu 12.04.4 LTS (GNU/Linux 3.11.0-15-generic x86_64)
  - Java JDK 1.6.25
  - Apache Hadoop 2.4.0

Experimental settings. Fifteen test setting experiments were performed varying both the number of cluster nodes ($p \in \{1, 2, 4\}$) and the number of configuration generated per loop ($n \in \{2000, 4000, 8000, 16000, 32000\}$). Such values have been

selected in order to be able to evaluate both the strong an weak scalability of the framework. Table 4.1 depicts the completion time in seconds required for the execution of each test. Results show that the system scales pretty well especially when the number of configurations is large. This result was not surprising since the tasks are all independent and do not generate any communication overhead.

p / n	**2000**	**4000**	**8000**	**16000**	**32000**
1	1817	3109	6615	12516	25656
2	1330	2517	3620	6562	13420
4	1058	1440	2471	4093	7854

Tab. 4.1.: Completion time (s) with different test settings where n is the number of simulation performed per loop and p is the number of cluster nodes.

Strong and Weak Scalability. In order to better evaluate the scalability efficiency of the framework, the weak and strong scaling efficiency (described in Section 1.4) test were computed. The strong scaling efficiency measures the capability of the framework to complete a set of simulations in a reasonable amount of time. In order to compute the strong scalability efficiency the problem size stays fixed but the number of processing elements are increased. The strong scalability efficiency (as percentage of the optimum) is given by $(t_1 \times 100)/(p \times t_p)\%$ where t_i denotes the completion time to perform the overall set of simulations using i cluster nodes. In our cases the strong scaling efficiency ranges from 34.16% ($p = 4, n = 2000$) to 95.59% ($p = 2, n = 32000$). The weak scaling efficiency measures the capability of the framework to solve larger problems as the number of processing element increases. In order to compute the weak scalability efficiency the problem size assigned to each processing element stays constant and additional elements are used to solve a larger problem. The weak scalability efficiency (as percentage of the optimum) is given by $(t_1/t_p) \times 100\%$ and in our cases is equal to 100% for $p = 2$ and 84.22% for $p = 4$. The speedup, on the larger problem ($n = 32000$), is 1.91 for $p = 2$ and 3.27 for $p = 4$.

4.3 EMEWS: Extreme-scale Model Exploration With Swift/T

Extreme-scale Model Exploration With Swift/T (**EMEWS**), uses the general-purpose parallel scripting language Swift [Arm+14] (described in Chapter 3) to generate highly concurrent simulation workflows. These workflows enable the integration of external ME algorithms to coordinate the running and evaluation of large numbers of simulations, these allow also to easily implement SO process. The general-purpose nature of the programming model allows the user to supplement the workflows with additional analysis and post-processing as well.

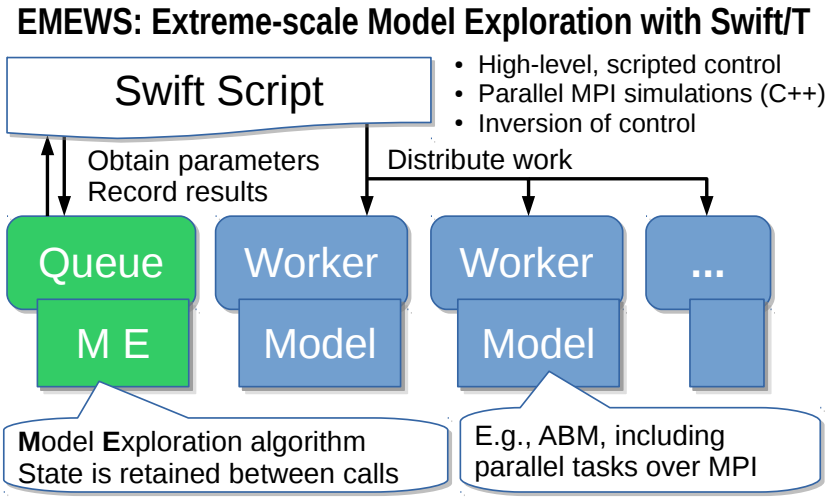

Fig. 4.4.: Overview of Extreme-scale Model Exploration with Swift/T (EMEWS) framework.

Figure 4.4 illustrates the main components of the EMEWS framework. The main user interface is the Swift script, a high-level program. The core novel contributions of EMEWS are shown in green, these allow the Swift script to access a running ME algorithm. This algorithm can be expressed in Python, R, C, C++, Fortran, Julia, Tcl, or any language supported by Swift/T.

EMEWS provides a high-level queue-like interface with (currently) two implementations: EQ/Py and EQ/R (EMEWS Queues for Python and R). These allow the Swift script to obtain candidate model parameter inputs and return model outputs to the ME. The simulation models are distributed by the Swift/T runtime over a potentially large computer system, but smaller systems that run one model at a time are also supported. The simulations can be implemented as external applications called through the shell, or in-memory libraries accessed directly by Swift (for faster invocation).

EMEWS thus offers the following contributions to the science and practice of simulation ME studies:

1. it offers the capability to run very large, highly concurrent ensembles of simulations of varying types;
2. it supports a wide class of model exploration algorithms, including those increasingly available to the community via Python and R libraries;
3. it offers a software sustainability solution, in that simulation studies based around EMEWS can easily be compared and distributed.

EMEWS as SOF is focused on agent-based models (ABMs). Extracting knowledge from ABMs requires the use of approximate, heuristic ME methods involving large simulation ensembles. To improve the current state of the art it has been noted elsewhere that: "... there is a clear need to provide software frameworks for meta-heuristics that promote software reuse and reduce developmental effort." [I.+13].

The EMEWS design aims to ease software integration while providing scalability to the largest scale (petascale plus) supercomputers, running millions of ABMs, thousands at a time. Initial scaling studies of EMEWS have shown robust scalability [Ozi+15]. The tools are also easy to install and run on an ordinary laptop, requiring only an MPI (Message Passing Interface) implementation, which can be easily obtained from common OS package repositories.

Section 4.3.2 describes three EMEWS use cases, which exploit ME algorithm to optimize the results of Repast simulation. This use cases are published on a public repository.

4.3.1 ABM integrations in EMEWS

EMEWS initially supports Repast and generic executable simulations. Two applications (wrappers) to easily integrate MASON and NetLogo simulations are described in the following. The use cases that are described in Section 4.3.2 are also implemented using MASON and NetLogo simulations and are published on public repository (*mutisim* branch).

The Repast simulator allows to easily execute Repast simulation from command line. The Listing 4.1 depicts an example $Bash$ script to execute Repast simulation. As shown, the most attractive feature of Repast is that it allows to run a particular parameters configuration (`param_line`) of a simulation model (`-scenario '$scenario'`).

MASON and NetLogo do not provide the same feature. Hence, it is quite complex to integrate these ABM toolkits in frameworks as EMEWS or SOF. In SOF the integration has been made by a programming level approach (the parameters configuration is loaded from Java code); On the other hand, EMEWS is based on Swift/T language

Listing 4.1: Repast script to execute simulation

```
#!/bin/bash
set -eu
# The upf param line,simulation arguments
param_line=$1

# The instance directory
instanceDir=$2

#root directory of the project
tproot=$3

#directory of the simulation model
modelDir=$tproot"/complete_model/"

#mkdir $instanceDir # since the swift script is making this
cd $instanceDir
ln -s $modelDir"data" data

cPath=$modelDir"lib/*"
pxml=$modelDir"scenario.rs/batch_params.xml"
scenario=$modelDir"scenario.rs"

if [[ ${JAVA:-0} == 0 ]]
then
  JAVA=java
fi

#Execute Repast Simphony
$JAVA -Xmx1536m -XX:-UseLargePages -cp "$cPath"
    repast.simphony.batch.InstanceRunner -pxml "$pxml" -scenario
    "$scenario" -id 1 "$param_line"
```

Listing 4.2: MASON script to execute simulation

```
1  #!/bin/bash
2
3  set -eu
4
5  # Args
6  param_line=$1   # The upf param line
7  #echo "$param_line"
8  instanceDir=$2  # The instance directory
9  tproot=$3
10
11
12 if [[ ${JAVA:-0} == 0 ]]
13 then
14   JAVA=java
15 fi
16
17 echo "INPUT" $param_line
18
19 $JAVA -Xmx1536m -XX:-UseLargePages
20 -jar $tproot/mason_model/mason-1.0-wrapper.jar
21 -m $tproot/mason_model/JZombieMason.jar
22 -simstate it.isislab.swiftlang.abm.mason.zombie.JZombie
23 -outfile $tproot/swift/$instanceDir/counts.csv
24 -runid 1
25 -s 150
26 -trial 1
27 -i $param_line
28 -o human_count,zombie_count
```

which does not allow to execute Java pure code, and consequently needs a different strategy. Two Java Wrappers (for MASON, see Listing 4.2 and NetLogo, see Listing 4.3) have been designed to add on EMEWS the support for both the MASON and NetLogo, with the same features available for Repast. Chapter 3, describes the new functionalities of Swift/T to invoke languages based for JVM (available from Swift/T 1.0), these new functionalities allow to configure the simulation by code, instead of executing bash script.

Two Java wrappers[1] were designed to face to this problem and are (the project is released under MIT License). Both the wrappers use the ABM toolkits Java API in order to create, initialize and execute new simulations. The wrappers usage is pretty similar, and change according the ABM toolkits characteristics. For instance the simulation model in MASON is specified using the full qualified name (parameter `-simstate`) of the MASON model class and the the simulation (parameter `-m`) is defined by a *Jar* file (see line $21-22$ Listing 4.2). Instead, in NetLogo (see line 21 Listing 4.3) the simulation (parameter `-m`) is a **.nlogo* file, which describes also the model. Both the wrappers allow to take the parameters configuration string (CSV

[1] Available on `https://github.com/spagnuolocarmine/swiftlangabm` public repository.

Listing 4.3: NetLogo script to execute simulation

```
#!/bin/bash

set -eu

# Args
param_line=$1   # The upf param line
#echo "$param_line"
instanceDir=$2  # The instance directory
tproot=$3


if [[ ${JAVA:-0} == 0 ]]
then
  JAVA=java
fi

echo "INPUT" $param_line

$JAVA -Xmx1536m -XX:-UseLargePages
-jar $tproot/netlogo_model/netlogo-1.0-wrapper.jar
-m $tproot/netlogo_model/JZombiesLogo.nlogo
-outfile $tproot/swift/$instanceDir/counts.csv
-runid 1
-s 150
-trial 1
-i $param_line
-o human_count,zombie_count
```

formatted), given as input using the parameter $-i$. The others parameters for the wrappers are:

- `-outfile`, output file for the outputs parameters;
- `-runid`, execution identify;
- `-trial`, number of trial to execute (for each trial a different random seed is used);
- `-s`, total number of simulation steps to execute;
- `-o`, list of output parameters (notice that the name of the parameters correspond to the name of the variables used in the MASON and NetLogo model implementations).

4.3.2 EMEWS USE CASES

There are many freely accessible libraries relevant to model exploration published in Python, Java, and R. For instance some of which are utilized in use cases 4.3.2 and 4.3.2.

In Python, the Distributed Evolutionary Algorithms in Python (DEAP) toolkit [For+12] is a framework for developing evolutionary algorithms. The scikit-learn package[2] provides a large number of machine learning methods and Theano[3] and Lasagne[4] enable deep learning, capabilities which are useful for surrogate/meta-modeling and active learning [Set12] applications.

On the R side, the EasyABC [F.+13] and ABC [Csi+12] packages provide approximate Bayesian computation (ABC) capabilities that are increasingly being applied to ABM. For machine learning and other statistical applications, R includes packages such as caret [Kuh08], randomForest [LW02], and many others.

In the following sections are described three use cases of EMEWS in combination with Java Repast, MASON, NetLogo simulation and ME libraries in Python and R. These examples are published on a public repository [5].

Simple Workflows with ABM

For a first demonstration ABM use case, the first example presents a Swift/T parallel parameter sweep to explore the parameter space of a model (PSE or ME process). The full Use Case One [`UC1`] project can be found at the tutorial website where, in addition to the Repast Simphony use cases examples, the repository branch **multisim** provides also NetLogo and MASON examples, developed using the ABM software integration described in the Section 4.3.1).

[2] Available at `http://scikit-learn.org`.
[3] Available at `http://deeplearning.net/software/theano/`.
[4] Available at `http://lasagne.readthedocs.io/`.
[5] Available at `https://bitbucket.org/jozik/wintersim2016_adv_tut`.

The example model used here is an adaptation of the JZombies demonstration model distributed with Repast Simphony [CN15]. (The fictional Zombies versus Humans model is intended to illustrate that Swift/T and Repast Simphony are domain agnostic.) The model has two kinds of agents, Zombies and Humans. Zombies chase the Humans, seeking to infect them, while Humans attempt to evade Zombies. When a Zombie is close enough to a Human, that Human is infected and becomes a Zombie. During a typical run all the Humans will eventually become Zombies.

These agents are located in a two dimensional continuous space where each agent has a x and y coordinate expressed as a floating point number (and in a corresponding discrete grid with integer coordinates). Movement is performed in the continuous space and translated into discrete grid coordinates. The grid is used for neighborhood queries (e.g. given a Zombie's location, where are the nearest Humans). The model records the grid coordinate of each agent as well as a count of each agent type (Zombie or Human) at each time step and writes this data to two files. The initial number of Zombies and Humans is specified by model input parameters `zombie_count` and `human_count`, and the distance a Zombie or Human can move at each time step is specified by the parameters `zombie_step_size` and `human_step_size` .

In order for Swift/T to call an external application such as the Zombies model, the application must be wrapped in a leaf function [fun16]. The Zombies model is written in Java which is not easily called via Tcl and thus an `app` function is the best choice for integrating the model into a Swift script. Repast Simphony provides command line compatible functionality, via an InstanceRunner class, for passing a set of parameters to a model and performing a single headless run of the model using those parameters. The command line invocation of Repast Simphony was wrapped in a bash script `repast.sh` that eases command line usage.

Listing 4.4: Zombies model parameter sweep

```
string tproot = getenv("T_PROJECT_ROOT");
app (file out, file err)
_F_repast_C__C_(file repast_sh, file upf, int i,
        string output, string scenario) {
    "bash" repast_sh i output scenario tproot
        @stdout=out @stderr=err;
}
cp_message_center() => {
  file repast_sh =
    input(tproot+"/scripts/repast.sh");
  file   upf      = input(argv("f"));
  string scenario = argv("sd", "scenario.rs");
  string upf_lines[] = file_lines(upf);
  foreach s,i in upf_lines {
    string instance = "instance_%i/" % i+1;
    make_dir(instance) => {
```

```
17        file out <instance+"out.txt">;
18        file err <instance+"err.txt">;
19        file ups <instance+"upf.txt"> = write(s);
20        (out,err) = repast(repast_sh, ups, i+1,
21                           instance_dir, scenario);
22      }
23    }
24  }
```

The Swift app function that calls Repast Simphony is shown in the top of 4.4. Prior to the actual function definition, the environment variable `T_PROJECT_ROOT` is accessed. This variable is used to define the project's top level directory, relative to which other directories (e.g., the directory that contains the Zombies model) are defined. On line 2, the `app` function definition begins. The function returns two files, one for standard output and one for standard error. The arguments are those required to run `repast.sh`: the name of the script, the current run number, the directory where the model run output should be written, and the model's input scenario directory. The body of the function calls the `bash` interpreter passing it the name of the script file to execute and the other function arguments as well as the project root directory. `@stdout=out` and `@stderr=err` redirect `stdout` and `stderr` to the files `out` and `err`. It should be easy to see how any model or application that can be run from the command line and wrapped in a bash script can be called from Swift in this way.

The full script performs a simple parameter sweep using the app function to run the model. The parameters to sweep are defined in a file where each row of the file contains a parameter set for an individual run. The script will read these parameter sets and launch as many parallel runs as possible for a given process configuration, passing each run a parameter set.

The Swift script is shown in 4.4. The script uses two additional functions that have been elided to save space. The first, `cp_message_center`, calls the unix `cp` command to copy a Repast Simphony logging configuration file into the current working directory. The second, `make_dir`, calls the Unix `mkdir` command to create a specified directory. Script execution begins in line 8, calling the `cp_message_center` app function. In the absence of any data flow dependency, Swift statements will execute in parallel whenever possible. However, in this case, the logging file must be in place before a Zombie model run begins. The `=>` symbol enforces the required sequential execution: the code on its left-hand side must complete execution before the code on the right-hand side begins execution.

Lines 11 and 12 parse command line arguments to the Swift script itself. The first of these is the name of the unrolled-parameter-file that contains the parameter sets that will be passed as input to the Zombies model. Each line of the file contains a parameter set, that is, the `random_seed`, `zombie_count`, `human_count`, `zombie_step_size`

and `human_step_size` for a single model run. The parameter set is passed as a single string (e.g. `random_seed = 14344, zombie_count = 10 ...`) to the Zombies model where it is parsed into the individual parameters. The `scenario` argument specifies the name of the Repast Simphony scenario file for the Zombies model. This defaults to "scenario.rs" if the argument is not given.

In line 13 the built-in Swift `file_lines` function is used to read the `upf` file into an array of strings where each line of the file is an element in the array. The `foreach` loop that begins on line 14 executes its loop iterations in parallel. In this way, the number of model runs that can be performed in parallel is limited only by hardware resources.

The variable `s` is set to an array element (that is, a single parameter set represented as a string) while the variable `i` is the index of that array element. Lines 15 and 16 create an `instance` directory into which each model run can write its output. The `=>` symbol is again used to ensure that the directory is created before the actual model run that uses that directory is performed in line 20. Lines 17 and 18 create file objects into which the standard out and standard error streams are redirected by the `repast` function (4.4). The Repast Simphony command line runs allows for the parameter input to be passed in as a file and so in line 19 the parameter string `s` is written to a `upf.txt` file in the instance directory. Lastly, in line 20, the app function, `repast`, that performs the Zombie model run is called with the required arguments.

In script 4.4 is shown how to run multiple instances of the Zombies model in parallel, each with a different set of parameters. The next example builds on this by adding some post-run analysis that explores the effect of simulated step size on the final number of humans. This analysis will be performed in R and executed within the Swift workflow. We present this in two parts. The first describes the changes to the `foreach` loop to gather the output and the second briefly describes how that output is analyzed to determine the "best" parameter combination.

This example assumes an unrolled-parameter-file where we vary `zombie_step_size` and `human_step_size`. For each run of the model, that is, for each combination of parameters, the model records a count of each agent type at each time step in an output file. As before the script will iterate through the "upf" file performing as many runs as possible in parallel. However an additional step that reads each output file and determines the parameter combination or combinations that resulted in the most humans surviving at time step 150 has been added. The relevant parts of the new script are shown in 4.5. Here the `repast` call is now followed by the execution of an R script (lines 2-3, a Swift multiple-line string literal) that retrieves the final number of humans from the output file. The R script reads the CSV file produced by a model run into a data frame, accesses the last row of that data frame, and then the value of the `human_count` column in that row. The `count_humans` string variable holds a

template of the R script where the instance directory (line 3) in which the output file (`counts.csv`) is written can be replaced with an actual instance directory. Line 11 performs this substitution with the directory for the current run. The resulting R code string is evaluated in line 12 using the Swift `R()` function. In this case, the `res` variable in the R script (line 3) contains the number of surviving humans, and the second argument in the `R` call in line 15 returns that value as a string. This string is then placed in the `results` array at the `i`th index.

Listing 4.5: Zombies sweep with human count

```
1  string count_humans = ----
2   last.row <- tail(read.csv("%s/counts.csv"), 1)
3   res <- last.row['human_count']
4  ----;
5
6  //-------------------------------------------------------------
7  ...
8  string results[];
9  foreach s, i in upf_lines {
10   ...
11   (out,err) = repast(repast_sh, ups, i+1,
12                      instance, scenario) => {
13     string code = count_humans % instance_dir;
14     results[i] = R(code, "toString(res)");
15   }
16 }
```

An additional workflow step in which R is used to determine the indices of the maximum values in the `results` array can be seen in the full script. Given that the value in `results[i]` is produced from the parameter combination in `upf_lines[i]`, the index of the maximum value or values in the array is the index of the "best" parameter combination or combinations. Swift code is used to iterate through the array of best indices as determined by R and write the corresponding best parameters to a file.

Workflow control with Python-based external algorithms

Due to the highly non-linear relationship between ABM input parameters and model outputs, as well as feedback loops and emergent behaviors, large-parameter spaces of realistic ABMs cannot generally be explored via brute force methods, such as full-factorial experiments, space-filling sampling techniques, or any other *a priori* determined sampling schemes. This is where adaptive, heuristics-based approaches are useful and this is the focus of the next two use cases.

Listing 4.6: Setting up a DEAP call from Swif

```
1  import EQPy;
```

```
2  (void o) _F_deap_C__C_(int ME_rank, int iters, int pop,
3                         int trials, int seed) {
4    location ME = locationFromRank(ME_rank);
5    algo_params = "%d,%d,%d,\"%s\"" %
6                  (iters, pop, seed, params_csv);
7    EQPy_init_package(ME, "deap_ga") =>
8      EQPy_get(ME) =>
9      EQPy_put(ME, algo_params) =>
10     doDEAP(ME, ME_rank, trials) => {
11     EQPy_stop(ME);
12     o = propagate();
13   }
14 }
```

In [Ozi+15] we describe an inversion of control (IoC) approach enabled by resident Python tasks in Swift/T and simple queue-based interfaces for passing parameters and simulation results, where a metaheuristic method (GA) developed with DEAP [For+12] is used to control a large workflow. The second use case shows how this is done with the EQ/Py extension. The benefit of using external libraries directly is threefold. First, there is no need to port the logic of a model exploration method into Swift/T, thereby removing the (possibly prohibitive) effort overhead and the possibility for translation errors. Second, the latest methods from the many available model exploration toolkits can be easily compared with each other for utility and performance. Third, the external libraries are not aware of their existence within the EMEWS framework, so methods developed without massively parallel computing resources in mind can be nonetheless utilized in such settings.

In this use case is still used the Repast Simphony JZombies demonstration model. For resident tasks, which retain state, the location of a worker is used so that the algorithm state can be repeatedly accessed. The EQ/Py extension provides an interface for interacting with Python-based resident tasks at specific locations. 4.6 listing shows how EQ/Py is used in the current example.The extension was imported at line 1. The `deap` function is defined to take the arguments `py_rank` (a unique rank), `iters` (the number of GA iterations), `trials` (the number of stochastic variations per parameter combination, or individual), `pop` (the number of individuals in the GA population), and `seed` (the random seed to use for the GA). A location `ME` is generated from `ME_rank` in line 4. This location is passed to the `EQPy_init_package` call, along with a package name (`deap_ga`), which loads the Python file named `deap_ga.py` (found by setting the appropriate `PYTHONPATH` environment variable), initializes input and output queues, and starts the `run` function in the `deap_ga.py` file, before returning.

At this point the resident task is available to interact with through the `EQPy_get()` and `EQPy_put()` calls, which get string values from and put string values into the resident task `OUT` and `IN` queues, respectively. The first call to `EQPy_get()` (line 8) is

made in order to push initialization parameters to the resident task via `EQPy_put(ME, algo_params)` (line 9). Then the `doDEAP()` function, to be discussed next, is called and, when it finishes executing, `EQPy_stop()` is called to shut down the resident task.

Listing 4.7 shows the main DEAP workflow loop, a general pattern for interacting with resident tasks. Unlike the `foreach` loop, which parallelizes the contents of its loop, the Swift `for` loop iterates in a sequential fashion, only guided by dataflow considerations. The `for` loop continues until the `EQPy_get()` call receives a message "FINAL", at which point `EQPy_get()` is called again to retrieve the final results and `doDEAP()` exits the loop and returns (lines 12-15). Otherwise, the next set of parameters is obtained by splitting (line 17) the string variable retrieved on line 10. The contents of the `pop` array are individual parameter combinations, also referred to as individuals of a GA population. Each individual is then sent to a summary objective function `obj` which creates `trials` stochastic variations of the individual, evaluates their objective function (the number of Humans remaining, the `count_humans` R code from 4.5) and returns the average value, (not shown here, full script] on tutorial website). Lines 25-29 transform the summary objective results for each individual into a string representation that can be evaluated within the Python resident task, and this value is sent to it via `EQPy_put()` (line 30).

The EQ/Py extension makes two functions, `IN_get` and `OUT_put`, available for the Python resident task and these can be used to pass candidate parameters to and get results from any Swift/T workflow. These functions are the complements to the `EQPy_get()` and `EQPy_put()` functions on the Swift/T side.

The DEAP framework provides flexibility in defining custom components for its GA algorithms and is taking advantage of this by overriding the `map()` function used to pass candidate parameters for evaluation to our custom evaluator with `toolbox.register("map", queue_map)`. The `queue_map` function executes calls to `OUT_put` and `IN_get`. In this way the Python resident task is unaware of being a component in an EMEWS workflow. The full Python resident task code (deap_ga.py) along with the full DEAP use case can be found in the Use Case Two (UC2) project, also for MASON and NetLogo in the branch *multisim*.

Calling a distributed MPI-based Model

In this use case, we will show how to integrate a multi-process distributed native code model written in C++ into a Swift/T workflow. The model is a variant of the Java Zombies model, written in C++ and using MPI and the Repast HPC toolkit [CN12] to distribute the model across multiple processes. The complete two dimensional continuous space and grid span processes and each individual process holds some subsection of the continuous space and grid. The Zombies and Humans behave as

Listing 4.7: The main DEAP workflow loop

```
(void v) _F_doDEAP_C__C_(location ME, int ME_rank,
                int trials) {
  string param_names =
    "zombie_step_size,human_step_size," +
    "zombie_count,human_count";
  for (boolean b = true, // Loop variables
       int i = 1;
       b;                // Loop condition
       b=c, i = i + 1) { // Loop updates
    string params =  EQPy_get(ME);
    boolean c;
    if (params == "FINAL") {
      string finals =  EQPy_get(ME);
      printf("Results: %s", finals) =>
        v = make_void() => c = false;
    } else {
      string pop[] = split(params, ";");
      float fitnesses[];
      foreach p, j in pop {
        fitnesses[j] =
          obj(param_names, p, trials,
              "%i_%i_%i" % (ME_rank,i,j),
              "deap_ga");
      }
      string rs[];
      foreach fitness, k in fitnesses {
        rs[k] = fromfloat(fitness);
      }
      string res = join(rs, ",");
      EQPy_put(ME, res) => c = true;
    }
  }
}
```

before but may cross process boundaries into another subsection of the continuous space and grid as they move about the complete space. The HPC Zombies source, a Makefile, the various files required to integrate it with Swift/T, and the Swift scripts can be found in the Use Case Three (UC3) project. There, the HPC Zombie model runs are driven by an active learning [Set12] algorithm using EQ/R, the R counter-part to EQ/Py described above 4.3.2.

In contrast to the previous two examples the MPI-based HPC Zombies model is compiled as a shared library that exposes a Swift/T Tcl interface [Woz+15]. Swift/T runs on Tcl and thus wrapping the library in Tcl provides tighter integration than an app function, but is also necessary in the case of multi-process distributed models that use MPI. Such models when run as standalone applications initialize an MPI Communicator of the correct size within which the model can be distributed and run. Since the HPC Zombies model uses MPI, as do all Repast HPC based models, it must be treated as an MPI library and passed an MPI communicator of the correct size when run from Swift/T 3.2.3.

Listing 4.8: Zombies model Swift interface

```
1 @par @dispatch=WORKER (string z) zombies_model_run(string config,
      string params)
2       "zombies_model" "0.0" "zombies_model_tcl";
```

The first step in integrating the HPC Zombies model with Swift/T is to compile it as library, converting *main()* into a function that runs the model. The next step is to make that function callable from Tcl via a SWIG created binding. SWIG[Bea96] is a software tool that generates the 'glue code' required for some target language, such as Tcl, to call C or C++ code. The SWIG tool processes an interface file and produces the 'glue-code' binding as a source code file. In this case, the C++ code we want to call from Tcl and ultimately from Swift is the Zombies model function: `std::string zombies_model_run(MPI_Comm comm, const std::string& config, const std::string& parameters)`. The function takes the MPI communicator in which the model runs, the filename of a Repast HPC config file, and the parameters for the current run. When called, it starts a model run using these arguments.

The SWIG interface file is run through SWIG and the resulting source code is compiled with the HPC Zombies model library code. The result is a `zombie_model_run` function that is callable from Tcl. The Makefile target, *./src/zombies_model_wrapper.cpp* in the Makefile template is an example of this process.

The next step is to create the Swift bindings for the library function. The Swift bindings define how the `zombie_model_run` function will be called from Swift. The Swift code is shown in 4.8. The function is annotated with `@par` 3.2.4 allowing it to be called as a parallel function. The `@dispatch=WORKER` 3.2.4 directs the function to

run on a worker node. The function itself returns a string that contains the number of humans and zombies at each time step. For arguments, the function takes the config file file name and a string containing the parameters for the run. The model will parse the individual parameters from this *params* string. The final 3 parts of the function definition are the Tcl package name, the required package version, and the Tcl function to call in the package. With this code included in the Swift script (either directly or through an import), it is possible to execute the HPC Zombies model with a call like:

```
string output = @par=4 zombies_model_run(config_file, params);
```

The Swift binding references a *zombies_model* Tcl package and a `zombies_model_tcl` function in that package. The final step in integrating the HPC Zombies model with Swift is to create this package and the function. A Tcl package is defined by its `pkgIndex.tcl` file that specifies the libraries that need to be loaded as part of the package and the Tcl code that is in the package. Tcl has a built-in function `::pkg::create` that can be used to create a `pkgIndex.tcl` given a package name, version, the name of the library to load, and the Tcl code file name. The HPC Zombies example uses some simple Tcl code to call this function as part of a Makefile to create the package (c.f. the `zombies_tcl_lib` Makefile target and the `make-package.tcl` script).

Listing 4.9: Swift/T Tcl interface functions for HPC Zombies

```
1  proc zombies_model { outs ins args } {
2    rule $ins \
3        "zombies_model_body $outs {*}$ins" \
4        {*}$args type $turbine::WORK
5  }
6
7  proc zombies_model_body { z cfg prms } {
8    set c [ retrieve_string $cfg ]
9    set p [ retrieve_string $prms ]
10   # Look up MPI information
11   set comm [ turbine::c::task_comm ]
12   set rank [ adlb::rank $comm]
13   # Run the Zombies model
14   set z_value \
15       [ zombies_model_run $comm $c $p ]
16   if { $rank == 0 } {
17     store_string $z $z_value
18   }
19 }
```

The code in the `zombies_model` Tcl package that the Swift function calls is shown in 4.9. For parallel tasks, Swift/T currently requires two Tcl procedures, a dataflow

interface (line 1) and a body (line 7) that calls the `zombies_model_run` function. The `rule` command (line 2) registers dataflow execution: when all input IDs `ins` are available, `zombies_model_body` will be released to the load balancer and executed on an available subset of workers. The `args` are Swift-specific task settings, including the `@par` parallel settings, and are not accessed by the user. The body function retrieves the input parameter values from Swift by applying the provided `retrieve_string` function on the IDs, obtaining the configuration file name and the sample model parameters. The MPI subcommunicator for the parallel task and the current MPI rank in that communicator are accessed using functions by Swift (lines 11-12). The communicator `comm` will be of the size specified by the `@par` annotation. The HPC Zombies library interface is called (line 15) and returns a string containing the model output that is stored in `z_value`. Only one process need store the output `z` in Swift memory; we use rank 0 (lines 16-17). (This interface / body pattern can easily be adapted for any MPI library, adjusting for any differences in the wrapped library function arguments and additional input/output parameters.) With the various bindings having been created, the HPC Zombies model can now be called from a Swift script.

4.3.3 Tutorial Site for EMEWS

A detailed tutorial[6] has been produced to present the Extreme-scale Model Exploration With Swift (EMEWS) framework for combining existing capabilities for model exploration approaches (e.g., model calibration, metaheuristics, data assimilation) and simulations (or any "black box" application code) with the Swift/T parallel scripting language to run scientific workflows on a variety of computing resources, from desktop to academic clusters to Top 500 level supercomputers. The tutorial presents the use-cases described in Section 4.3.2. With more details, the tutorial aims to describe through examples the following main elements of the EMEWS framework:

- How external code can be incorporated with minimal modifications;
- How the EMEWS Queues (EQ/Xs) are used to communicate between model exploration code and Swift;
- How EMEWS enables the scaling of simulation and black box model exploration to large computing resources;
- How modularized, multi-language code can be effectively tested and integrated within the EMEWS framework.

Tutorial Window. The tutorial has been designed with an innovative approach which is able to provide within a single web IDE the description, the code and the project structure. The tutorial window, depicted in Figure 4.5, is composed of five panes. The tutorial text appears in pane 1. Within the tutorial text you will see

[6] Available at `http://www.mcs.anl.gov/~emews/tutorial/`.

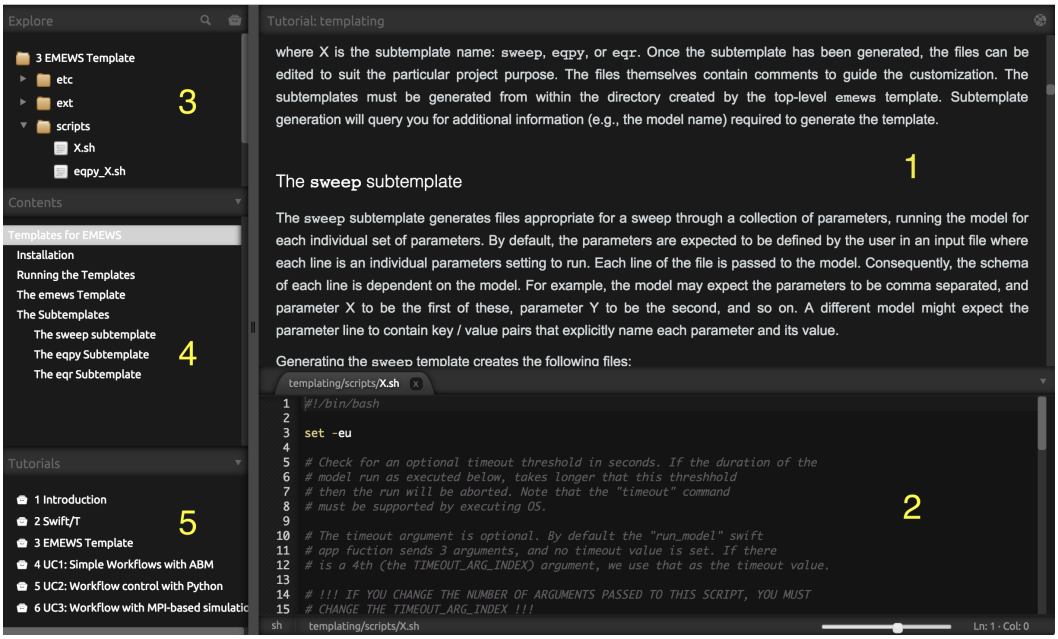

Fig. 4.5.: EMEWS Tutorial Site – http://www.mcs.anl.gov/~emews/tutorial/

hyperlinks to source code that is being explained. These links are of two types. Both types will display the content of the relevant source code file in pane 2. The second type will also scroll the source code and highlight lines of interest. For example, clicking on the link: *swiftrun.swift* will open the swiftrun.swift file in pane 2, and clicking on this one will scroll and highlight lines $13 - -15$ in that file. Other links refer to sections in other tutorials, or bibliographic references. These will appear as a modal browser window, displaying the relevant text. Pane 3 displays the source files and project structure for the tutorial currently displayed in pane 1. Double-clicking a folder or single-clicking the triangle symbol to the left of the folder will reveal its contents. Pane 4 displays a table of contents for the current tutorial. Single click on an entry to scroll the tutorial text in pane 1 to that section. Pane 5 displays the list of tutorials. Double click on an entry to open that tutorial.

5 Scalable Web Scientific Visualization

> *The greatest value of a picture is when it forces us to notice what we never expected to see.*
>
> — **John Tukey**
> American Mathematician

5.1 Introduction

As discussed in Section 1.1, the Computational Science field comprises research interests on scientific data visualizations. Despite the fact that scientific data visualization many times is considered as a subfield of computer graphics, it is an interdisciplinary research branch which aims to visualize data, using a computational approach and/or computer science tools, in a way that helps the reader grasp the nature of (potentially) large datasets.

This Chapter describes an extensible and pluggable architecture for the visualization of data on the Web, tailored particularly for Open Data. Open data is data freely available to everyone, without restrictions from copyright, patents or other mechanisms of control. The advantage of the Open data is that they are freely accessible on the Web in machine readable format. The architecture aims to introduce a scalable way to easily visualize Open data in classical HTML pages.

The most important feature of the proposed architecture is that it moves the computation client side: data gahtering, data filtering and the rendering of the visualization are made client side, and not server side, as in other architecture. This ensure the *scalability* in terms of number of concurrent visualizations, *data trustiness* and *privacy* (because the data are dynamically loaded client side, without any server interactions). These design features are the fundamentals of a novel paradigm of Distributed Computing named *Edge-Centric Computing* or *Fog Compunting*.

5.1.1 Edge-centric Computing

Edge-centric Computing (EcC) is a novel Distributed Computing paradigm presented in [Lop+15]. This paradigm is based on an observation that in Computing as in in many aspects of human activity there has been a continuous struggle between the forces of centralization and decentralization.

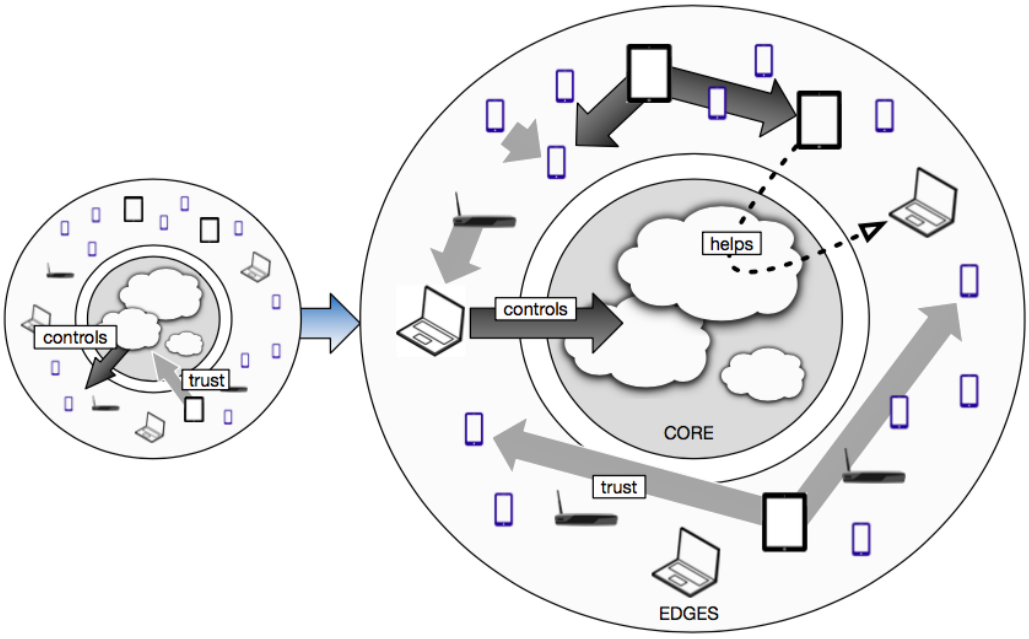

Fig. 5.1.: Centralized cloud model (left) versus Edge-centric Computing (right).

In EcC the most of the computation is moved client side, this affects the cloud paradigm that becomes a support infrastructure to the computing. These kind of decentralization allows to easily ensure the data trustiness and scalability on million of users, moreover, allows to develop novel kind of human-centered applications.

The Figure 5.1 depicts the difference between the classical Cloud Computing and EcC paradigms. The EcC architecture is composed of a core and some edges. The core is represented by smaller web servers and content distribution networks, while the edges consist of individual human-controlled devices such as desktop PCs, tablets, smart phones, and nano data centers. As shown in the picture the core (that may be a cloud infrastructure or service provider) is responsible for a minimum part of the total computation and information sharing, the edges in the architecture should operate without of core interference in order to ensure trust and massive number of operations. The EcC architecture may be seen as an attractive and complete definition of other architectures, such as *Content Delivery Networks*(CDN), *Peer-to-Peer*(P2P), *Decentralized Cloud Architecture* and *Fog Computing*.

Problem of centralized computing. The EcC has been designed to face three main problems. The first fundamental problem is the loss of privacy by releasing personal and social data to centralized services. A second fundamental problem is the delegation of the total control to cloud or service infrastructures. These can not easily ensure any kind of trust both on the data and on the operations performed on it. For instance, delegating the visualization of some data to a cloud service, requires

that the user must trust the service provider, that may change the data or use them for other purposes. The final problem is the scalability, the centralization does not allow to design scalable application on a massive number of computations. Moreover it misses the opportunity of exploiting the enormous amount of computation of modern personal devices (e.g. Desktops, Laptops, Smart phones, etc..).

5.1.2 Data Visualization process

A "visualization is worth then thousands of words", and very often they are used to convey information in a pleasant, understandable and intuitive way. Hence, our vision is to provide a tool to help and engage users in the creation of interactive visualizations of data. In this way, the data can be represented through visualizations, such as, line charts, bar charts, maps and so on, that summarizes and emphasizes the information hidden behind the data.

The data visualization process may be described by three main steps:

1. *Formulate the question*, the first step of the data visualization process is to be clear on the question to be answered. The answer to the question will be given by a visualization of data. However, the question may be refined, often, after a good understanding, gathering and filtering of data.
2. *Data gathering*, once the question has been identified, it is necessary to identify and collect data related to the formulated question. This can be done by using own data or using data available elsewhere, as in the case of Open Data. These kind of data are accessible by service providers. Table 5.1 shows 6 popular open data software providers. Many of them allow to access the data in machine readable formats using Web API.
3. *Apply a visual representation*, the last step correspond to identify the appropriate way to visualize the data. The challenges facing visualization researchers often involve finding innovative graphic and interactive techniques to represent the complexity of information structures. In general, the visual representation of data is the crucial point in data visualization process, and may be a complex task. It is important to have some tools able to help users to chose to right way of visualizing the data.

The architecture described in this chapter aims to help in the final two steps of the Data Visualization process, providing automatic mechanisms to gather the data, software libraries and Web tools to apply a visual representation to the data.

Name	License	Gathering data	Link
CKAN	GPL v3.0	Web Interface and Web API	www.ckan.org
OGDI	Microsoft Public License	Web Interface and Web API	https://github.com/openlab/OGDI-DataLab
Libre	GNU Public v3.o	Web API	https://github.com/commonwealth-of-puerto-rico/libre
DKAN	Free for non-profit	Web Interface and Web API	https://www.drupal.org/project/dkan
OpenDataSoft	Free for non-profit	Web Interface and Web API	https://www.opendatasoft.com/
OGPL	Open source	Web Interface and Web API	http://opengovplatform.org/

Tab. 5.1.: Popular Open Data softwares providers.

5.1.3 Open Data in a Nutshell

An adequate single definition of Open Data is the Open Definition published by Open Knowledge. In which Open mean:

> "Knowledge is open if anyone is free to access, use, modify, and share it — subject, at most, to measures that preserve provenance and openness."

An Open Data [Ope16a] covers two different aspects usage:

1. it is published under an open license, hence is legally open;
2. it is free to access for everybody, without using property format or restriction, and should be searchable in machine readable format. That means that is technically open;
3. in other words, it it can be freely used, modified, and shared by anyone.

The architecture, we are going to present, is based on the fundamentals of Open Data [Ope16a], exploiting the standardization induced by their definition, in order to provide an easily/scalable mechanism to visualize data on Web pages. Many are the advantages of having an architecture able to easily exploit Open Data. For instance, it can be used to improve transparency of governments and public administration, that could provide to citizen information about their activities and tools for understanding them. Making public sector data open is crucial for different reasons:

- allow citizens to understand better the government and public activities;
- enable the citizens to be better involved in the public choices;
- represent a starting point to develop new services for citizens.

The Open Data is a structured data and is developed to be easily processed by machines. Different types of machine-readable data are considered according to the level of interoperability that exploits, the Table 5.2 shows several data format classified according to their level of interoperability. JSON and CSV formats are the most common adopted by the Web API of Open Data platforms that are typically available through RESTful Web API, establishing an automated way to gather data. The communities has produced, during the last decade, many Open Data platforms such as CKAN [Ope16d] and OpenDataSoft [Ope16b] that support these kinds of interactions.

Following the EcC paradigm, in this dissertation, is proposed *DatalEt-Ecosystem Provider* (DEEP), an open source, extensible and pluggable architecture for the visualization of data which enables to **gather** (dynamic data), **query** and **visualize** data in classical HTML pages for a massive amount of concurrent visualizations (briefly described in [Mal+16]). The most important design feature concerns data manipulation that is made on the client side, and not on the server side, as in other architectures. This ensures the scalability in terms of number of concurrent

Machine Readable	Geodata Machine Readable	Less Readable	Closed
JSON	Shapefile	PDF	Images (PNG, JPG,etc.)
XML	GeoJSON	Text	Charts
RDF	GML	HTML	
CSV	KML	Excel	
TSV	WKT	Word	
ODF			

Tab. 5.2.: Open Data formats and machines interoperability.

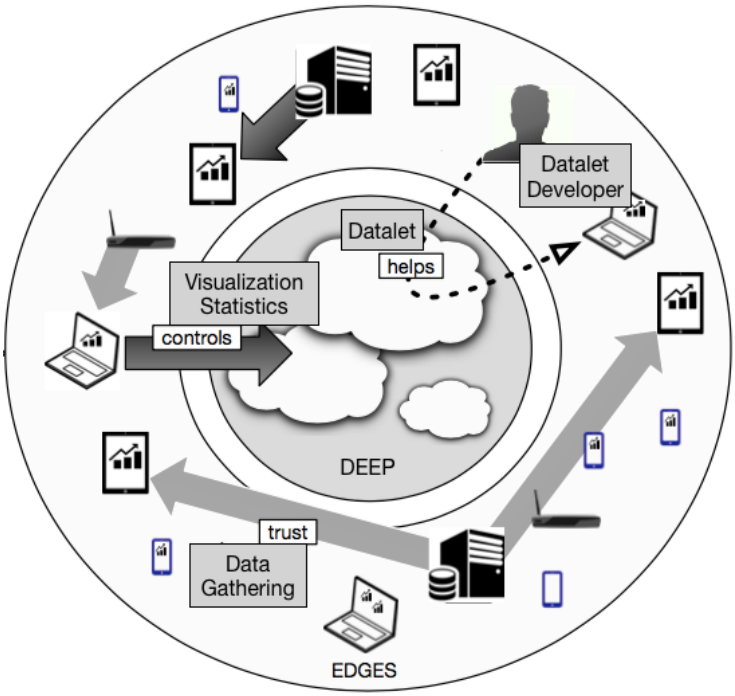

Fig. 5.2.: Edge-centric Computing Architecture of *DatalEt-Ecosystem Provider*

visualizations, and dependability of the data and privacy (because the data are dynamically loaded client side, without any server interactions).

5.2 DatalEt-Ecosystem Provider

DEEP is an open source, extensible and pluggable architecture providing visualizations of Open Data in a distributed web computing fashion. The code is published on a public repository[1], introduced in [Mal+16]. The visualizations are provided by an object that is an off-the-shelf, reusable web-component able to load, query, filter, and visualize any dataset content, that includes all the information to be rendered by a standard web page. This kind of object, named *Datalet*, is based on the novel web standard of web-component [Wor16] and exploits different JavaScript visualization libraries like Highcharts [Hig16] or Leaflet[Lea16]. Close to our architecture is

[1]DEEP code repository: https://github.com/routetopa/deep

Google Charts [Goo16a], however it does not support different visualization libraries and dynamic data (it supports only the Chart Tools Datasource protocol).

The DEEP architecture is compliant with the EcC architecture (see the grey functional blocks in Fig. 5.2): Datalets guarantee the provenance of data (see [`Data Gathering`] in Fig.5.2), showing the link to the original dataset used to create the visualization. In this way, any user can determine whether information is trusted, and whether data have been manipulated; Datalets ensure the scalability in terms of visualizations (see [`Datalet`] in Fig. 5.1). The computation is made client side, and does not experience bottlenecks due to overloading of the core. The core may provide other services to the edges; for instance: reports, statistics, forecasting for certain data exploiting the Datalets usage (see [`Visualization Statistics`] in Fig. 5.1).

DEEP may be seen as a cloud provider of visualization for some Open Data. An increasing interest about public discussion around data visualization is witnessed by several sites of data-journalism. Among the other, the Economist's Graphic detail blog[2] regularly posts data visualizations (i.e., charts, maps and info-graphics) along with the articles. On-line users contribute by actively commenting and sharing their interpretations, observations, and hypothesis of the visualizations. A recent study of the users' comments [Hul+15] observed that 42.2% of them are focused on the available content, such as, the visualizations, data and articles. On this blog, the comments are text-based without the opportunity for users to create and share alternative visualizations of the same dataset or other datasets. The aim, through the DEEP architecture is to enable the social sharing and the collaboration around data visualizations. The idea is that users can reuse and share data visualizations to explain their interpretation of data and support their argumentation during the discussion.

Visualization of Open Data is an essential tool to understand and interpret the dataset content, foster the collaboration and engage users in discussions. In their comments, users can support the same argumentation or answer back by creating and commenting with other datalets. In particular, a user can reply by commenting with another visualization of the same dataset or a Datalet of a different dataset. In this way, social interactions evolve around visualizations through the creation and posting of datalets to support argumentations. Datalets guarantee the provenance of data, showing the link to the original dataset used to create the visualization. In this way, any user can determine whether information is trusted, which is the data source and how to trust it, whether data have been manipulated.

[2]Economist's Graphic Detail blog is accessible to `http://www.economist.com/blogs/graphicdetail`

5.2.1 DEEP Background

Web Components Standard

A Datalet may be seen as reusable web widgets, DEEP exploits the web components standard, currently being produced by Google engineers compliant with W3C specification [Wor16]. The goal is to use component-based software engineering to develop a bundle of web widgets that can be used whenever needed, without having to rewrite the common fragments shared by several pages of the platform. The components model allows for encapsulation and interoperability of individual HTML element.

The web component technologies enable to create your own HTML elements and the support for these components is present in some WebKit-based browsers like Google Chrome, Opera and in Mozilla Firefox (but it requires a manual configuration change). Microsoft's Internet Explorer has not implemented any Web Components specifications yet. Backward compatibility with older browsers is implemented using JavaScript-based polyfills [Web16] (a library you can import in the web page that implements the web component specification).

Web components specifications consist of four elements, which can be used also separately:

1. *Custom Elements*[3], define the method to create new types of DOM elements in a document. The element definition consists of custom element type, local name, namespace, custom element prototype and lifecycle callbacks.
2. *HTML Imports*[4], is a way to include and reuse HTML documents, typically web component definitions, in other HTML document called master document. The imported definitions are linked as external resources to master document.
3. *Templates*[5], describe a method to declare inert DOM subtree in HTML and manipulate them to instantiate document fragment with identical content.
4. *Shadow DOM*[6], defines a method of establishing and maintaining functional boundaries between DOM tree and how these trees interact with each other within document enabling better functional encapsulation within the DOM.

The major support libraries that implements the web components are:

- *X-Tags by Mozilla* [X-T16]: allows developers to easily create elements to encapsulate common operations.

[3] http://w3c.github.io/webcomponents/spec/custom/
[4] http://w3c.github.io/webcomponents/spec/imports/
[5] https://html.spec.whatwg.org/multipage/scripting.html#the-template-element
[6] http://w3c.github.io/webcomponents/spec/shadow/

- *Polymer by Google*[Goo16b]: allows developers to create interoperable custom elements that extend HTML. Polymer allows to enable browsers that does not support the standard to use web components.
- *Bosonic*[Bos16]: similar to Polymer library.

According to the specifics of the web components standard, DEEP architecture uses Polymer. Polymer is the library that supports the major number of requirements as template, web components, material components, data binding, filters, events handling, touch and gestures, and AJAX/REST.

JavaScript Chart Libraries

The idea is to provide to the user an easily developing to include a data visualization in web pages. However, this work does not aim to develop new JavaScript visualization libraries, but reusing existing libraries and providing an innovative transparent way to use it. Visualizations JavaScripts libraries are defined by the following features:

1. *Cross Browser Compatibility*: a visualization library must be compatible with all browsers or modern browser taking in account the target audience;
2. *Cross Device Compatibility*: a visualization library must be responsive on both desktop and hand-held devices;
3. *Input Data Format*: supports to different input data format, such as JSON (Javascript Object Notation);
4. *Customizability*: possibility to customize the visualizations, for instance, configuring legends, attaching events (e.g., hover, click);
5. *A set of available charts*: number of visualizations offered by the library;
6. *Performance*: many factors must be consider, such as, size of library, memory usage, garbage collection and number of browser repaint cycles;
7. *Exporting*: support to export visualizations in different formats, such as PDF or images (e.g. JPEG, PNG, SVG stc.);
8. *License*: the license of the library is a crucial point, due to this work is an open source project.

Many popular JavaScript for visualizations are:

1. *Dygraphs*[Dyg16] is fast, flexible open source library that allows users to explore and interpret dense dataset. It is customizable, it works in all major browsers with zoom and mobile and tablet devices support.
2. *Leaflet*[Lea16] is a library for maps based on OpenStreetMap. It provides an interactive geo-localized data visualization in an HTML5/CSS3 wrapper. It is extensible with a wide range of plugins with specific functionality such as animated markers, masks and heatmaps.

3. *jqPlot*[A V16] is a jQuery (a well-know general purpose JavaScript library) plugin for line and point charts. It has a few additional features such as the ability to generate trend lines automatically. It is extensible by plugins. There are plenty of hooks into core code allowing for custom event handlers, creation of new plot type and more. It has been tested on IE7, IE8, Firefox, Safari and Opera.
4. *D3.js*[D3.16] is a very powerful library that uses HTML, SVG and CSS to render different set of diagram and chart from a variety data source. It is, more than most, capable to provide some seriously advanced visualization with complex features and includes some advanced user interaction support.
5. *Highcharts*[Hig16] is a charting library with a very huge range of chart options available. It uses SVG for modern browsers and VML in Internet Explorer. All charts have a really attractive looks and animation. It is well documented and used by a tens of thousands of developers (great community support) and it is also very simple to use.
6. *Google Charts*[Goo16a] is a set of powerful chart tools that provide a way to visualize data on a website. The charts gallery provides a large number of ready-to-use chart type, from simple line chart to complex hierarchical tree map. It is used by embedding simple Javascript in the webpage. Charts are highly interactive and expose events that the developer could use to create complex dashboard or advanced user interaction mechanism. All charts are rendered using HTML5/SVG technology to provide cross-browser compatibility. The purposes of this library are similar to the DEEP ones, but it is not open source, it does not ensure mechanism to gathering data, and the computation in its initial implementation is made on the severs side.
7. *Crossfilter*[Cro16] is a library for exploring large multivariate datasets in the browser. Crossfilter is specialized to support extremely fast (< 30ms) interaction with coordinates view with datasets containing a million or more records. This library displays data and make user able to restrict the range of data and see other linked charts react.
8. *Polymaps*[A J16] is a free library for making dynamic, interactive vector-tiled maps and images in modern web browser using SVG. It allows to define the design of the data by CSS rules and provides the display of multi-zoom datasets over maps, supports a variety of visual presentation for tiled vector data. It is ideal for showing information from country level on down to state, cities and individual streets because of it can load data at full range of scales.
9. *Flot*[Att16] is an attractive charting library for JQuery with focus on simple usage, attractive looks and interactive features. It is extensible by plugin and provides a basic support for lines, points, filled areas, bars and other type of charts.
10. *Raphael*[Rap16] is a very simple and small library that provides support for a wide range of data visualization options, which are rendered using SVG.

11. *jQuery Visualize*[jQu16] is written by the team behind jQuery's ThemeRoller and jQuery UI websites, jQuery Visualize Plugin is an open source charting plugin for jQuery that uses HTML Canvas to draw a number of different chart types. One of the key features of this plugin is its focus on achieving ARIA support, making it friendly to screen-readers.
12. *OpenLayers*[Ope16e] is probably the most robust of mapping libraries. The documentation is not great and the learning curve is steep, but for certain tasks nothing else can compete.
13. *Dimple*[Dim16] is a library to aid in the creation of standard business visualizations based on D3.js. It makes easy for anyone, analyst or not, to develop stunning, three-dimensional graphics without any real JavaScript training. The dimple API tested against Firefox, Chrome, Safari and IE9. Its browsers support is largely inherited from D3.js so using it on IE8 and earlier will be difficult/impossible.

5.2.2 DEEP Architecture

The core of the DEEP architecture is the concept of Datalet. Datalets move the data visualizing work-flow to the edges of the network providing scalability, data trustiness and privacy. Furthermore, it enables all users (not only computer or data scientists) to include visualizations in their Web pages, while the system automatically ensures the scalability in terms of efficiency of visualization. That means that the users are sure about the performances of their pages, because the computation is made client side, and no server overload is possible. Datalets ensure also that the visualizations code is each time automatically updated, see Figure 5.1 [Datalet].

DEEP Datalets

A Datalet may be seen as reusable web widgets; the DEEP exploits the *web components* standard compliant with *W3C* specification [Wor16]. According to this standard, in this architecture we exploit Polymer [Goo16b] (see section 5.2.1), a library developed by Google engineers that supports the major number of requirements as template, web components, material components, data binding, filters, events handling, touch and gestures, and `AJAX/RESTful` support.

The idea is to provide to the user an easy development tool to include a data visualization in web pages. However, this work does not aim to develop new JavaScript visualization libraries, but on reusing of existing libraries, *providing an innovative transparent way to use them*.

Datalets have been designed to process any dataset as input, provided that the Data Provider enables to access the data using Web API in a machine readable format

(such as JSON or CSV). In the current implementation, datalets are tested on data from CKAN installations, and from a commercial provider such as OpenDataSoft.

The datalet design follows the Object-Oriented paradigm (OOP), according to it all the datasets are build, brick by brick, inheriting from other bricks. Each brick is a web component that exports some functionalities. The building of a datalet is made on top of four layers, see Figure 5.3: *Architectural* layer; *Library* layer; *Visualization-depended* layer; the *Datalet* layer.

The *Architectural* layer provides common behaviors for all datalets, that are:

- *BaseDataletBehavior,* which defines the mandatory attributes that all datalet must have:
  - `data_url`: a string URL used to get the data from the open data provider (e.g. the CKAN API).
  - `fields`: a JSON array of strings representing user selected dataset fields.
  - `data`: an array data structure that store the data retrieved from open data provider.
- *WorkcycleBehavior,* which implements the work-cycle of each datalets, composed by the following steps:
  - `GET`: it is responsible for data retrieving from an open data platform.
  - `SELECT`: it is responsible for extracting a query related subset of information from the entire dataset. A multidimensional array will be made available for the transformation step.
  - `FILTER`: it enable to select a certain number of row by applying one of the following relational operators (<,>, =, <=, >=, etc.).
  - `TRANSFORM`: it is responsible for aggregating and organizing the data in order to obtain a coherent data representation.
  - `PRESENT`: it provides a specific visualization for the selected data.
- *AjaxJsonAlasqlBehavior,* which uses AJAX to request, select, filter data.

The *Library* layer includes all behavior referred to a particular visualization library (e.g. Highcharts [Hig16]).

The *Visualization-depended* layer encloses the behavior refereed to a specific visualization (e.g. Bar chart, table chart, etc.) developed using a particular library.

The *Datalet* layer is the real implementation of the web component datalet, and is developed by some of the hierarchy behavior.

Clearly, the datalet building process takes advantage of this hierarchy; different datalets may share low level behaviors facilitating code reuse and error fix to datalet developers.

Hence, each datalet is a web-component that implements a reusable lifecycle (see Fig. 5.4) with all required behaviors to load the datalet source code, the dataset

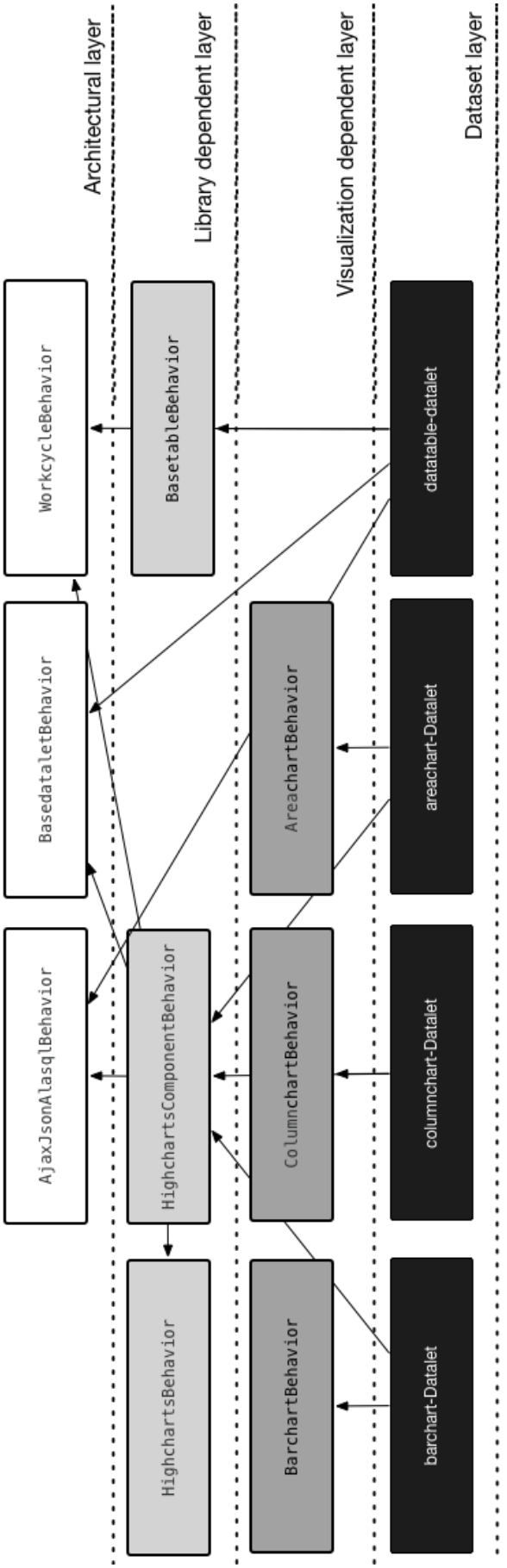

Fig. 5.3.: Datalets Object-Oriented paradigm embedding. The four layer of DEEP datalets architecture.

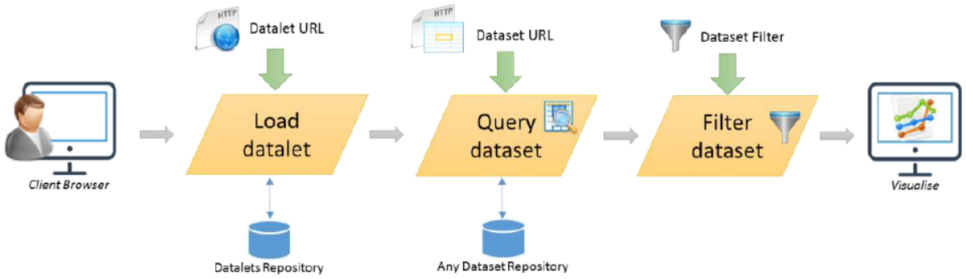

Fig. 5.4.: Datalet lifecycle..

content, filter and group the data to finally render the visualization in the web-page. Fig. 5.4 shows the whole process that evolves from left to the right; on top, there are the inputs, below repositories. The datalet is included in the web-page source code. When the browser loads the web-page, it processes the datalet source code that is self-consistent with all parameters (i.e., datalet url, dataset URL, filters and other configuration parameters). When the web page loads for the first time, the browser contacts the DEEP architecture (see Figure 5.6) to download the datalet source code from the datalets repository. The datalet source code reads the URL of the input dataset, thus, it downloads the data in real-time and performs further processing (such as data filtering, grouping and so on).

DEEP Web Service

The DEEP component is a simple RESTful service, providing the list of available datalets (i.e., a lookup service) and the mapping among the visualization names and their relevant URL within a datalet repository. A datalet takes as input: a dataset URL, a query to be performed on the data, and -optionally- a filter and/or some additional configuration parameters. Currently, on `http://deep.routetopa.eu` several kind of datalets (bar chart, map, treemap and others) are available, and many more are planned. All datalets are published on a public repository[7].

Both the DEEP and the Datalet repository have been designed to be extensible: they can log all the visualization requests and, as planned future work, they could also provide aggregated statistics on both users preferences and on data and their visualizations. For instance, the most popular Datalet visualizations, most used datasets, most popular visualizations for a particular dataset, most visualized field for a particular dataset, and so on (see Figure 5.2 [Visualization Statistics]). Such information will be also used to provide useful suggestions to inexpert users, both on data selection, their filtering and their visualization.

From the user point of view, datalets are interactive, real-time and dynamic visualizations for Open Data. Datalets are interactive (they are not static pictures), allowing

[7]DEEP Datalets code repository: `https://github.com/routetopa/deep-components`

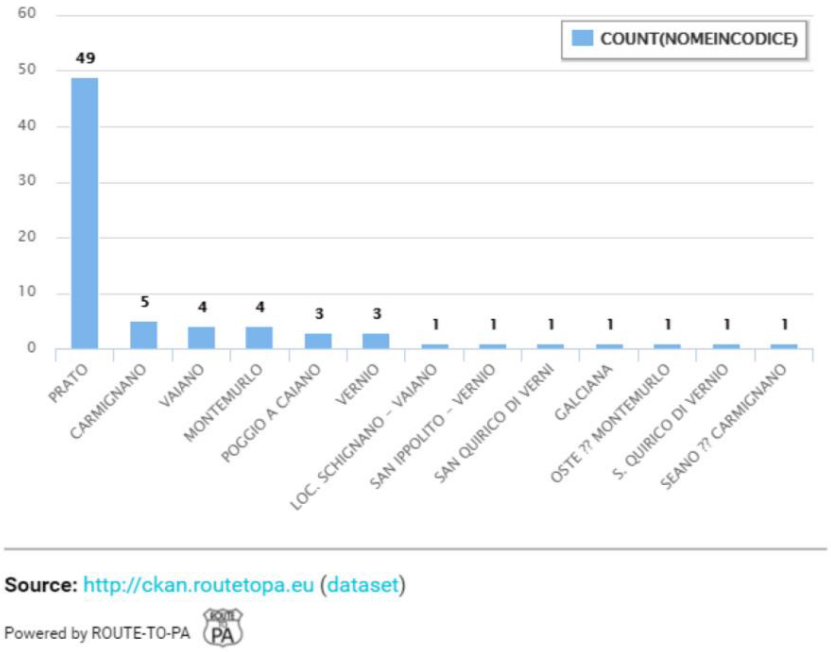

Fig. 5.5.: An Example of datalet..

users to zoom in and out in the data, move the mouse on the visualization items and have additional information. Figure 5.5 is an example of datalet: a bar chart that shows the number of Wi-Fi antennas for each subarea in the city of Prato (Italy).

5.2.3 Datalets in HTML page

Datalets are reusable and portable in any web-page. For example, a data journalist can write an article exploiting open data from government portals and publish the article content along with visualizations to explain and show, for instance, trends. They can be published along with the blog posts.

The Figure 5.6 depicts the work-cycle to include a Datalet in a Web page, that is composed by:

- Client web page that exploits DEEP-Client functionality;
- DEEP (DatalEts-Ecosystem Provider);
- Datalet.

First, (1) the Client page sends a request to DEEP for a specific datalet. Then, (2) the DEEP responds with the information needed to inject the datalet into the page. Finally, (3) the Client retrieves the Datalet from the DEEP repository and includes it into the page. When the datalet is injected in the page the behavior of the datalet is executed.

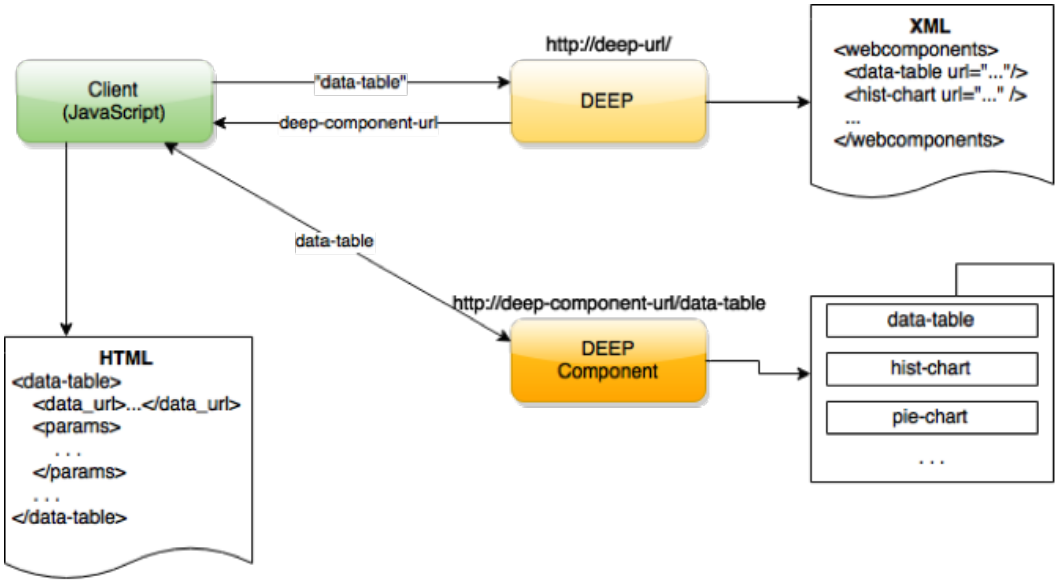

Fig. 5.6.: Datalets usage in HTML page.

Listing 5.1 shows the code to include a datalet in a classical web page. Lines 2, 3, 4 load the JavaScript libraries, in particular the `deepClient.js`[8] library enables the page to dynamically download the Datalet code from the DEEP repository. This request is made using jQuery (lines 6–14), specifying the needed parameters. The DEEP answer with the corresponding datalet code, which is then injected into the page.

Listing 5.1: Datalet in HTML page

```
1  <html> <head>
2  <script type="text/javascript" src="js/jquery-1.11.2.min.js"></script>
3  <script type="text/javascript" src="js/webcomponents.js"></script>
4  <script type="text/javascript" src="js/deepClient.js"></script>
5  <script type="text/javascript">
6  jQuery(document).ready(function($) {
7    var datalet_params ={
8      component: "DATALET_NAME",
9      params : {  data-url: "DATA_URL", layout-param-1: "LAYOUT-VALUE"}
10     fields      : Array("FIELD1", "FIELD2"),
11     placeHolder : "HTML_PLACEHOLDER"};
12   ComponentService.deep_url = 'DEEP_URL';
13   ComponentService. getComponent(datalet_params);
14 });
15 </script></head><body>
16 <div id="HTML_PLACEHOLDER"></div>
17 </body></html>
```

[8]DEEP Client JS: https://github.com/routetopa/deep-client/blob/master/js/deepClient.js

5.2.4 Controllet

In order to easily develop the code to include a datalet, a web wizard controllet[9] is available online. This wizard allows the users to create their own datalets (guiding the choice of the dataset, the type of visualization and the additional parameters) to be included in sites, blogs, forums and so on, through a copy-and-paste of its generated source code. The wizard guides also users reducing errors and helping them to understand the dataset quality.

The wizard is composed by three steps:

- The first step is the dataset selection. The Controllet provides a list of pre-configured datasets. When the user selects a dataset from the list, the system shows the relevant meta-data. Of course, the user if free to select datasets available in other external data portals by providing the data URL.
- Once the dataset has been selected, the next step is to decide what data to display in the visualization. Practically this means that the user must select the column of the datasets to pick up the data to display. Then it is possible to filter (selecting the row that satisfy some specific requirements) the data. Grouping operations are also available to group together rows based on the same value of a specific column. Of course, it is essential to specify a rule that explain how to aggregate data.
- The final step supports the selection of the visualization. The Controllet assists the user in the selection of a compatible visualization with the filtered dataset. For instance, when the user selects latitude and longitude from a dataset, the Controllet suggests the map as possible visualization.

5.2.5 DEEP Use case: *Social Platform for Open Data*

The DEEP architecture has exploited in the Social Platform for Open Data (SPOD) [Cor+16a] within the Raising Open and User-friendly Transparency-Enabling Technologies fOr Public Administrations[10] (ROUTE-TO-PA) Horizon 2020 European funded innovation project.

Within SPOD, discussions evolve using a forum-like approach, where the users post visualizations, comments and can discuss in a time-centric way. SPOD has been designed considering the well-known issues pointed out in literature for this type of interactions, such as scattered content, low-signal-to-noise ratio, dysfunctional argumentation [Kle10]. In contrast, another approach is to collaborate using a map-based visualization [KC14], where users interact by adding, moving and modifying concepts as well as linking them. This kind of visualization aims to support better exploration of the problem space, to provide a rational organization of the content,

[9] DEEP Wizard: http://deep.routetopa.eu/deep_1_9_rev/COMPONENTS/demo.html
[10] ROUTE-TO-PA site: http://routetopa.eu/about-project/

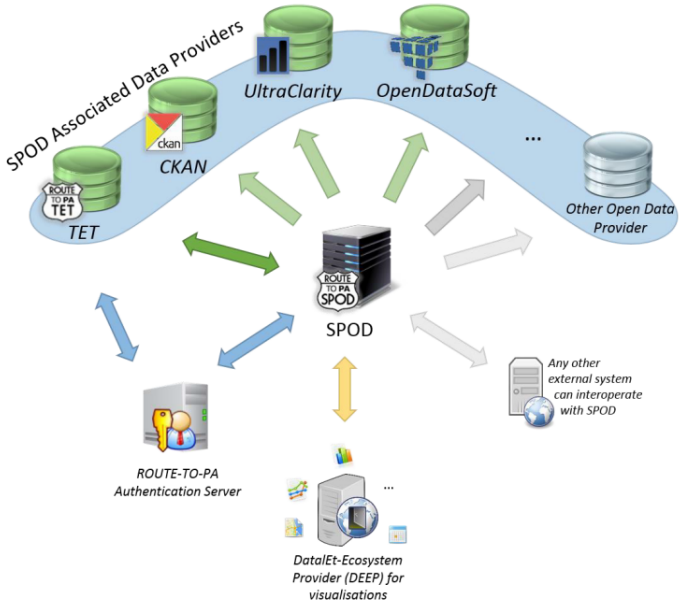

Fig. 5.7.: Overall architecture with the Open Data platforms on top, SPOD in the center, and the DEEP component, which provides visualizations of open datasets..

to stimulate critical thinking. This also showed its limitations, because it suppresses conversational dynamics and the usual reply structure, therefore disrupting the way people communicate [Ian+15]. SPOD introduces an hybrid approach where the forum-like approach is alongside the map-based visualization, and the two are interlinked. Selecting a Datalet visualization node on the map, one can see all the comments that contain the selected dataset to support their argumentations.

SPOD has built over Oxwall [Oxw16], and exploiting the DEEP architecture is able to access to external existing Open Data platforms based on CKAN [Ope16d], OpenDataSoft [Ope16b], or any other platform. Fig. 5.7 shows the architecture, where SPOD is in the middle, and on top there are the external open data portals. SPOD retrieves the dataset directly from these sources any time the Datalet is visualized. SPOD belongs a federation of other systems, thus a central authentication server, the RAS (ROUTE-TO-PA Authentication Server) has been provided. End-users have one username and password to gain access to the federation systems.

Conclusion

6

Computational Science aims to solve complex problem through a computational approach. This novel way to do science, is extremely demanding in terms of computational requirements and consequently has to exploit efficiently the resources available in HPC and Cloud infrastructures. From a Computer Science point of view, the challenge in SC is to improve the current status of methods, algorithms and applications in order to enhance the support for SC in terms of both efficiency and effectiveness of the solutions.

This thesis analyzed *Frameworks*, *Parallel Languages* and *Architectures* for SC considering the scalability of the proposed solutions as the central challenge, is to realize our idea of a *Scalable Computational Science*.

Frameworks for SCS. Problems and framework level design solutions for efficient and scalable *Distributed and Parallel Agent-Based Simulation (DPABS)* have been discussed. The proposed solutions are fully developed and tested on the DPABS framework D-MASON, a distributed version of the well known ABS Java toolkit MASON.

Summarizing, the proposed solutions in DPABS have been designed to face the following issues:

Work partitioning. The problem of decomposing a program to a set of processors (LPs) in order to achieve load balancing and efficient communication. D-MASON provides three different kinds of work partitioning strategy. The first two strategies use a space partitioning approach that exploits spatial relationships of the agents: *Uniform partitioning* in which the simulation field is evenly divided among the LPs. This solution does not consider that the agents can be non-uniformly positioned on the field; *Non-uniform partitioning* in which the simulation field is partitioned in such a way to balance the workload among the LPs. The last partitioning strategy does not use the spatial relationships, but considers that the relationships between the agents are described through a graph. In this case, the simulations use a Network field for mapping the relationships between the agents. D-MASON provides a new distributed field, called DNetwork. This field is extremely useful in several areas such as biology (cellular networks), social and political science (communication and collaboration networks) as well as chemistry (metabolic network) and so on.

Memory consistency. In an ABM, the overall system evolves in discrete events (ideally all agents change their state simultaneously). However, the agents of a region are updated sequentially. In this case, the system, or the modeler, must ensure that the accesses to the states of the agents are consistent. D-MASON solves this problem at framework-level, by exploiting the Java Method Handler mechanism.

Scalable communication. D-MASON LPs communicate via a well-known mechanism, based on the Publish/Subscribe (P/S) design pattern: a multicast channel is assigned to each topic; LPs then simply subscribe to their interested topics in order to receive relevant message updates. D-MASON is designed to be used with *any* Message Oriented Middleware that implements the PS pattern. Furthermore, D-MASON can also be deployed on HPC systems. In order to better exploit such homogeneous environments, D-MASON provides an MPI-based Publish/Subscribe communication.

Execution on Cloud Computing systems. SIMulation-as-a-Service (SIMaaS) infrastructure provides a very attractive prospective for the future environment to execute simulations, due to the good price-performance ratio. D-MASON provides easy-to-use system management based on Web technologies and tools to execute and visualize simulations on cloud computing systems.

The simulation method is mainly used to analyze behaviors that are too complex to be studied analytically, or too risky/expensive to be tested experimentally. The representation of such complex systems results in a mathematical model comprising several parameters. Hence, there arises a need for tuning a simulation model, that is finding optimal parameter values which maximize the effectiveness of the model. Considering a multi-dimensionality of the parameter space, finding out the optimal parameters configuration is not an easy undertaking and requires extensive computing power. *Simulations Optimization (SO)* and *Model Exploration (ME)* are used to refer to the techniques studied for ascertaining the parameters of the model that minimize (or maximize) given criteria (one or many), which can only be computed by performing a simulation run. Scalable solutions for such problems have been analyzed. Moreover two frameworks for SO and ME, focusing respectively on Cloud Computing (*SOF: Simulation Optimization Framework*) and on HPC systems (*EMEWS: Extreme-scale Model Exploration with Swift/T*) have been presented.

Parallel languages for SCS. The EMEWS framework is made on the top of a parallel programming language for scientific workflow, named *Swift/T*. A scientific workflow is designed specifically to compose and execute a series of computational or data manipulation steps in a scientific application. One peculiarity of Swift/T is that it enables to easily execute code written in other languages, such as C, C++, Fortran, Python, R, Tcl, Julia, Qt Script, but also invoke executable programs. EMEWS exploits the capability of Swift/T to realize SO and ME workflows adopting optimization

algorithms provided by general purpose libraries written in R and Python languages. This dissertation presented the architecture and a library that has been integrated in Swift/T (since the version 1.0) for supporting others kinds of interpreted languages. The presented architecture enables to invoke, from Swift/T, code supported by a *Java Virtual Machine* (JVM). In particular, the support for four JVM-based interpreted languages: Clojure, Groovy, Javascript and Scala, has been described.

Architectures for SCS. The final contribution of this dissertation is an architecture for scalable scientific visualization of Open-Data on the Web. The described architecture is an instance of *Edge-centric Computing* (EcC) paradigm. EcC is a novel Distributed Computing paradigm based on the observation that in computing, as in in many aspects of human activity, there has been a continuous struggle between the forces of centralization and decentralization. Following the EcC paradigm, this work presented the DatalEt-Ecosystem Provider (DEEP), an architecture for the visualization of data which enables to gather, query and visualize (dynamic) data in standard HTML pages for a massive amount of concurrent visualizations. The most important design feature concerns data manipulation that is made on the client side, and not on the server side, as in other architectures. This ensures the scalability in terms of number of concurrent visualizations, and dependability of the data and privacy (because the data is dynamically loaded client side, without any server interactions).

Appendices

A Graph Partitioning Problem for Agent-Based Simulation

As discussed in Section 2.3.1, D-MASON adopts a framework-level parallelization mechanism approach, which allows the harnessing of computational power of a parallel environment and, at the same time, hides the details of the architecture so that users, even with limited knowledge of parallel computer programming, can easily develop and run simulation models. D-MASON allows modelers to parallelize simulation based on geometric fields. It adopts a space partitioning approach, see Section 2.3.2, which allowed the balancing of workload among the resources involved for the computation with a limited amount of communication overhead.

The space partitioning approach described in Section 2.3.2 is devoted to decomposing ABMs based on geometric fields. On the other hand, when agents lie and/or interact on a network [CH05] – where the network can represent social, geographical or even a semantic space – a different approach is needed. The problem is to (dynamically) partition the network into a fixed set of sub-networks in such a way that: (i) the components have roughly the same size and (ii) both the number of connections and the communication volume between vertices belonging to different components are minimized. D-MASON enables developers to use the Network field providing its distributed version, named *D-Network* field, described in Section 2.4.1. The *D-Network* field is designed according to the following results.

A.1 *k*-way Graph partitioning Problem

Finding good network partitions (see Figure A.1) is a well-studied problem in graph theory [AK95]. Several are the problems that motivate the study of this problem. They range from computer science problems like integrated circuit design, VLSI circuits, domain decomposition for parallel computing, image segmentation, data mining [KL70; Kar+99], etc., to other problems raised by physicists, biologists, and applied mathematicians, with applications to social and biological networks (community structure detection, structuring cellular networks and matrix decomposition [Gup96; New06]).

The most common formulation of the balanced graph partitioning problem is the following:

BALANCED k-WAY PARTITIONING(G, k, ϵ).
Instance: A graph $G = (V, E)$, an integer $k > 1$ (number of components) and a

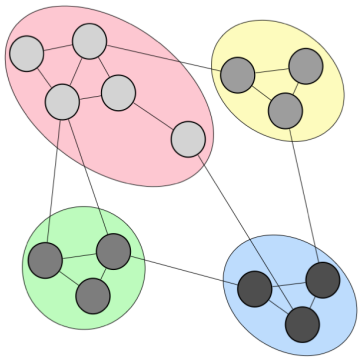

Fig. A.1.: Graph partitioning Problem for DNetwork Field.

rational ϵ (imbalance factor).
Problem: Compute a partition Π of V into k pairwise disjoint subsets (components) $V_1, V_2, \ldots, V_k$ of size at most $(1+\epsilon)\lceil |V|/k \rceil$, while minimizing the size of the edge-cut $\sum_{i<j} |E_{ij}|$, where $E_{ij} = \{(u,v) \in E : u \in V_i, v \in V_j\} \subseteq E$.

This problem has been extensively studied (see [Bad+13] for a comprehensive presentation) and is known to be NP-hard [GJ90].

Being a hard problem, exact solutions are found in reasonable time only for small graphs. However the applications of this problem require to partition much larger graphs and so several heuristic solutions have been proposed.

The graph partitioning problem was faced using several approaches. Two version of this problem have been considered: the former takes into account the coordinate information in the space of the vertices (this is common in graphs describing a physical domain) while, in the latter problem, vertices are coordinate free. In this thesis is discussed the coordinate free problem which better fits the ABMs' domain.

The graph partitioning coordinate free problem requires combinatorial heuristics to partition them. For instance, considering the simplest version of the partitioning problem (2-way partitioning), that is find a bisection of the graph $G = (V, E)$ that minimize the size of the cut. A really simple solution of the problem uses the breadth first search (BFS) visit of the graph to generate a subgraph $T = (V, E' \subseteq E)$ of G also called a BFS tree. Given the subgraph T, is possible to find a cut to generate two disjoint subnetwork N_1 and N_2 such that (i) $N_1 \cup N_2 = V$ and (ii) $|N_1| \approx |N_2|$. The fact that T has been built using the BFS ensures that the size of the edge-cut is bounded.

This solution, which works well for planar graphs, is not efficient for complex graph. A different approach is presented in the Kernighan-Lin (KL) algorithm [KL70] that, starting with two sets N_1 and N_2 (describing a partition of V), greedily improves the quality of the partitioning by iteratively swapping vertices among the two sets.

This solution converges to the global optimum if the initial partition is fairly good. Other approaches are the Spectral partitioning [ST96] and the Multilevel Approach [KK98].

The most promising techniques that either use a multilevel approach or a distributed algorithm exploiting a local search approach are:

- *METIS* is a graph multilevel k-way partitioning suite developed in the Karypis lab of University of Minnesota. Shortly, METIS comprises three phases: during the coarsening phase the vertices are collapsed in order to decrease the size of the initial graph G. Consequently, starting from $G = G_0$ a sequence of graphs $G_0, G_1, \ldots, G_\ell$ is generated. Then a k-way partitioning is performed on the smallest graph G_ℓ. Then, during the uncoarsening phase the partitioning is refined, using a variant of the KL algorithm, and is projected to larger graphs on the above sequence.
- *KaHIP (Karlsruhe High Quality Partitioning)* is a suite of graph partitioning algorithms. The suite comprises two main algorithms *KaFFPa* (Karlsruhe Fast Flow Partitioner) [SS13], which is a multilevel graph partitioning algorithm, and *KaFFPaE* (KaFFPa Evolutionary) that uses an evolutionary algorithm approach. KaFFPa, like METIS, uses the multilevel graph partitioning approach with a different strategy for the uncoarsening phase, which exploits a local search method instead of the KL approach.
- *Ja-be-Ja*[Rah+13] exploits a distributed computing approach. It uses a local search technique (simulated annealing), to find a good partitions of the graph minimizing the edge-cut size. The energy of the system is measured by counting the number of edges that have endpoints in different components. Ja-be-Ja starts with a random balanced partitioning and then it iteratively applies the local search heuristic to obtain a configuration having a lower energy state (edge-cut size). The size of the initial components is preserved since Ja-be-Ja allows only the swapping of vertices among two components.

A.2 Experiment Setting

Five k-way partitioning algorithms on several networks, taken from [Thece], were considered for the experiments. The data sets considered include networks having different structural features (see Table A.1). For each network, partitions into $k = 2, 4, 8, 16, 32$ and 64 components have been considered.

The tests performed aim to compare the analytical results obtained (i.e., size of the edge-cut, number of communication channels required and imbalance) by each algorithm with the real performances (overall simulation time) in an ABM scenario.

Name	# of vertices	# of edges	Avg degree	Max degree	Triangles	Clustering Coeff	Modularity
uk	4824	6837	2.83	3	1	0	0.7934
data	2851	15093	10.59	17	24442	0.485719	0.7596
4elt	15606	45878	5.88	10	30269	0.40765	0.6274
cti	16840	48232	5.73	6	362	0.004895	0.9063
t60k	60005	89440	2.98	3	0	0	0.5419
wing	62032	121544	3.92	4	6685	0.055595	0.5403
finan512	74752	261120	6.99	54	211456	0.503401	0.6469
fe_ocean	143437	409593	5.71	6	0	0	0.5947
powergrid	4941	6594	2.67	19	651	0.1065	0.6105

Tab. A.1.: Networks.

A.2.1 Simulation Environment

To evaluate real performances a toy distributed SIR (Susceptible, Infected, and Removed) simulation was developed, where, for each simulation step, each agent (a vertex of the network) has to communicate with its neighbors. The SIR simulation has been developed on top of D-MASON, exploiting the novel communication strategy which realizes a Publish/Subscribe paradigm through a layer based on the MPI standard [Cor+14a; Cor+14b] (see Section 2.4.2). Simulations have been performed on a cluster of eight computer nodes, each equipped as follows:

- Hardware:
  - CPUs: 2 x Intel(R) Xeon(R) CPU E5-2680 @ 2.70GHz (#core 16, #threads 32)
  - RAM: 256 GB
  - Network: adapters Intel Corporation I350 Gigabit
- Software:
  - Ubuntu 12.04.4 LTS (GNU/Linux 3.11.0-15-generic x86_64)
  - Java JDK 1.6.25
  - OpenMPI 1.7.4 (feature version of Feb 5, 2014).

Simulation results, on k-way partitioning, have been obtained using k logical processors (one logical processor per component). Notice that, when the simulation is distributed, the communication between agents in the same component is much faster than the communication between agents belonging to different components. On the other hand, balancing is important because the simulation is synchronized and evolves with the speed of the slowest component.

A.2.2 The competing algorithms

The five algorithms, briefly discussed in Section A.1, were compared:

- Multilevel approach:
  - **METIS:** (cf. Section A.1);

  - **METIS Relaxed:** this version of the METIS algorithm uses a relaxed version of the balancing constraint (i.e., a larger value of the parameter ϵ), in order to improve on other parameters (like the egde-cut size);
  - **KaFFPa:** (cf. Section A.1);
- Distributed Computing Approach:
  - **Ja-be-Ja:** (cf. Section A.1). Unfortunately, we were not able to find a real implementation of the algorithm. We used an implementation available on the public Ja-be-Ja GitHub repository [Ja-ce]. This implementation is not truly distributed but is simulated through the use of the Java library GraphChi [Grace], that enables modellers to simulate a distributed computation on multi-cores machines. Clearly the computational efficiency of this implementation is limited and, for this reason, we could only run 100 iterations of the algorithm for each test setting. We assume that the poor results of the algorithm (cf. Section A.3) are, at least, partially due to the small number of iteration used in our tests. In order to better evaluate the real performances of the algorithm, a real distributed implementation of the Ja-be-Ja algorithm is needed.
- **Random:** This algorithm assigns each vertex to a random component. It always provides an optimal balancing. This algorithm will be used as baseline in our comparisons.

A.2.3 Performance metrics

Let $G = (V, E)$ the analyzed network and let $\Pi = (V_1, V_2, \ldots, V_k)$ the partition provided by a given algorithm, we evaluate algorithms' performances using the following metrics:

- Edge-cut size (W), the total number of edges having their incident vertices in different components;
- Number of communication channels (E), two components U_1 and U_2 requires a communication channel when $\exists v_1 \in U_1,\ v_2 \in U_2$ such that $(v_1, v_2) \in E$. In other words, we are counting the number of edges in the supergraph S_G obtained by clustering the nodes of each component in a single node.
  Notice that this unconventional metric is motivated by our specific distributed ABMs scenario. In this simulation environment, a communication channel, between two components U_1 and U_2, is established when at least two vertices (agents) $u_1 \in U_1$ and $u_2 \in U_2$ share an edge. Thereafter, the same communication channel is used for every communication between U_1 and U_2, consequently, these additional communications have less impact on system performances;
- Imbalance (I), the minimum value of ϵ such that each component has size at most $(1 + \epsilon)\lceil |V|/k \rceil$.

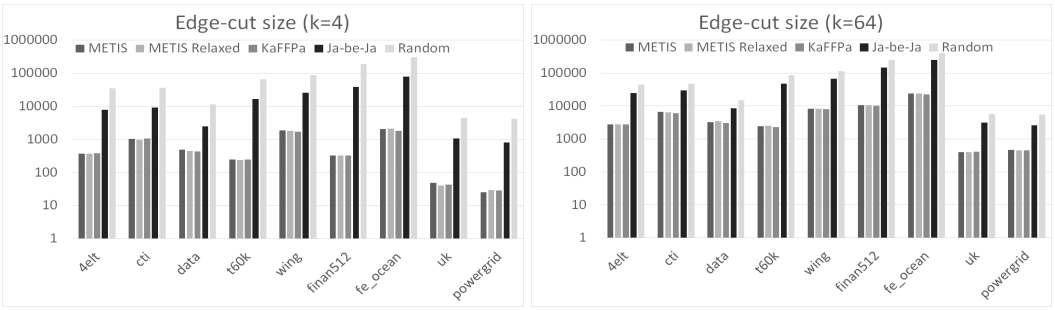

Fig. A.2.: Edge-cut size (W) comparison:(left) $k = 4$, (right) $k = 64$. Y-axes appear in log scale.

Moreover, the real performances of each strategy, by measuring the overall simulation time (T) to perform 10 simulation steps on the distributed SIR simulation, were evaluated.

Summarizing, the experiments compares the performances (both analytically and on a real setting) of 5 k-way partitioning algorithms ($A \in \{$METIS, METIS Relaxed, KaFFPa, Ja-be-Ja and Random$\}$) with $k \in \{2, 4, 8, 16, 32, 64\}$ on 9 networks ($N \in \{$uk, data, 4elt, cti, t60k, wing, finan512, fe_ocean, powergrid$\}$). Overall, $5 \times 6 \times 9 = 270$ tests have been performed.

A.3 Analytical results

Figures A.2, A.3 and A.4 depict the analytical results for $k \in \{4, 64\}$; results for the other values of k exhibit similar behaviors. For each plot the networks appear along the X-axis, while the values of the measured parameter appear along the Y-axis.

Analyzing the results from Figures A.2 and A.3 notice that the performances of the multilevel approach algorithms are comparable both in terms of edge-cut size and number of communication channels. Ja-be-Ja performances are a bit worse (this is probably due to the small number of iteration used in our tests as observed in Section A.2) but always better than the random strategy.

Results on imbalance are fluctuating (see Figure A.4). In general all the algorithms provide a quite balanced partition. Apart from the random strategy that by construction provides the optimal solution, no strategy dominates the others.

A.4 Real setting results

Figure A.5 reports on the results obtained in the real simulation setting. The results are consistent with the analytical ones, in terms of both edge-cut size and number of communication channels, although the gaps are amplified. The results thus confirm

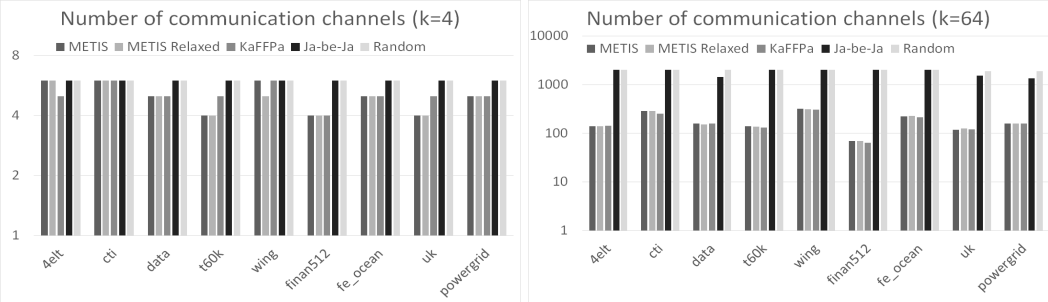

Fig. A.3.: Number of communication channels (E) comparison:(left) $k = 4$, (right) $k = 64$. Y-axes appear in log scale.

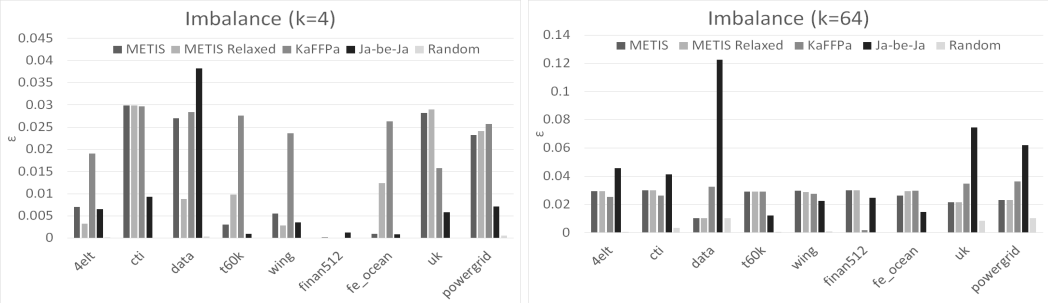

Fig. A.4.: Imbalance (I) comparison: (left) $k = 4$, (right)$k = 64$.

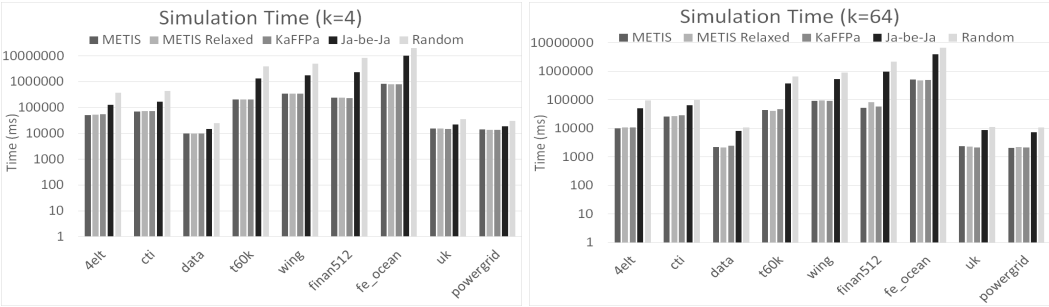

Fig. A.5.: Simulation time (T) comparison:(left) $k = 4$, (right) $k = 64$. Y-axes appear in log scale.

that the choice of partitioning strategy has a significant impact on performance in a real scenario.

In order to better understand how the metrics evolves according to k, Figure A.6 depicts four plots which describes, for each algorithm, the growth of the Edge-cut size (top-left), the Imbalance (top-right), the number of communication channels (bottom-left) and the Simulation time on the f_ocean network as function of the parameter k (X-axis).

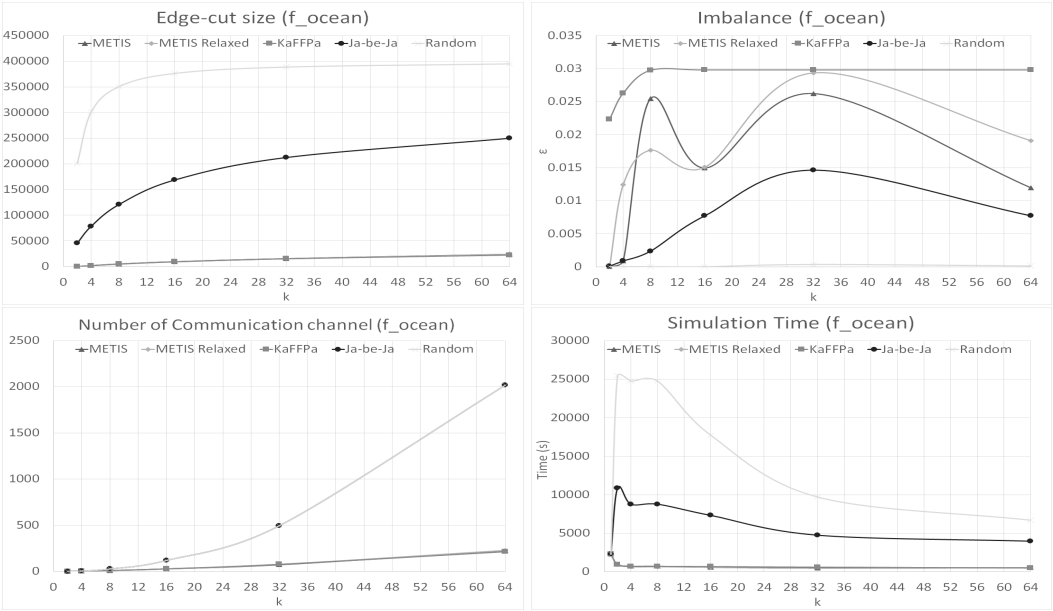

Fig. A.6.: Edge-cut size (top-left), Imbalance (top-right), Number of communication channels (bottom-left) and Simulation Time(bottom-right) on the f_ocean network, $k \in \{2, 4, 8, 16, 32, 64\}$.

A.5 Correlation between analytical and real setting results

Analyzing the results from Figures A.2 - A.5, is possible to observe that the performances of the distributed simulations are influenced by the analytical metrics.

In order to better evaluate the relation between the overall simulation times and the performances of the algorithm (measured considering the edge-cut size, the number of communication channels and the imbalance), we measured the correlation using a statistical metric: the Pearson product-moment Correlation Coefficient (PCC).

PCC is one of the measures of correlation which quantifies the strength as well as direction of the relationship between two variables. The correlation coefficient ranges from -1 (strong negative correlation) to 1 (strong positive correlation). A value of 0 implies that there is no correlation between the variables. The correlation PCC between simulation time (T) and the three analytical metrics (W, E and I) was computed, with all the considered value of the parameter k.

In particular, four variables were considered that are parametrized by the class N of Networks ($N \in \{$uk, data, 4elt, cti, t60k, wing, finan512, fe_ocean, powergrid$\}$), the considered algorithm A ($A \in \{$METIS, METIS Relaxed, KaFFPa, Ja-be-Ja and Random$\}$), and the number of components k ($k \in \{2, 4, 8, 16, 32, 64\}$).

The variable $T(N, A, k)$ denotes the Simulation time; the variable $W(N, A, k)$ denotes Edge-cut size; $E(N, A, k)$ denotes the Number of communication channels; finally the variable $I(N, A, k)$ denotes the Maximum Imbalance. Table A.2 presents the correlation values obtained.

It is possible to observe that:

- there is a strong positive correlation between simulation time and edge-cut size (the PCC always over 0.92);
- there is a weak/moderate positive correlation between simulation time and the number of communication channels[1] (the PCC ranges between 0.22 and 0.4). Moreover this correlation seems to be increasing in k;
- there is a weak negative correlation between simulation time and imbalance (the PCC ranges between -0.22 and -0.32).

This final result is counterintuitive: theoretically, the greater the imbalance, the larger the simulation time should be and this should lead to a positive correlation. The key observation is that a small amount of imbalance has a limited impact on the simulation time but can be extremely helpful for reducing both the edge-cut size and the number of communication channels, which seems to have a sensible payoff in terms of real performances.

	k					
	2	4	8	16	32	64
$r(T,\ W)$	**0.9256**	**0.9392**	**0.9431**	**0.9424**	**0.9473**	**0.9474**
$r(T,\ E)$	N.A.	0.2265	0.3094	0.3349	0.3509	0.3922
$r(T,\ I)$	-0.2244	-0.2750	-0.2903	-0.3188	-0.2971	-0.3025

Tab. A.2.: Correlation between analytical and real setting results.

A.6 Best practices for ABS and Network

Considering the problem of partitioning a network into k balanced components such that the number of edges that cross the boundaries of components is minimized. Experimental results show that the choice of the partitioning strategy strongly influence the performance of a real distributed environment. Moreover analytical results (the edge-cut size in particular) correlate with the overall simulation time in a real setting. On the other hand, according to the previous results, the quality of the balance among components does not relate to the real performances on the field. The best practice is to partition the graph using one of these heuristics (*METIS* show the best results) and load the agents in the simulation according to the given partition.

[1] The correlation between $T(N, A, 2)$ and $E(N, A, 2)$ cannot be computed, since for $k = 2$ all the partitioning strategy require exactly 1 communication channel and so $E(N, A, 2)$ has standard deviation equal to 0.

D-MASON Work Partitioning and GIS

B

— More realistic (complex) is better? — The design of a social simulation model is an intricate task [GT05]. Social dynamics are the result of complex structures of interactions that involve at different levels individual cognition and behavior, groups, institutions and the surrounding environment. The modeling enterprise implies the operational description of factors involved in generating the macro phenomenon under investigation.

This Appendix describes some technical issues arising when dealing with simulations with a high number of agents, real data, and GIS based environment. The goal is twofold: analyze computational and programming issues arising when adding complexity to a relatively simple social simulation model, in particular showing how to use GIS in D-MASON simulation; show how distributed computing can support advances in the investigation of social issues.

B.1 Agent-based model and Geographical Information Systems

GIS [GIS16] (Geographic Information Science or Geographic Information Systems) term refers to a set of theories and techniques (especially computer-based) that enable to add geographical data and metadata in the modeling of ABM. GIS has been used in many fields as geography, geology, ecology, sociology, urban planning, health studies, and others. The application of GIS data in the field of ABM is relatively recent, but the interest in this field led to the creation of dedicated community [com] and, as described in the chapter four of a recent book *"Geocomputation: a practical primer"* [BS15], the interest in this field is intended to grow.

GIS data into an ABM model, in a first instance, may be seen as an unnecessary level of details, but simulation is a model based approximation of a real system and so adding more details may produce better results. GIS can be thought of as a high-tech equivalent of a geospatial map. This high-tech map provides a number of additional data about the environments and the status of it. This information can be used to model more complex interactions between the agents and the environment. Hence, the complexity of the models getting bigger inevitably requires the introduction of parallel and distributed computing.

Many ABM tools support the GIS data. Among them we have:

- MASON [Bal+03] toolkit (see also Section 2.2). MASON provides support for GIS data in an additional library named GeoMASON[Sul+10]. GeoMASON follows the same MASON design philosophy of being lightweight, modular, and efficient. GeoMASON represents the basic GIS data in a geometric shapes supported via the *JTS* (Java Topology Suite API), which allows geometries related operations. GeoMASON supports the ESRI [for] shape files providing a `GeomVectorField` Java object that represents the GIS data in the memory, and provides functionalities to access to geometries in order to read geospatial metadata and obtain geospatial positions of the objects.
- Netlogo [TW04] is a multi-agent programmable modeling environment. It is developed at the The Center for Connected Learning (CCL) and Computer-Based Modeling at Northwestern University. Netlogo provides an extension to support GIS data field. The extension allows the programmer to load vector GIS data (points, lines, and polygons), and raster GIS data (grids) into their models. Netlogo extension also supports the ESRI shapefiles.

The next sections describes how to use GIS in D-MASON simulation and the results obtained on a GIS simulation scenario.

B.2 The simulation scenario

This section describes the simulation model used to experiment the distribution of ABM that uses GIS in D-MASON.

B.2.1 Model space representation

The environment in ABM [Ben13] is not only a particular property of the model, but could be a relevant entity to understand the complex behavior of natural and artificial systems. Interesting features of ABM, compared to others modeling tools, concern the interactions between the agents that do not take place in a vacuum, but happen in a structured environment that could influence and could be influenced by the agents interactions. These structured environments are named fields and can be social and physical environments (fields based on mathematical modeling like matrix), but also more complex structure like networks.

The environment representation is really crucial in order to address real-world problems (e.g., simulating the segregation in a particular area or simulating emergency strategies in natural disaster [CW13]). In this work we used a set of GIS data (*"Campania dataset"*) to support a model that simulates an experiment, inspired by a cognitive architecture model [BS15].

GIS Campania dataset. The open-data platform of the *"Regione Campania"* provides the GIS dataset [Cam15] about its region. The dataset is a ShapeFile ESRI shape

on the geographical coordinate system WGS84 UTM zone 32N and provides the subdivision of the region in geographical zones identified by a unique identifier.

B.2.2 Model agents movement representation

A single agent in our model is a citizen living in Campania that has to travel every day to his work/study place. Agents behavior is based upon public available data released by ISTAT [dat16] (the Italian National Institute of Statistics), produced after the national population census made in October 2011.

The ISTAT's table contains information about 2.5M citizens living in Campania, which each day travel to work (or study) and then go back home. It contains two kind of records: a *S*-record and a more detailed *L*-record.

L-records exposes the following data: city of residence, gender, reason to travel (work or study), city of work/study, vehicle (on foot, by car, by train, etc.), time of departure, travel duration.

Since each record is aggregate (that is, each record represents several people with the same attributes), *time of departure* and *travel duration* fields are quantized, meaning that data is represented in classes. For instance, travel duration can be one of these classes: up to 15 minutes; from 16 to 30 minutes; from 31 to 60 minutes; 61 minutes or more. Considering the table format, we chose to randomize data. For instance, if a record says that 100 people travel for $16-30$ minutes, each of the 100 agents created will travel for a number of minutes randomly chosen in the $16-30$ minutes interval.

B.2.3 Simulation Model description

The model is a simplified version of the cognitive model designed by *Andrighetto et al.* in [And+14]. In the proposed model, agents move in a representation of the Campania environment: agents' home and working places are placed on the Campania map according to real data from ISTAT.

Here is described the cognitive aspect of the model. There is a set of norms (graphically depicted using different colors). For each norm, agents will hold a *salience* (a value in the [0,1] interval) that represents how much that norm is important for the agent. The norm that is the most important for an agent, will characterize the agent's color (belief of the norm).

Agents continuously interact with neighbor agents trying to spread their opinion (color). When an agent meet an agent advertising a particular color, it will increase the salience for that specific norm. Agents are influenced by neighbors throughout the day, but with different weight depending on agent's state (traveling, staying at home, staying at work/study). Of course, saliences will naturally decay over time.

Example. Imagine there is an agent A of color Blue. During its travel and staying at work, he meets a lot of Red agents. He will so receive a lot of input about the Red norm, while salience about the Blue norm decreases time after time. At some point, agent A will turn Red and start spreading to other agents that Red is the *right* norm to follow, thus trying to convert other agents to its color.

Campania is divided into five *provinces*: Naples, Salerno, Benevento, Avellino and Caserta. Suppose that in each province of Campania there is a norm that is prevalent (i.e. 80% of the inhabitants are of that color). For instance: Naples is mostly Red; Salerno is mostly Green; Avellino, Benevento, Caserta are mostly Blue. In the remaining 20% of the population, the 15% will be Yellow (the color not advertized by any region), while the last 5% will be of a random color (different from the region's color). Back to our example, 80% of people in the Salerno province will be Green; 15% will be Yellow, and the remaining 5% will be a random chosen between Red and Blue.

The model simulates an entire day (24 hours) starting from midnight. When the simulation starts, agents are at their home. Time of departure, travel duration, time of stay at work/study all depend on ISTAT data. Times and durations need to be converted into simulation steps: for instance, if travel duration for the agent is *from* 16 *to* 30 *minutes*, the simulation will assign a random duration in that interval. This duration will be converted in a number of steps, according to discretization time of the simulation (see section B.4).

The size of the simulated field and discretization time have a significant impact on the performances (in terms of efficiency) of the simulation. An agent moves at a speed that is calculated dividing the travel distance by the travel duration (simulation steps). This gives the speed of an agents, that is the maximum distance covered in a single simulation step. The largest agents' travel distance is called maximum agents ride (α) and will require a certain number of steps (maximum number of steps to perform a ride, β). So we can compute the maximum speed (α/β): this parameter has a strong impact on the distributed model performances (the smaller the better).

In order to evaluate the performances of the distributed simulation framework D-MASON, two modifications of the model, concerning the way in which the agents are placed on the map, have been included. Three different agents model distribution are considered. ***Real*** positions; agents are place according to the real population density. A second model, called ***CRandom***, places agents uniformly at random on the entire Campania territory. The latest model, called ***Random***, places agents uniformly random on a 2D continuous space. The latest model represents the best case for distribution, as it allows the model to balance the workload on multiple LPs (Logical Processors). It is, although, very unrealistic. The ***CRandom*** model represents an

in-between case, since agents are uniformly distributed on the territory, but are still placed within the Campania boundaries.

B.3 D-MASON Simulation

Noticing that most ABMs are inspired by natural models, where agents' limited visibility allowa to bound the range of interaction to a fixed range named agent's Area of Interest (AOI), D-MASON adopta the so-called *space-partitioning* technique [Cos+11], where the agents' world (the *field*) is split into tiles, each assigned to an LP.

Since citizens are basically moving on a map, the agents' space consists in a rectangular area. MASON includes the `Continuous2D` field, where agents contained in it are located by a couple of continuous coordinates ranging from point (0,0) to point ($[W]idth$, $[H]eight$)). The distributed version embedded into D-MASON is called `DContinuous2D`: it retains all the features of the `Continuous2D` field, adding the support for two approaches to distribute the field and agents contained in it: dividing the space into vertical rectangles (1-dimensional space partitioning or *horizontal*); or dividing it into a $rows \times columns$ matrix (2-dimensional space partitioning or *square*, see Figure B.3) . The square partitioning (space-based) mode provides a significant speedup over the horizontal partitioning, lowering the communication effort while distributing the computational workload of the agents to LPs.

The behavior of agents is influenced by GIS data (map, zones and cities), nevertheless GIS data is static and does not require any synchronization among LPs.

When reading ISTAT data, each LP manages an area of competence, and take care of agents that live in its area of competence. This is done by reading agent's home location from the ISTAT dataset, looking for correspondent coordinates into GIS data, and converting it into 2-dimensional D-MASON coordinates.

B.4 Experiments

Simulation Environment. Server test was performed to evaluate the performance of the model in D-MASON. Both the communications strategies available in D-MASON were tested: AMQ (Apache ActiveMQ) that is the centralized communication strategy; and MPI that uses the MPI standard [Cor+14a; Cor+14b]. Simulations have been performed on a cluster of eight computer nodes, each equipped as follows:

- Hardware:
  - CPUs: 2 x Intel(R) Xeon(R) CPU E5-2680 @ 2.70GHz (#core 16, #threads 32)
  - RAM: 256 GB

  - Network: adapters Intel Corporation I350 Gigabit
- Software:
  - Ubuntu 12.04.4 LTS (GNU/Linux 3.11.0-15-generic x86_64)
  - Java JDK 1.6.25
  - OpenMPI 1.7.4 (feature version of Feb 5, 2014).

Experiments settings. The scalability of the simulation considering the overall simulation time needed to simulate a 5 (simulated) minutes of real world system, was investigated changing both the number of LPs and the simulation workload (# of agents). As described above the model uses a discretization time to simulate the real life clock. In the following tests, the discretization time is 2400 steps per hour, the field size is 3600×2400 and the neighbors' influence radius (NIR) is 1.

D-MASON allows two kinds of square space partitioning: 1-dimensional and 2-dimensional. After several pilot experiments, only the 2-dimensional partitioning was considered. The unbalanced density of agents will be a crucial part of our investigation.

The performance trend was investigated by varying the agents positioning over the field: ***Real*** that is the positioning of the agents among the region using the real data; ***Random*** is positioning strategy that set the agents uniformly random among the whole field; ***CRandom*** sets the agents among the field uniformly random but only in the limits of the region.

Two kinds of experiments are presented:

Simulate 5 minutes of real life. In this test we are interested in evaluating *How much time is needed to simulate* 5 *minutes of real life?* (which corresponds to 200 simulation steps). Four configurations were tested using different partitioning scheme of the field in 4×4, 6×6, 8×8 and 10×10 cells assigned, respectively, to 16, 36, 64 and 100 LPs (each test uses one region per LP). Each configuration was performed on 2.5 million of agents. Figure B.1 shows the results for each model (*Real*, *Random* and *CRandom*) and for each communication layer (MPI and AMQ). For each configuration, we show the total simulation time as well as how it is partitioned into the communication overhead (that includes the management overhead introduced by D-MASON) and the computation time.

The performance of the simulation is strongly influenced by the positioning model. The *Random* and *CRandom* models exhibit the same unimodal trend, as the number of LPs increase and manifest a balanced ratio of communication and computation.

The *Real* test provides the worst performance and unusual trend due to an unbalanced communication overhead. We investigated this problem analyzing the simulations with different agents positioning models and we discovered that this trend is due to the non uniform positioning of the agents (see Figure B.4).

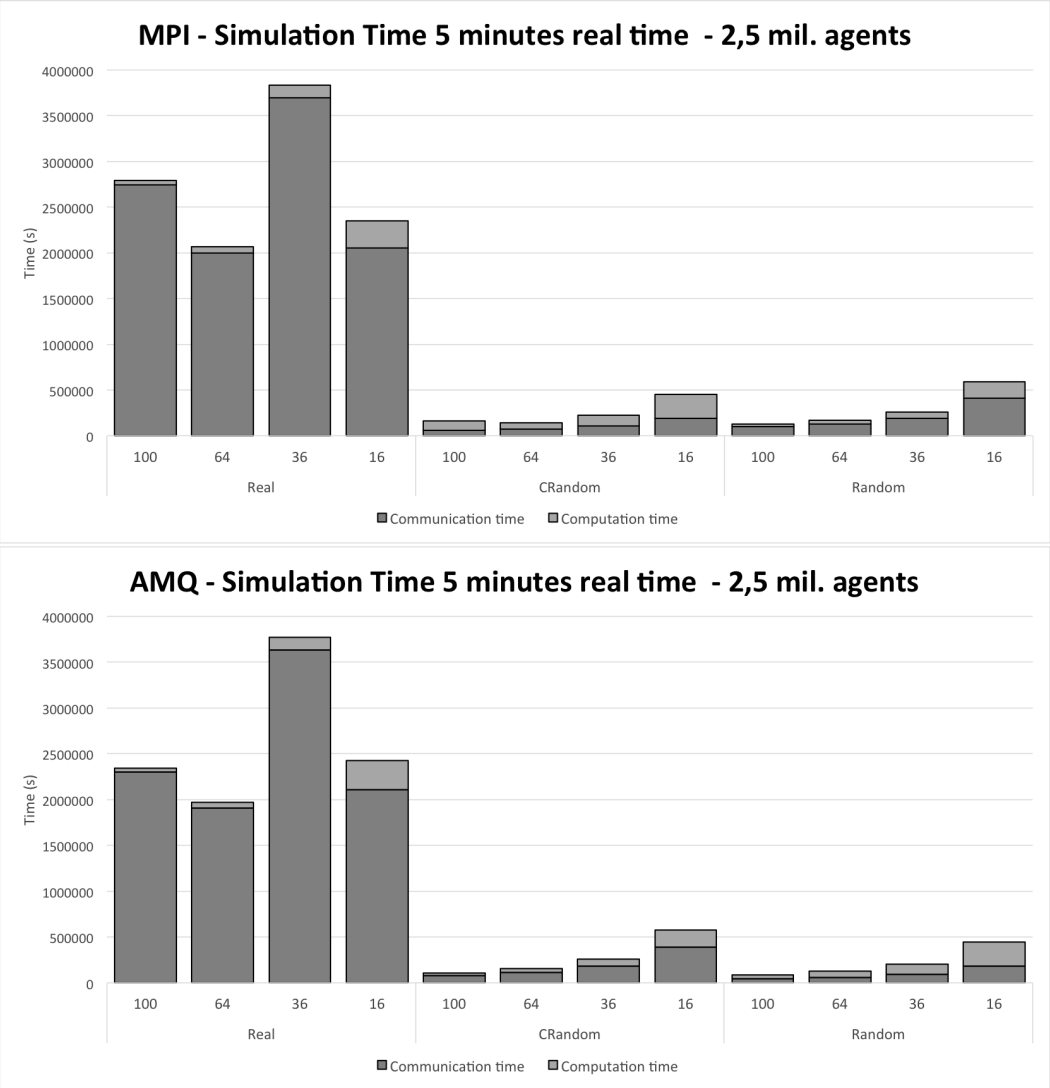

Fig. B.1.: Simulation performance with 4×4, 6×6 , 8×8, 10×10 partitionings - 5 minutes of real clock.

Weak scalability. This experiment aims to evaluate the simulation efficiency varying the total computation workload. In this test four configurations was considered by changing the total number of agents $10\%, 40\%, 70\%$ and 100% ($100\% = 2.5$M). Each configuration was performed on a 10×10 partitioning with 100 LPs. Figure B.2 depicts the results of the three models for each communication layer. Moreover we also compare the performances with the sequential version of the model implemented in MASON (we refers to this with the name SEQ). *Random* and *CRandom* tests provide a similar behavior showing good scalability. This results demonstrate the good performance of a 2-dimensional field partitioning on a uniform and quasi-uniform positioning density. On the other hand, the *Real* model manifests the worst scalability (just a bit better than the sequential version). This result is due to the communications overhead that is extremely irregular over the LPs. The table B.1 reports the speedup obtained during the weak scalability test. For each configuration,

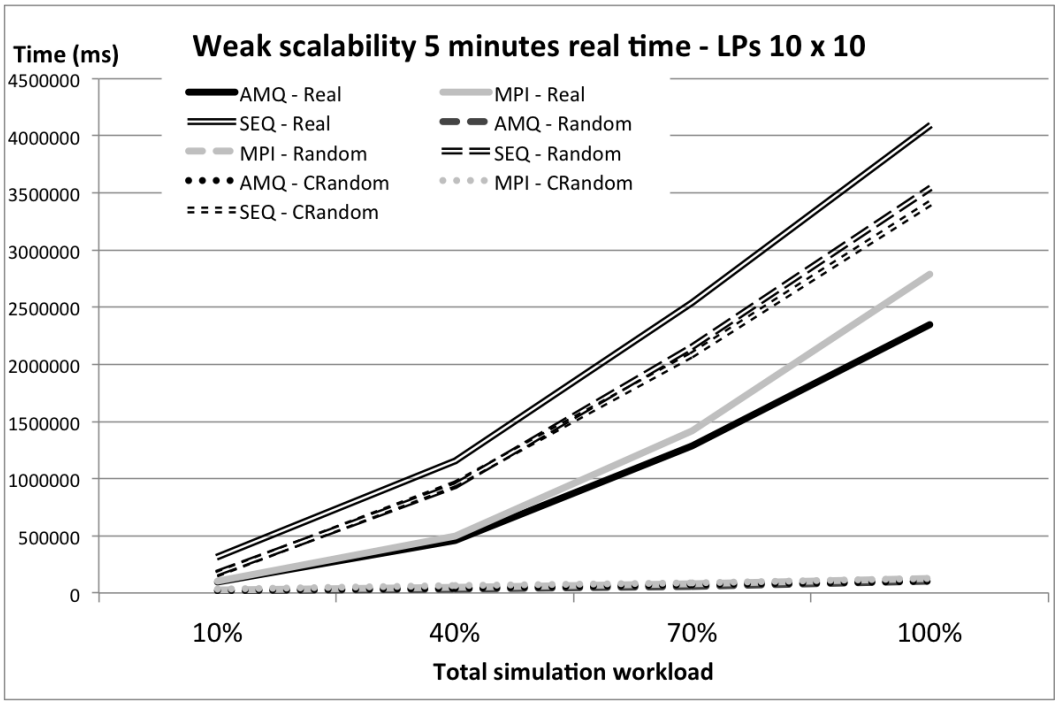

Fig. B.2.: Weak scalability. 10×10 partitioning, 5 minutes of real clock.

the minimum and maximum speedup are emphasized in bold. The best results have been obtained by the *Random* model with 70% of computation amount and *AMQ* as communication layer; the worst performance is achieved by the *Real* positioning using the *MPI* communication layer. This confirms the hypothesis that the speedup is strongly related to agents distribution.

	Workload			
Test Name	*10 %*	*40%*	*70%*	*100%*
AMQ - Real	**3,36**	2,49	1,97	1,74
MPI - Real	3,07	2,32	1,79	**1,46**
AMQ - Random	11,32	25,14	**35,53**	33,06
MPI - Random	**7,63**	20,01	27,29	27,78
AMQ - CRandom	7,69	21,13	29,14	**31,86**
MPI - CRandom	**5,60**	15,53	23,38	26,78

Tab. B.1.: Experiments speedup varying the workload. 10×10 partitioning, 5 minutes of real clock.

B.5 Analytical analysis of ABM and GIS

Considering the results obtained in the preceding section, in this sections is described an analytically evaluation of the communication effort required by a GIS based distributed simulation that exploits a uniform 2D space partitioning approach (Figure B.3).

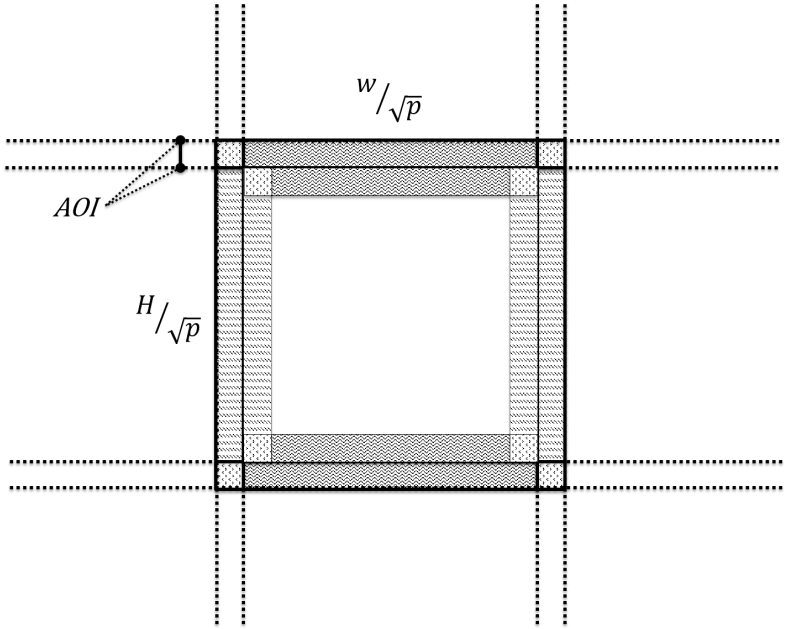

Fig. B.3.: D-MASON 2-dimensional uniform field partitioning on p tiles.

When a space partitioning approach is used, the amount of communication performed before each simulation step is related to: the size of the whole field ($[W]idth \times [H]eight$ in this specific analysis), the agents density distribution (d) i.e., the positioning of the agents over the field, the number of LPs (p), the maximum agents ride distance (α), the maximum number of steps to perform a ride (β) and the agents area of interest radius (AOI) which depends on the neighbors' influence radius (NIR).

Recalling that using the space partitioning approach we have

$$AOI \geq \max\left(NIR, \frac{\alpha}{\beta}\right), \tag{B.1}$$

therefore the AOI should be at least equal to the ratio α/β.

Since, in euclidean space, the diameter of our field is $\alpha = \sqrt{W^2 + H^2}$, one can easily verify that

$$\frac{W+H}{2} \leq \alpha \leq W + H. \tag{B.2}$$

Hence, by using (B.1) and (B.2) we have that

$$AOI \geq \frac{\sqrt{W^2+H^2}}{\beta} \geq \frac{W+H}{2\beta} \tag{B.3}$$

We can now evaluate the communication effort required by a GIS based distributed simulation. For each region, the communication effort δ_c is obtained by counting

the number of agents which belong to the edges of the region. The edges space, as shown in figure B.3, is composed by 16 regions of sizes $\frac{W}{\sqrt{p}} \times AOI$ (top and bottom), $\frac{H}{\sqrt{p}} \times AOI$ (left and right) and $AOI \times AOI$ (corners). The expected number of agents is obtained multiplying the size of the above described region by the density. Overall we have,

$$\begin{aligned}
\delta_c &= p \times \left[4\left(\frac{W}{\sqrt{p}} AOI\right) + 4\left(\frac{H}{\sqrt{p}} AOI\right) + 8AOI^2 \right] \times d \\
&= 4p \times d \times \left[AOI \times \left(\frac{W+H}{\sqrt{p}}\right)\right] + 8p \times d \times AOI^2 \\
&= 4\sqrt{p} \times d \times AOI \times (W+H) + 8p \times d \times AOI^2 \\
&\leq 8\sqrt{p} \times d \times \beta \times AOI^2 + 8p \times d \times AOI^2 \\
&= 8\sqrt{p} \times d \times AOI^2 \times (\beta + \sqrt{p}) \qquad \text{(B.4)}
\end{aligned}$$

where the inequality is due to Equation (B.3).

Consequently the communication effort is influenced by the AOI (which depends on the simulation model) and the density distribution of agents (d). In details the value of δ_c varies according to the agents positioning over the field. When such value is irregular, the communication increases and affects all the regions since the whole system synchronizes before each simulation step.

This analysis motivates the poor performance of the simulation in the *Real* agents positioning experiment. Figure B.4 depicts the positioning of the agents on the geographical zones in the *Campania* region. Real positioning provides a lots of zones with a small number of agents but there are also a small number of highly populated zones. Indeed, the density d over the field is non-uniform (the variance, in the number of agents per zone, is 302600129.2) and by equation (B.3) the communication effort δ_c grows proportionally with the larger value of d.

B.6 Motivation to *Non*-Uniform D-MASON work partitioning strategy

Exploiting GIS data in ABM is an important innovation in the ABS. Several ABM examples [CW13] and users community [com] demonstrate the importance of this approach for improving the effectiveness of ABM model in complex systems study.

Experimental results on a toy model, inspired by [And+14], demonstrate that the work partitioning, in a distributed GIS based ABM simulation is quite hard. According to our analysis, the main issue is the non-uniform distribution of the agents over the field, which jeopardize the performance of the simulation. Indeed,

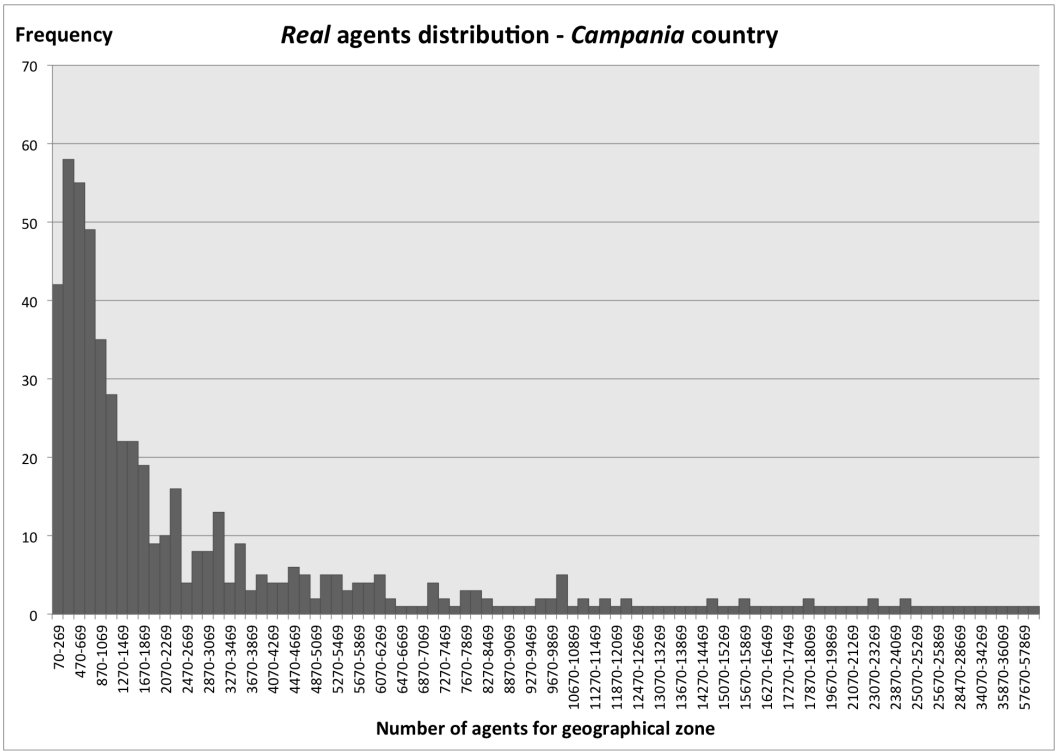

Fig. B.4.: Agents positioning over the region zones in *Campania*. In the figure is shown the frequency of zones in *Campania* with a certain density that ranging from 70 to 57869 people.

the speedup depends on communication effort δ_c which, in a 2-dimensional uniform field partitioning approach, is:

$$\delta_c \leq 8\sqrt{p} \times d \times AOI^2 \times (\beta + \sqrt{p}).$$

Therefore, the performance of the simulation scale up as a quadratic function of the AOI (which depends on the model) and is linearly influenced by the density distribution (d) of the agents over the field and the discretization time (β) used in the model.

The above results motivate the need to have a non-uniform work partitioning strategy in DPABS, that has been developed and is described in Section 2.3.2. Section 2.6.1 provides a complete analysis of the performances of the non-uniform partitioning strategy in a real simulation scenario. These analysis shows that the non-uniform partitioning strategy, ensures better load balancing but introduces a communication overhead.

C D-MASON: The Developer Point of View

This appendix aims to provide a guide for D-MASON simulation model developers. In the following an example of simulation porting from MASON to D-MASON is discussed. Thereafter, the usage of D-MASON on HPC systems is briefly described.

C.1 Simulation Example: Particles

As described in the Chapter 2 one of the D-MASON goal was to maintain the compatibility with MASON. This section provides an example of simulation porting from MASON to D-MASON. The considered simulation is the MASON simulation *Particles* (available in the *Tutorial3* of MASON [MAS16]). The Particles simulation is an example of 2D particles bouncing around on a grid. When particles collide, they bounce off in a random direction (including staying put). Moreover, the particles bounce off the wall and leave trails of their passage.

According to the MASON guidelines a simulation package is composed by three classes:

1. *agent*, a class that implements the simulation agents;
2. *state*, a class that represents the simulation environment;
3. *gui*, a class that provides visualization of the simulation.

As discussed in Section 2.4, D-MASON provides the distributed version of the three main classes above:

1. *distributed agent*, a class that implements the simulation agents in a distributed system;
2. *distributed state*, a class that represents the simulation environment in a distributed system;
3. *gui*, a class that provide the visualization of the simulation in the distributed system.

Particles Simulation. The package `sim.app.tutorial3` is composed by three classes:

1. `Particle`, which implements the agent;
2. `Tutorial3`, which implements the simulation environment;
3. `Tutorial3WithUI`, which implements the simulation GUI.

In D-MASON the simulation is named *DParticles* and is available on the D-MASON source code repository [Repce].

C.1.1 (D)Agent definition

The MASON `Particle` agent implements the `Steppable` interface and, in particular, the `step()` method defines the agent's behavior. In D-MASON the abstract class `RemoteParticle` implements `RemotePositionedAgent` that is a sub-interface of `Steppable`. The concrete implementation of agents in D-MASON is provided by the class `DParticle` that extends the class `RemoteParticle`.

In this example, the class `RemotePositionsAgent` is parameterized with an Int2D object-type, that corresponds to the object used to maintain the agent's position.

The `Particle` class, shown in the Listing C.1, implements the interface `Steppable` and declares a constructor that takes two integer parameters, xdir and ydir (which represents the direction of a particle).

Listing C.1: Particle constructor

```
1  ...
2  public  Particle implements Steppable {
3      public boolean randomize = false;
4      public int xdir;  // -1, 0, or 1
5      public int ydir;  // -1, 0, or 1
6
7      public Particle(int xdir, int ydir)
8      {
9          this.xdir = xdir;
10         this.ydir = ydir;
11      }
12
13 ...
```

In D-MASON, the `DParticle` class, shown in the Listing C.2, extends the `RemoteParticle` and declares two constructors: the first takes no parameters and it is required for D-MASON object serialization, and the second that takes a `DistributedState` as parameter.

Listing C.2: DParticle constructor

```
1 ...
2 public class DParticle extends RemoteParticle<Int2D> {
3     public int xdir; // -1, 0, or 1
4     public int ydir; // -1, 0, or 1
5     public DParticle(){ }
6     public DParticle(DistributedState state) {
7         super(state);
8     }
9 ...
```

The agent's behavior is defined for both version in the method `step`. The Listing C.3 and C.4 show, respectively, the code for the MASON and D-MASON implementations.

The reader can observe that the two implementation are basically the same, the difference consists in the way the particles movements are randomized. The changes in the two implementation are mainly due to the fact that MASON and D-MASON exploit a different synchronization strategy: as discussed in Section 2.4.1, D-MASON provides a self-synchronized environment where, all the agents update their status at step t, considering the status of all neighbor agents at step $t-1$(synchronous update); on the other hand, in MASON, agents are updated asynchronously, that is, their status reflect the changes of the agents that have already performed their computing in the current iteration. Consequently, the randomization of the movement in D-MASON is computed, synchronously, at the beginning of each agents `step` method while in MASON the randomization is performed asynchronously.

Furthermore, the method used to update the position of the agent is `setDistributedObjectLocation`. This method enables to change the position of the agent on the field handling the migration of the agent to another cells whenever is needs.

Listing C.3: Particle step method

```
1
2 ....
3 public void step(SimState state) {
4    Tutorial3 tut = (Tutorial3)state;
5          Int2D location = tut.particles.getObjectLocation(this);
6          tut.trails.field[location.x][location.y] = 1.0;
7          if (randomize)
8          {
9             xdir = tut.random.nextInt(3) - 1;
10            ydir = tut.random.nextInt(3) - 1;
11            randomize = false;
12         }
13        int newx = location.x + xdir;
```

```
14          int newy = location.y + ydir;
15          if (newx < 0) { newx++; xdir = -xdir; }
16          else if (newx >= tut.trails.getWidth()) {newx--; xdir =-xdir;
                }
17          if (newy < 0) { newy++ ; ydir = -ydir; }
18          else if (newy >= tut.trails.getHeight()) {newy--; ydir =
                -ydir; }
19          Int2D newloc = new Int2D(newx,newy);
20          tut.particles.setObjectLocation(this,newloc);
21
22          Bag p = tut.particles.getObjectsAtLocation(newloc);
23          if (p.numObjs > 1)
24          {
25              for(int x=0;x<p.numObjs;x++)
26                  ((Particle)(p.objs[x])).randomize = true;
27            }
28  }
```

Listing C.4: DParticle step method

```
1  ....
2  ....
3  public void step(SimState state) {
4     DParticles tut = (DParticles)state;
5           Int2D location = tut.particles.getObjectLocation(this);
6           tut.trails.setDistributedObjectLocation(location,1.0,state);
7           Bag p = tut.particles.getObjectsAtLocation(location);
8           if (p.numObjs > 1)
9           {
10                  xdir = tut.random.nextInt(3) - 1;
11              ydir = tut.random.nextInt(3) - 1;
12          }
13          int newx = location.x + xdir;
14          int newy = location.y + ydir;
15          if (newx < 0) { newx++; xdir = -xdir; }
16          else if (newx >= tut.gridWidth) {newx--; xdir = -xdir; }
17          if (newy < 0) { newy++ ; ydir = -ydir; }
18          else if (newy >= tut.gridHeight) {newy--; ydir = -ydir; }
19          Int2D newloc = new Int2D(newx,newy);
20          tut.particles.setDistributedObjectLocation(newloc, this,
                state);
21 }
22 ....
```

C.1.2 (D)Simulation State

The second class is the simulation state. In MASON the class `Tutorial3` extends the `SimState` class, while in D-MASON the class `DParticles` extends `DistributedState` and is parametrized with an `Int2D` object.

The Listing C.5 shows the significant part of the `Tutorial3` class in MASON. Listing C.6 shows the code for the D-MASON class `DParticles`.

The mainly differences between the two implementations are:

- *The initializations of the simulation field and of the agents.* In `Tutorial3` there are two fields: the former contains the agents, while the latter contains the trails. In the original simulation, after the initialization of simulation fields all agents are initialized and randomly positioned over the field. In D-MASON not all agents can be initialized and positioned, but only the agents that must be simulated by the corresponding LP. Hence, each LP initializes a portion of the agents (proportional to the size of the associated cells) and position them to the associated cell. The method `getAvailableRandomLocation` enable to generate a random position on a particular cell of a D-MASON simulation field.
- *The method used to schedule agents.* MASON schedules the agents using the method `scheduleRepeating`, which schedules an agent, for each simulation step. In D-MASON agents can migrate from one LP to another. For this reason the method `scheduleRepeating` is forbidden (it may happen that in a successive step the agent must be scheduled by another LP). In D-MASON uses the method `scheduleOnce` that schedule an agent only for the successive simulation step. Hence, the method `scheduleOnce` has to be executed for each simulation step. This is performed by the method `setDistributedObjectLocation` which, handling the migration of agents, id always aware of the agents that need to scheduled for each simulation step.

Listing C.5: Tutorial3

```
1  public class Tutorial3 extends SimState {
2      public DoubleGrid2D trails;
3      public SparseGrid2D particles;
4      ...
5      public Tutorial3(long seed) {
6          super(seed);
7      }
8      public void start() {
9          ...
10         for(int i=0 ; i<numParticles ; i++) {
11             p = new Particle(random.nextInt(3) - 1, random.nextInt(3)
                   - 1); // random direction
12             schedule.scheduleRepeating(p);
13             ...
14             particles.setObjectLocation(p,new Int2D(x,y)); // random
                   location
15         }
16     }
17     public static void main(String[] args) {
18         doLoop(Particles.class, args);
```

```
19            System.exit(0);
20        }
21    }
```

Listing C.6: DParticles

```
1  ...
2  public class DParticles extends DistributedState<Int2D> {
3      protected DSparseGrid2D particles;
4      protected DDoubleGrid2D trails;
5      protected SparseGridPortrayal2D p;
6      public int gridWidth ;
7      public int gridHeight ;
8      public int MODE;
9      private String topicPrefix = "";
10     public DParticles(){ super();}
11     public DParticles(GeneralParam params, String prefix)
12     {
13       super(params,new DistributedMultiSchedule<Int2D>(),
14       prefix,params.getConnectionType());
15       this.MODE=params.getMode();
16       this.topicPrefix=prefix;
17       gridWidth=params.getWidth();
18       gridHeight=params.getHeight();
19     }
20
21     @Override
22     public void start(){
23      super.start();
24      try{
25
26         particles =
               DSparseGrid2DFactory.createDSparseGrid2D(gridWidth,
27               gridHeight,this,
28                     super.AOI,TYPE.pos_i,TYPE.pos_j,
29                     super.rows,super.columns,MODE,
30                     "particles", topicPrefix,false);
31         trails = DDoubleGrid2DFactory.createDDoubleGrid2D(gridWidth,
32               gridHeight,this,
33                     super.AOI,TYPE.pos_i,TYPE.pos_j,
34                     super.rows,super.columns,MODE,
35                     0,false,"trails",topicPrefix,false);
36
37           init_connection();
38
39      }catch (DMasonException e) { e.printStackTrace();}
40
41      DParticle p=new DParticle(this);
42      int agentsToCreate=0;
43      int remainder=super.NUMAGENTS%super.NUMPEERS;
44      if(remainder==0){
```

```
45      agentsToCreate= super.NUMAGENTS / super.NUMPEERS;
46        }
47
48        else if(remainder!=0 && TYPE.pos_i==0 && TYPE.pos_j==0){
49            agentsToCreate= (super.NUMAGENTS /
50                        super.NUMPEERS)+remainder;
51        }
52        else{
53          agentsToCreate= super.NUMAGENTS / super.NUMPEERS;
54        }
55
56        while(particles.size() != agentsToCreate)
57
58           p.setPos(particles.getAvailableRandomLocation());
59           p.xdir = random.nextInt(3)-1;
60           p.ydir = random.nextInt(3)-1;
61
62        if(particles.setObjectLocation(p, new
              Int2D(p.pos.getX(),p.pos.getY())))
63        {
64           schedule.scheduleOnce(schedule.getTime()+1.0,p);
65
66             if(particles.size() != super.NUMAGENTS) p=new
                   DParticle(this);
67           }
68           }
69           Steppable decreaser = new Steppable()
70           {
71               @Override
72          public void step(SimState state) { trails.multiply(0.9); }
73           };
74           schedule.scheduleRepeating(Schedule.EPOCH,2,decreaser,1);
75      }
76      public static void main(String[] args)
77      {
78          doLoop(DParticles.class, args);
79          System.exit(0);
80      }
81      @Override
82      public DistributedField2D getField(){ return particles; }
83      @Override
84      public SimState getState(){ return this; }
85      @Override
86      public void addToField(RemotePositionedAgent<Int2D> rm,Int2D loc){
87        particles.setObjectLocation(rm, loc);
88      }
89      public boolean setPortrayalForObject(Object o){
90        if(p!=null){
91           p.setPortrayalForObject(o,
92           new sim.portrayal.simple.OvalPortrayal2D(Color.YELLOW) );
93           return true;
```

```
94              }
95      return false;
96       }
97
98  }
```

C.1.3 (D)Visualization

The visualization of the simulation is provided for MASON in the class `Tutorial3WithUI` (see Listing C.7), while for D-MASON it is in the class `DParticlesWithUI` (see Listing C.8). Both the classes are subclasses of the MASON class `GUIState`. In this case there are no significant differences between the two implementations, this is due to the fact that the visualization refers to the visualization of the status of corresponding LP, so the status of the others LPs is not required for the visualization.

Listing C.7: DParticles

```
1  public class Tutorial3WithUI extends GUIState {
2      ...
3      public static void main(String[] args) {
4      Tutorial3WithUi t = new Tutorial3WithUi();
5          t.createController();
6      }
7      public Tutorial3WithUI() {
8          super(new Tutorial3(System.currentTimeMillis()));
9      }
10     public Tutorial3WithUI(SimState state) {
11         super(state);
12     }
13     ...
14 }
```

Listing C.8: DParticlesWithUI

```
1  public class DParticlesWithUI extends GUIState {
2      ...
3      public static String name;
4      ...
5      public DParticlesWithUI(Object[] args) {
6          super(new DParticles(args));
7          name = String.valueOf(args[7]) + "" +
               (String.valueOf(args[8]));
8      }
9      public static String getName() {
10         return "Peer: <"+name+">";
11     }
12     ...
13 }
```

C.2 D-MASON usage on a HPC environment

D-MASON is a Java project and is deployed using Apache Maven. D-MASON is distributed under MIT license on a public repository [Repce]. After building the project, by Maven commands, it is possible to execute the D-MASON System Management, described in Section 2.4.3. Assuming that we want to perform a simulation on an HPC environment that supports Java. First of all the D-MASON Master has to be started, using the following command: `java -jar DMASON-X.X.jar -m master`. Then it is possible to start the workers. There are two way to run the workers.

1. *Running on each node the command*: `java -jar DMASON-X.X.jar -m worker -ip <ipactivemQ> -p <portActivemq> -ns M`, where M is the number of LPs that this worker may execute.
2. *Running the command*: `java -jar DMASON-X.X.jar -m worker -ip <ipactivemq> -p <portactivemq> -h <ipslave1 ipslave2 ... ipslaveN> -ns <M>`, that execute automatically the D-MASON worker on each slave node given in input to the command using the parameter $-h$.

Both the commands above assumes that the nodes of the HPC systems uses a SSH Key-Based Authentication[1].

Finally it is possible to execute the simulation using the system management (see Section 2.4.3). It enable to access the Master web control, from any web browser, where it is possible to submit new simulations. Simulations consist of Java `Jar` containers that comprise all classes and resources required to perform a D-MASON simulation.

[1] SSH Key-Based Authentication `https://cs.calvin.edu/courses/cs/374/MPI/ssh.html` accessed on February 28, 2017.

Bibliography

[Ack71] R. L. Ackoff. *Towards a System of Systems Concepts*. Institute of Mngmt Sciences, 1971 (cit. on p. 117).

[Adl+03] M. Adler, Y. Gong, and A. L. Rosenberg. "Optimal sharing of bags of tasks in heterogeneous clusters." In: *SPAA*. 2003 (cit. on pp. 121, 130).

[AK95] C.J. Alpert and A. B. Kahng. "Recent directions in netlist partitioning: A survey". In: *Integration: The VLSI Journal* (1995) (cit. on p. 173).

[Alb+13] E. Alba, G. Luque, and S. Nesmachnow. "Parallel metaheuristics: recent advances and new Parallel metaheuristics: recent advances and new trends". In: *International Transactions in Operational Research*. 2013 (cit. on p. 119).

[Amm+11] A. Ammeri, W. Hachicha, H. Chabchoub, and F. Masmoudi. "A comprehensive literature review of mono-objective simulation optimization methods". In: *Advances in Production Engineering & Management* (2011) (cit. on p. 117).

[And+14] G. Andrighetto, M. Campennì, R. Conte, and M. Paolucci. *On the Immergence of Norms: a Normative Agent Architecture*. 2014 (cit. on pp. 185, 192).

[And04] D.P. Anderson. "BOINC: A System for Public-Resource Computing and Storage". In: *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. 2004 (cit. on p. 24).

[Ant+15] A. Antelmi, G. Cordasco, C. Spagnuolo, and L. Vicidomini. "On Evaluating Graph Partitioning Algorithms for Distributed Agent Based Models on Networks". In: *3rd Workshop on Parallel and Distributed Agent-Based Simulations of Euro-Par 2015 conference.* 2015 (cit. on pp. 19, 54).

[Arm+14] T.G. Armstrong, J.M. Wozniak, M. Wilde, and I. T. Foster. "Compiler techniques for massively Compiler techniques for massively scalable implicit task parallelism". In: *SC*. 2014 (cit. on p. 132).

[Bad+13] D.A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. "Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings". In: 2013 (cit. on pp. 54, 174).

[Bak+06] M. Baker, B. Carpenter, and A. Shafi. "MPJ Express: towards thread safe Java HPC". In: *Cluster Computing, 2006 IEEE International Conference on*. 2006 (cit. on p. 61).

[Bak+99] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. “mpiJava: An object-oriented Java interface to MPI”. In: *Parallel and Distributed Processing*. 1999 (cit. on p. 61).

[Bal+03] G. C. Balan, C. Cioffi-Revilla, S. Luke, L. Panait, and S. Paus. “MASON: A Java Multi-Agent Simulation Library”. In: *Proceedings of the Agent 2003 Conference*. 2003 (cit. on p. 184).

[Bea96] D. M. Beazley. “SWIG: An easy to use tool for integrating scripting languages with C and C++”. In: *USENIX Tcl/Tk Workshop*. 1996 (cit. on p. 145).

[Beh] NetLogo BehaviorSpace. URL: http://ccl.northwestern.edu/netlogo/docs/behaviorspace.html (cit. on p. 119).

[Ben13] I. Benenson. “Agent-based models of geographical systems”. In: *International Journal of Geographical Information Science* (2013) (cit. on p. 184).

[BM05] J.P. Brans and B. Mareschal. *Multiple Criteria Decision Analysis: State of the Art Surveys*. Springer Science, 2005 (cit. on p. 118).

[Bon00] A.B. Bondi. “Characteristics of Scalability and Their Impact on Performance”. In: *Proceedings of the 2Nd International Workshop on Software and Performance*. 2000 (cit. on p. 4).

[BS15] C. Brunsdon and A. Singleton. *Geocomputation: a practical primer*. SAGE, 2015 (cit. on pp. 183, 184).

[Ca] C-JVM-Scripting: evaluate JVM scripting languages in C applications. URL: https://github.com/spagnuolocarmine/swift-lang-swift-t-jvm-engine (cit. on p. 101).

[Cam15] Regione Campania. *GeoPortale: Geographic Information System of Campania*. 2015. URL: http://sit.regione.campania.it/ (cit. on p. 184).

[Car+16a] M. Carillo, G. Cordasco, F. Serrapica, C. Spagnuolo, P. Szufel, and L. Vicidomini. “D-Mason on the Cloud: an Experience with Amazon Web Services”. In: *Proceedings of Euro-Par* (2016) (cit. on p. 19).

[Car+16b] M. Carillo, G. Cordasco, V. Scarano, F. Serrapica, C. Spagnuolo, and P. Szufel. “SOF: Zero Configuration Simulation Optimization Framework on the Cloud.” In: *Parallel, Distributed, and Network-Based Processing*. 2016 (cit. on p. 21).

[Car+99] B. Carpenter, G.C. Fox, S.H. Ko, and S. Lim. *mpijava 1.2: API Specification*. 1999 (cit. on p. 61).

[Cas+02] M. Castro, P. Druschel, A.M. Kermarrec, and A. Rowstron. “SCRIBE: A large-scale and decentralized application-level multicast infrastructure”. In: *IEEE Journal on Selected Areas in Communications (JSAC)* (2002) (cit. on p. 37).

[Cas06] L.N. De Castro. *Fundamentals of natural computing: basic concepts, algorithms, and applications*. 2006 (cit. on p. 123).

[CC95] R. Conte and C. Castelfranchi. *Cognitive and Social Action*. 1995 (cit. on p. 25).

[CH05] B. Calvez and G. Hutzler. “Parameter Space Exploration of Agent-Based Models”. In: *Knowledge-Based Intelligent Information and Engineering Systems*. 2005 (cit. on pp. 54, 116, 173).

[Cha10] The Economist Agents of Change. 2010. URL: http://www.economist.com/node/16636121 (cit. on p. 24).

[Che+08] D. Chen, G.K. Theodoropoulos, S.J. Turner, W. Cai, R. Minson, and Y. Zhang. “Large scale agent-based simulation on the grid”. In: *Future Gener. Comput. Syst.* (2008) (cit. on p. 27).

[CL10] C. Chen and L.H. Lee. *Stochastic Simulation Optimization: An Optimal Computing Budget Allocation*. World Scientific Publishing Co., Inc., 2010 (cit. on pp. 117, 118).

[CN11] N. Collier and M. North. “A Platform for Large-scale Agent-based Modeling”. In: *W. Dubitzky, K. Kurowski, and B. Schott, eds., Large-Scale Computing Techniques for Complex System Simulations, Wiley*. 2011 (cit. on pp. 26, 27, 29).

[CN12] N. Collier and M. North. “Parallel agent-based simulation with Repast for High Performance Computing”. In: *SIMULATION: Transactions of the Society for Modeling and Simulation International* (2012) (cit. on p. 143).

[CN15] N. Collier and M. North. *Repast Java Getting Started*. 2015 (cit. on p. 138).

[com] GIS ABM community. URL: http://www.gisagents.org (cit. on pp. 183, 192).

[Com] Swift/T High Performance Dataflow Computing. URL: http://swift-lang.org/Swift-T/ (cit. on pp. 20, 95, 99, 109).

[Com05] President’s Information Technology Advisory Committee. *Computational Science: Ensuring America’s Competitiveness*. 2005 (cit. on pp. 1, 3).

[Con99] R. Conte. “Social Intelligence Among Autonomous Agents”. In: *Comput. Math. Organ. Theory* (1999) (cit. on p. 25).

[Cor+11] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo. “A Framework for distributing Agent-based simulations”. In: *Ninth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar’2011)*. 2011 (cit. on p. 18).

[Cor+13a] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo. “Bringing together efficiency and effectiveness in distributed simulations: the experience with D-MASON.” In: SIMULATION: Transactions of The Society for Modeling and Simulation International, 2013 (cit. on p. 18).

[Cor+13b] G. Cordasco, R. De Chiara, F. Raia, V. Scarano, C. Spagnuolo, and L. Vicidomini. “Designing Computational Steering Facilities for Distributed Agent Based Simulations.” In: *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 2013 (cit. on p. 18).

[Cor+14a] G. Cordasco, A. Mancuso, F. Milone, and C. Spagnuolo. “Communication Strategies in Distributed Agent-Based Simulations: The Experience with D-Mason”. In: *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations (PADABS). Euro-Par 2013*. 2014 (cit. on pp. 18, 59, 83, 176, 187).

[Cor+14b] G. Cordasco, F. Milone, C. Spagnuolo, and L. Vicidomini. “Exploiting D-Mason on Parallel Platforms: A Novel Communication Strategy”. In: *Proceedings of the 2nd Workshop on Parallel and Distributed Agent-Based Simulations (PADABS). Euro-Par 2014*. 2014 (cit. on pp. 18, 58, 59, 176, 187).

[Cor+15] G. Cordasco, D. Malandrino, P. Palmieri, A. Petta, D. Pirozzi, V. Scarano, L. Serra, C. Spagnuolo, and L. Vicidomini. “An extensible architecture for an ecosystem of visualization web-components for Open Data”. In: *Maximising interoperability Workshop— core vocabularies, location-aware data and more*. 2015 (cit. on p. 22).

[Cor+16a] G. Cordasco, R. De Donato, D. Malandrino, P. Palmieri, A. Petta, D. Pirozzi, V. Scarano, L. Serra, C. Spagnuolo, and L. Vicidomini. “Data–Driven Discussions: A Social Platform for Open Data to discuss and visualize from heterogeneous data sources”. In: *Data for Policy 2016 International Conference "Frontiers of Data Science for Government: Ideas, Practices, and Projections", University of Cambridge* (2016) (cit. on p. 165).

[Cor+16b] G. Cordasco, C. Spagnuolo, and V. Scarano. “Toward the new version of D-MASON: Efficiency, Effectiveness and Correctness in Parallel and Distributed Agent- based Simulations”. In: *1st IEEE Workshop on Parallel and Distributed Processing for Computational Social Systems. IEEE International Parallel & Distributed Processing Symposium (IPDPS), Chicago Hyatt Regency, Chicago, Illinois USA.* 2016 (cit. on p. 17).

[Cor+17a] G. Cordasco, D. Malandrino, P. Palmieri, A. Petta, D. Pirozzi, V. Scarano, L. Serra, C. Spagnuolo, and L. Vicidomini. “A Scalable Data Web Visualization Architecture”. In: *Parallel, Distributed, and Network-Based Processing*. 2017 (cit. on p. 21).

[Cor+17b] G. Cordasco, C. Spagnuolo, and V. Scarano. “Work Partitioning on Parallel and Distributed Agent-Based Simulation”. In: 2017 (cit. on p. 19).

[Cos+11] B. Cosenza, G. Cordasco, R. De Chiara, and V. Scarano. “Distributed Load Balancing for Parallel Agent-Based Simulations”. In: *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. 2011 (cit. on pp. 30, 34, 187).

[CR10] C. Cioffi-Revilla. “A Methodology for Complex Social Simulations”. In: *Journal of Artificial Societies and Social Simulation* (2010) (cit. on p. 25).

[Cra+10] B. Craenen, G. Theodoropoulos, V. Suryanarayanan, V. Gaffney, P. Murgatroyd, and J. Haldon. “Medieval military logistics: a case for distributed agent-based simulation”. In: *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. 2010 (cit. on pp. 24, 29).

[Csi+12] K. Csillery, O. Francois, and M.G.B. Blum. “abc: an R package for approximate Bayesian computation (ABC)”. In: *Methods in Ecology and Evolution*. 2012 (cit. on p. 137).

[CW13] A.T. Crooks and S. Wise. “GIS and Agent-Based models for Humanitarian Assistance, Computers, Environment and Urban Systems”. In: (2013) (cit. on pp. 184, 192).

[D.P+84] J.H.Saltzer D.P., Reed, and D.D. Clark. “End-to-end arguments in system design”. In: *ACM Trans. Comput. Syst.* (1984) (cit. on p. 31).

[Dak] Dakota. URL: https://dakota.sandia.gov (cit. on p. 120).

[dat16] ISTAT Commuters 2011 dataset. 2016. URL: http://www.istat.it/it/archivio/139381 (cit. on p. 185).

[DG08] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM.* (2008) (cit. on p. 121).

[DK14] A. Deckert and R. Klein. “Simulation-based optimization of an agent-based simulation”. In: *NETNOMICS: Economic Research and Electronic Networking* (2014) (cit. on p. 116).

[DR01] P. Druschel and A. Rowstron. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In: *Proc. of the 18th IFIP/ACM Inter. Conference on Distributed Systems Platforms (Middleware ’01)*. 2001 (cit. on p. 37).

[Dur+16] F.R. Duro, J.G. Blas, F. Isaila, J.M. Wozniak, J. Carretero, and R. Ross. “Flexible data-aware scheduling for workflows over an in-memory object store”. In: *CCGrid*. 2016 (cit. on p. 98).

[ECJ] ECJ. URL: https://cs.gmu.edu/~eclab/projects/ecj/ (cit. on p. 119).

[ECoF09] Organisation for Economic Co-operation and Development (OECD) Global Science Forum. *Applications of Complexity Science for Public Policy: new tools for finding unanticipated consequences and unrealized opportunities*. 2009 (cit. on p. 24).

[EK10] D. Easley and J. Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, 2010 (cit. on p. 54).

[EK13] P. Evripidou and C. Kyriacou. “Data-Flow vs Control-Flow for Extreme Level Computing”. In: *2013 Data-Flow Execution Models for Extreme Scale Computing (DFM)* (2013) (cit. on p. 97).

[Eps+07] J.M. Epstein, S.A. Levin, and S.H. Strogatz, eds. *Generative Social Science: Studies in Agent-Based Computational Modeling*. Princeton University Press, 2007 (cit. on pp. 24, 25).

[ERAEB05] H. El-Rewini and M. Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing*. Wiley-Interscience, 2005 (cit. on pp. 4, 5).

[exp13] http://mpj express.org/performance.html. 2013. URL: {MPJExpressPerformance} (cit. on p. 61).

[F.+13] Jabot F., T. Faure, and N. Dumoulin. "EasyABC: performing efficient approximate Bayesian computation sampling schemes using R". In: *Methods in Ecology and Evolution 4*. 2013 (cit. on p. 137).

[for] ESRI format. URL: https://www.esri.com/library/whitepapers/pdfs/shapefile.pdf (cit. on p. 184).

[For+12] F.A. Fortin, F.M.D. Rainville, M.A. Gardner, M. Parizeau, and C. Gagn. "DEAP: Evolutionary Algorithms Made Easy". In: *Journal of Machine Learning Research*. 2012 (cit. on pp. 137, 142).

[Fos95] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on p. 24).

[fun16] Swift/T High Performance Dataflow Computing Defining leaf functions. 2016. URL: http://swift-lang.github.io/swift-t/guide.html (cit. on p. 138).

[Ghe+03] S. Ghemawat, H. Gobioff, and S. Leung. "The Google File System". In: *ACM Symposium on Operating Systems Principles* (2003) (cit. on p. 121).

[GIS16] GIS. 2016. URL: http://www.dartmouth.edu/~geog58/Documents/Geog58Syllabus.pdf (cit. on p. 183).

[GJ90] M.R Garey and D.S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990 (cit. on pp. 54, 174).

[GT05] N. Gilbert and K. Troitzsch. *Simulation for the social scientist*. 2005 (cit. on p. 183).

[GT07] W. Gropp and R. Thakur. "Thread-safety in an MPI implementation: Requirements and analysis". In: *Parallel Comput.* (2007) (cit. on p. 61).

[Gui16] Swift/T High Performance Dataflow Computing Developer Guide. 2016. URL: http://swift-lang.github.io/swift-t/guide.html (cit. on p. 109).

[Gul+11] L. Gulyás, A. Szabó, R. Legéndi, T. Máhr, R. Bocsi, and G. Kampis. "Tools for Large Scale (Distributed) Agent-Based Computational Experiments". In: *Proceedings of CSSSA*. 2011 (cit. on p. 119).

[Gup96] A. Gupta. *Graph partitioning based sparse matrix orderings for interior-point algorithms*. IBM Thomas J. Watson Research Division, 1996 (cit. on p. 173).

[Haf+11] M. Hafeez, S. Asghar, U.A. Malik, A. ur Rehman, and N. Riaz. "Survey of MPI Implementations". In: *Digital Information and Communication Technology and Its Applications*. Springer, 2011 (cit. on p. 61).

[He+10] D. He, L. Lee H., C. Chen, M. Fu, and S. Wasserkrug. "Simulation Optimization Using the Cross-entropy Method with Optimal Computing Budget Allocation". In: *ACM Trans. Model. Comput. Simul.* (2010) (cit. on pp. 20, 115).

[Hil90] M.D. Hill. “What is Scalability?” In: *SIGARCH Comput. Archit. News* (1990) (cit. on p. 4).

[Hul+15] J. Hullman, N. Diakopoulos, E. Momeni, and E. Adar. “Content, Context, and Critique: Commenting on a Data Visualization Blog”. In: *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. 2015 (cit. on p. 155).

[HX98] K. Hwang and Z. Xu. *Scalable parallel computing: technology, architecture, programming*. WCB/McGraw-Hill, 1998 (cit. on pp. 30, 33).

[Hyb+06] M. Hybinette, E. Kraemer, Y. Xiong, G. Matthews, and J. Ahmed. “SASSY: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure”. In: *Winter Simulation Conference'06*. 2006 (cit. on pp. 29, 48).

[I.+13] Boussad I., J. Lepagnot, and P. Siarry. “A survey on optimization metaheuristics”. In: *Information Information Sciences*. 2013 (cit. on p. 133).

[Ian+15] L. Iandoli, I. Quinto, A. De Liddo, and S. Buckingham Shum. “On online collaboration and construction of shared knowledge: Assessing mediation capability in computer supported argument visualization tools”. In: *Journal of the Association for Information Science and Technology* (2015) (cit. on p. 166).

[Jp] Groovy A multi-faceted language for the Java platform. URL: http://www.groovy-lang.org/ (cit. on p. 102).

[Kar+99] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. “Multilevel hypergraph partitioning: applications in VLSI domain”. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* (1999) (cit. on p. 173).

[KC14] M. Klein and G. Convertino. “An embarrassment of riches”. In: *Communications of the ACM* (2014) (cit. on p. 165).

[KK98] G. Karypis and V. Kumar. “Multilevel k-way Partitioning Scheme for Irregular Graphs”. In: *Journal of Parallel and Distributed Computing* (1998) (cit. on pp. 54, 175).

[KL70] B.W. Kernighan and S. Lin. “An Efficient Heuristic Procedure for Partitioning Graphs”. In: *The Bell Systems Technical Journal* (1970) (cit. on pp. 173, 174).

[Kle10] M. Klein. “Using metrics to enable large-scale deliberation”. In: *Collective intelligence in organizations: A workshop of the ACM Group 2010 Conference*. 2010 (cit. on p. 165).

[Kuh08] M. Kuhn. “Building Predictive Models in R Using the caret Package”. In: *Journal of Statistical Software*. 2008 (cit. on p. 137).

[Lan] The Clojure Programming Language. URL: https://clojure.org/ (cit. on p. 101).

[lan] The Scala programming language. URL: https://www.scala-lang.org/ (cit. on p. 102).

[Law07] A.M. Law. *Simulation modeling and analysis*. McGraw-Hill, 2007 (cit. on pp. 20, 115–117).

[Let+15] N. Lettieri, C. Spagnuolo, and L. Vicidomini. "Distributed Agent-based Simulation and GIS: An Experiment With the dynamics of Social Norm". In: *3rd Workshop on Parallel and Distributed Agent-Based Simulations of Euro-Par 2015 conference.* 2015 (cit. on p. 19).

[LM09] X. Liu and T. Murata. "Community Detection in Large-Scale Bipartite Networks". In: *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*. 2009 (cit. on p. 48).

[Lop+15] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E.Riviere. "Edge-centric Computing: Vision and Challenges". In: *SIGCOMM Comput. Commun. Rev.* (2015) (cit. on p. 149).

[LP+12] A. López-Paredes, B. Edmonds, and F. Klugl. "Editorial of the Special Issue: Agent Based Simulation of Complex Social Systems". In: *Simulation: Transactions of the Society for Modeling and Simulation International* (2012) (cit. on p. 24).

[LT00] B. Logan and G. Theodoropoulos. "The Distributed Simulation of Multi-Agent Systems". In: *Proceedings of the IEEE*. 2000 (cit. on p. 29).

[Luk+04] S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan. "MASON: A new multi-agent simulation toolkit". In: *Proceedings of the 2004 SwarmFest Workshop, Ann Arbor (Michigan), USA.* 2004 (cit. on pp. 26, 28, 30, 116).

[Luk+05] S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan. "MASON: A Multiagent Simulation Environment". In: *Simulation* (2005) (cit. on pp. 17, 28, 29, 48, 116, 119).

[Luk13] S. Luke. "Essentials of Metaheuristics". In: *Essentials of Metaheuristics (Second Edition)*. 2013 (cit. on p. 119).

[LW02] A. Liaw and M. Wiener. "Classification and Regression by randomForest". In: *R News 2*. 2002 (cit. on p. 137).

[Mal+12] N. Malleson, L. See, A. Evans, and A. Heppenstall. "Implementing comprehensive offender behaviour in a realistic agent-based model of burglary". In: *Simulation* (2012) (cit. on p. 31).

[Mal+16] D. Malandrino, I. Manno, P. Palmieri, A. Petta, D. Pirozzi, V. Scarano, L. Serra, C. Spagnuoloand L. Vicidomini, and G. Cordasco. "An Architecture for Social Sharing and Collaboration around Open Data Visualisation". In: *In Poster Proc. of the 19th ACM conference on Computer-Supported Cooperative Work and Social Computing*. 2016 (cit. on pp. 21, 153, 154).

[Mam+] A.R. Mamidala, R. Kumar, D. De, and D.K. Panda. *MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics* (cit. on p. 62).

[Mat08] B. Matthew. “Review of Software Platforms for Agent Based Models”. In: (2008) (cit. on pp. 24, 26, 28, 29).

[Men+08] D. Mengistu, P. Troger, L. Lundberg, and P. Davidsson. “Scalability in Distributed Multi-Agent Based Simulations: The JADE Case”. In: *Proc. Second Int. Conf. Future Generation Communication and Networking Symposia FGCNS ’08*. 2008 (cit. on p. 29).

[Mes97] Message. *MPI-2: Extensions to the Message-Passing Interface*. 1997 (cit. on p. 62).

[MG92] J. Misra and D. Gries. “A Constructive Proof of Vizing’s Theorem.” In: *Inf. Process. Lett.* (1992) (cit. on p. 65).

[Min+96] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. *The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations*. 1996 (cit. on pp. 29, 48).

[MN+13] N. T. Collier M.J. North, J. Ozik, E.R. Tatara, C.M. Macal, M. Bragen, and P. Sydelko. “Complex adaptive systems modeling with Repast Simphony”. In: *Complex Adaptive Systems Modeling*. 2013 (cit. on p. 118).

[MN05] C.M. Macal and M.J North. “Tutorial on Agent-based Modeling and Simulation”. In: *Proceedings of the 37th Conference on Winter Simulation*. 2005 (cit. on pp. 115, 116).

[MO15] K. Moreland and R. Oldfield. “Formal Metrics for Large-Scale Parallel Performance”. In: ed. by J.M. Kunkel and T. Ludwig. Springer International Publishing, 2015 (cit. on p. 13).

[MT09] N. Mustafee and J.E.S. Taylor. “Speeding up simulation applications using WinGrid”. In: *Concurr. Comput.: Pract. Exper.* (2009) (cit. on p. 24).

[Mus+09] N. Mustafee, S.J.E. Taylor, K. Katsaliaki, and S. Brailsford. “Facilitating the Analysis of a UK National Blood Service Supply Chain Using Distributed Simulation”. In: *Simulation* (2009) (cit. on pp. 24, 29).

[Naj+01] R. Najlis, M.A. Janssen, and D. C. Parkerx. “Software tools and communication issues”. In: *Proc. Agent-Based Models of Land-Use and Land-Cover Change Workshop*. 2001 (cit. on pp. 24, 28, 29).

[Nel10] B.L. Nelson. “Optimization via simulation over discrete decision variables”. In: *Tutorials in operations research* (2010) (cit. on p. 117).

[New06] M.E.J. Newman. “Modularity and community structure in networks”. In: *Proceedings of the National Academy of Sciences (PNAS)* (2006) (cit. on p. 173).

[Nor+07] M.J. North, T.R. Howe, N.T. Collier., and J.R. Vos. “Declarative Model Assembly Infrastructure for Verification and Validation”. In: *S. Takahashi, D.L. Sallach and J. Rouchier, eds., Advancing Social Simulation* (2007) (cit. on pp. 26, 29, 48, 116).

[OL82] M.H. Overmars and J. Leeuwen. “Dynamic Multi-dimensional Data Structures Based on Quad- and K–d Trees”. In: *Acta Inf.* (1982) (cit. on p. 34).

[opt] OptTek metaheuristic optimization. URL: http://www.opttek.com (cit. on p. 119).

[Ozi+14] J. Ozik, M. Wilde, N. Collier, and C.M. Macal. "Adaptive Simulation with Repast Simphony and Swift". In: *Euro-Par 2014: Parallel Processing Workshops*. 2014 (cit. on p. 118).

[Ozi+15] J. Ozik, N. T. Collier, and J.M. Wozniak. "Many Resident Task Computing in Support of Dynamic Ensemble Computations". In: *8th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers Proceedings*. 2015 (cit. on pp. 98, 133, 142).

[Ozi+16a] J. Ozik, N.T. Collier, J.M. Wozniak, and C. Spagnuolo. "From Desktop To Large-scale Model Exploration with Swift/T." In: *Winter Simulation Conference (WSC) 2016*. 2016 (cit. on p. 21).

[Ozi+16b] J. Ozik, N. Collier, J. Wozniak, C. Macal, C. Cockrell, M. Stack, and G. An. "High performance model exploration of mutation patterns in an agent-based model of colorectal cance". In: *Computational Approches For Cancer, Workshop of International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016 (cit. on p. 15).

[Par] ParadisEO. URL: http://paradiseo.gforge.inria.fr (cit. on p. 120).

[Par+11] J. A. Parejo, A. Ruiz-Cortsa, S. Lozano, and P. Fernandez. "Metaheuristic optimization frameworks: a survey and benchmarking". In: *Soft Computing*. 2011 (cit. on p. 119).

[PE08] Hazel R. Parry and Andrew J. Evans. "A comparative analysis of parallel processing and super-individual methods for improving the computational performance of a large individual-based model". In: *Ecological Modelling* (2008) (cit. on p. 31).

[PS09] D. Pawlaszczyk and S. Strassburger. "Scalability in distributed simulations of agent-based models". In: *Proc. Winter Simulation Conf. (WSC) the 2009*. 2009 (cit. on p. 29).

[Rah+13] F. Rahimian, A.H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. "JA-BE-JA: A Distributed Algorithm for Balanced Graph Partitioning". In: *7th International Conference on Self-Adaptive and SelfOrganizing Systems. IEEE*. 2013 (cit. on p. 175).

[Rai+06] F.L. Railsback, L. Lytinen Steven, and K.S. Jackson. "Agent-based Simulation Platforms: Review and Development Recommendations". In: *Simulation* (2006) (cit. on pp. 24, 28, 29).

[Ree79] T. M. H. Reenskaug. *Thing-Model-View-Editor – an Example from a planningsystem*. 1979 (cit. on p. 28).

[Repce] D-MASON Official GitHub Repository. Accessed October 2016. URL: https://github.com/isislab-unisa/dmason (cit. on pp. 81, 196, 203).

[Reu+13] R. Reuillon, M. Leclaire, and S. Rey-Coyrehourcq. “OpenMOLE, a Workflow Engine Specifically Tailored for the Distributed Exploration of Simulation Models”. In: *Future Gener. Comput. Syst.* (2013) (cit. on pp. 119, 130).

[Rey87] C. Reynolds. “Flocks, Herds and Schools: A Distributed Behavioral Model”. In: *SIGGRAPH Comput. Graph.* (1987) (cit. on pp. 66, 82).

[Ros08] A.L. Rosenberg. “Cellular ANTomata: Food-Finding and Maze-Threading”. In: *Proceedings of the 2008 37th International Conference on Parallel Processing*. 2008 (cit. on p. 24).

[Rou+14] A. Rousset, B. Herrmann, C. Lang, and L. Philippe. “A Survey on Parallel and Distributed Multi-Agent Systems”. In: *Euro-Par 2014: Parallel Processing Workshops*. 2014 (cit. on p. 30).

[Set12] B. Settles. “Active Learning”. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning*. 2012 (cit. on pp. 137, 145).

[Shv+10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. “The Hadoop Distributed File System”. In: *Proceedings of the 2010 IEEE 26th Symposium on MSST*. 2010 (cit. on p. 121).

[SS13] P. Sanders and C. Schulz. “Think Locally, Act Globally: Highly Balanced Graph Partitioning”. In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. 2013 (cit. on p. 175).

[ST96] D.A. Spielmat and S.H. TenG. “Spectral partitioning works: planar graphs and finite element meshes”. In: *Proceedings 37th Annual Symposium on Foundations of Computer Science*. 1996 (cit. on p. 175).

[Sto11] F.J. Stonedahl. “Genetic Algorithms for the Exploration of Parameter Spaces in Agent-based Models”. In: *P.h.D. Thesis*. 2011 (cit. on p. 119).

[Sul+10] K. Sullivan, M. Coletti, and S. Luke. *GeoMason: GeoSpatial support for MASON*. 2010 (cit. on p. 184).

[Tan+11] W. Tang, D.A. Bennett, and S.Wang. “A parallel agent-based model of land use opinions”. In: *Journal of Land Use Science* (2011) (cit. on p. 31).

[TS04] E. Tekin and I. Sabuncuoglu. “Simulation optimization: A comprehensive review on theory and applications”. In: *IIE Transactions* (2004) (cit. on pp. 20, 115, 117).

[Tur13] G. Turkington. *Hadoop Beginner's Guide*. Packt Publishing, 2013 (cit. on pp. 21, 117).

[TW04] S. Tisue and U. Wilensky. “NetLogo: A simple environment for modeling complexity”. In: *International Conference on Complex Systems*. 2004 (cit. on pp. 21, 26, 116, 117, 119, 184).

[Wei+00] T. WeiQin, Y. Hua, and Y. WenSheng. “PJMPI: pure Java implementation of MPI”. In: *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*. 2000 (cit. on p. 61).

[Wil+09] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. “Parallel scripting for applications at the petascale and beyond. Computer”. In: *Computer*. 2009 (cit. on p. 95).

[Wil97] U. Wilensky. *NetLogo Fire model*. 1997 (cit. on p. 130).

[Woz+13] J.M. Wozniak, T. Peterka, T. G. Armstrong, J. Dinan, E. L. Lusk, M. Wilde, and I.T. Foste. “Dataflow coordination of data-parallel tasks via MPI 3.0”. In: *Recent Advances in Message Passing Interface (EuroMPI)*. 2013 (cit. on p. 99).

[Woz+15] J.M. Wozniak, T.G. Armstrong, K.C. Maheshwari, D.S. Katz, and M. Wilde. “Interlanguage parallel scripting for distributed-memory scientific computing”. In: *WORKS at SC*. 2015 (cit. on p. 145).

[Zha+10] Y. Zhang, F. Mueller, X. Cui, and T. Potok. “Large-Scale Multi-Dimensional Document Clustering on GPU Clusters”. In: *IEEE International Parallel and Distributed Processing Symposium*. 2010 (cit. on p. 34).

[ZZ04] B. Zhou and S. Zhou. “Parallel simulation of group behaviors”. In: *WSC ’04: Proceedings of the 36th conference on Winter simulation*. 2004 (cit. on p. 34).

[A J16] A JavaScript library for image- and vector-tiled maps using SVG. 2016. URL: http://polymaps.org/ (cit. on p. 158).

[A V16] A Versatile and Expandable jQuery Plotting Plugin. 2016. URL: http://www.jqplot.com/ (cit. on p. 158).

[Ama16] Amazon EC2. 2016. URL: https://aws.amazon.com/ec2 (cit. on p. 80).

[Apa11] Apache ActiveMQ. 2011 (cit. on pp. 37, 59).

[Att16] Attractive JavaScript plotting for jQuery. 2016. URL: http://www.flotcharts.org/ (cit. on p. 158).

[Bos16] Bosonic a practical collection of everyday Web Components. 2016. URL: http://bosonic.github.io/ (cit. on p. 157).

[Cro16] Crossfilter a JavaScript library for exploring large multivariate datasets in the browser. 2016. URL: http://square.github.io/crossfilter/ (cit. on p. 158).

[D3.16] D3.js a JavaScript library for manipulating documents based on data. 2016. URL: https://d3js.org/ (cit. on p. 158).

[Dim16] Dimple an object-oriented API for business analytics powered by D3.js. 2016. URL: http://dimplejs.org/ (cit. on p. 159).

[Dyg16] Dygraphs a fast, flexible open source JavaScript charting library. 2016. URL: http://dygraphs.com/ (cit. on p. 157).

[Goo16a] Google Chart. 2016. URL: https://developers.google.com/chart/ (cit. on pp. 155, 158).

[Goo16b] Google Inc. Polymer Library. 2016. URL: https://www.polymer-project.org/1.0/ (cit. on pp. 71, 157, 159).

[Goo16c] Google Material Design. 2016. URL: https://www.google.com/design/spec/material-design (cit. on p. 71).

[Grace] GraphChi's Java version. Accessed May 2015. URL: https://github.com/GraphChi/graphchi-java (cit. on p. 177).

[Hig16] HighCharts JavaScript Visualization Library. 2016. URL: http://www.highcharts.com/ (cit. on pp. 154, 158, 160).

[Ja-ce] Ja-be-Ja GitHub repository. Accessed May 2015. URL: https://github.com/fatemehr/jabeja (cit. on p. 177).

[Jav] JavaScript the programming language of HTML and the Web. URL: http://www.w3schools.com/js/ (cit. on p. 102).

[Jet13] Jetty. 2013. URL: http://www.eclipse.org/jetty/ (cit. on p. 71).

[Lea16] Leaflet open-source JavaScript library interactive maps. 2016. URL: http://leafletjs.com/ (cit. on pp. 154, 157).

[MAS16] MASON Particles simulation. 2016. URL: https://github.com/eclab/mason/tree/master/mason/sim/app/tutorial3 (cit. on p. 195).

[MPI13] MPI Standard Official Website. 2013. URL: http://www.mcs.anl.gov/research/projects/mpi/index.htm (cit. on p. 61).

[MPI16] MPI Forum. 2016. URL: http://www.mpi-forum.org/docs/docs.html (cit. on p. 59).

[Ope13] Open MPI Official Website. 2013. URL: http://www.open-mpi.org/ (cit. on pp. 61, 64).

[Ope16a] Open Data in a Nutshell. 2016. URL: www.europeandataportal.eu (cit. on p. 153).

[Ope16b] Open Data Software. 2016. URL: https://www.opendatasoft.com/ (cit. on pp. 153, 166).

[Ope16c] Open MPI: Open Source High Performance Computing. 2016. URL: http://www.open-mpi.org/ (cit. on p. 59).

[Ope16d] Open Source Data Portal Sofware. 2016. URL: http://ckan.org/ (cit. on pp. 153, 166).

[Ope16e] OpenLayers a high-performance, feature-packed library for all your mapping needs. 2016. URL: https://openlayers.org/ (cit. on p. 159).

[Oxw16] Oxwall is an Open Source Mobile-frindly community website platorm. 2016. URL: https://www.oxwall.com/ (cit. on p. 166).

[Rap16] Raphaël a small JavaScript library that should simplify your work with vector graphics on the web. 2016. URL: http://dmitrybaranovskiy.github.io/raphael/ (cit. on p. 158).

[Sta16] StarCluster. 2016. URL: http://star.mit.edu/cluster/index.html (cit. on p. 80).

[Thece] The Graph Partitioning Archive. Accessed May 2015. URL: http://staffweb.cms.gre.ac.uk/~wc06/partition/ (cit. on p. 175).

[Web16] Webcomponent.js polyfills enable Web Components in (evergreen) browsers that lack native support. 2016. URL: http://webcomponents.org/polyfills/ (cit. on p. 156).

[Wor16] World Wide Web Consortium. 2016. URL: https://www.w3.org/standards/techs/components (cit. on pp. 154, 156, 159).

[X-T16] X-Tag is a Microsoft supported, open source, JavaScript library that wraps the W3C standard Web Components. 2016. URL: http://x-tag.github.io/ (cit. on p. 156).

[cgl16] cglib. 2016. URL: https://github.com/cglib/cglib (cit. on pp. 49, 51).

[jQu16] jQuery Visualize. HTML5 canvas charts driven by HTML table elements. 2016. URL: https://github.com/filamentgroup/jQuery-Visualize (cit. on p. 159).

List of Figures

List of Tables