Università degli Studi di Salerno

Dottorato di Ricerca in Informatica e Ingegneria dell'Informazione
Ciclo 33 – a.a 2020/2021

TESI DI DOTTORATO / PH.D. THESIS

Empowering Computational Science through Extreme Scalability

MATTEO D'AURIA

SUPERVISOR: **PROF. VITTORIO SCARANO**

PHD PROGRAM DIRECTOR: **PROF. PASQUALE CHIACCHIO**

Dipartimento di Ingegneria dell'Informazione ed Elettrica
e Matematica Applicata
Dipartimento di Informatica

Abstract

Computational science is an ever-expanding research field. It combines technologies, modern computational methods, and simulations to address problems too complex to be effectively predicted by theory alone or too expensive or dangerous to be reproduced in the laboratory. This scientific domain is a multidisciplinary field and impacts several sciences, engineering, and humanities problems. The success of the computational science approach has resulted in an increasing demand for computing resources to improve the performance of solutions and enable the growth of models, both in size and quality. For these reasons, parallel and distributed computing paradigms and exploiting Cloud Computing have become essential in computational scientists' everyday lives. Cloud provides a huge amount of computational power, easily accessible to everyone in a price-aware manner. As a result, the notion of "scalability" of an application, running over parallel or distributed systems, has become central in computational science. In a nutshell, an application is scalable if it can efficiently exploit an increasing amount of computational power (e.g., number of nodes or processors). In this domain, a relevant research challenge is to provide scalability at different levels, from software libraries to frameworks and tools for helping the solution of scientific problems. Providing scalability permits scientists to face massive and complex problems in a transparent and easy as possible way. This dissertation discusses frameworks and parallel languages that allow scientists to approach computational science problems under the lens of the extreme scalability requirement. Contributions of this work can be summarized in three principal categories: languages and tools, Agent-based Model (ABM) and simulation, and Optimization via Simulation (OvS).

Many real-world scientific applications consist of orchestrating several different independent tasks or methods for accomplishing a particular workload. These workflows are usually computationally and time demanding, thus exploiting parallel and distributed techniques became essential. This dissertation presents FLY, a domain-specific language for scientific applications, which aims at reconciling Cloud and High-Performance Computing paradigms. FLY adopts a multi-cloud approach by providing a powerful, effective, and pricing-efficient tool for developing scalable workflow-based scientific applications. FLY exploits different and at the same time Function-as-a-Service cloud providers as computational backends in a transparent manner. This dissertation describes the programming model of FLY, its language definition, and the FLY source-to-source compiler. Furthermore, it is discussed a performance evaluation of FLY on a popular benchmark for distributed computing frameworks, along with a collection of case studies with an analysis of their performance results and costs. Finally, a real use case scenario is shown that implements an Optimization via Simulation process using FLY for carrying out a distributed evaluation of simulations on an AWS backend.

ABM is a bottom-up modeling approach, where independent decision-making agents model a complex system. Large-scale emergent behavior in ABMs is affected by the dimension of the population and the complexity of each agent's behavior. However, the computational cost of the simulation grows together with the increasing of model details. This dissertation presents the architecture and the implementation of an open-source library for developing Agent-Based Models using the Rust language. Rust-AB is able to exploit both sequential and parallel computing platforms. An investigation on the ability of Rust to develop ABM simulations are discussed, as well as several models developed with Rust-AB are described. Finally, a performance comparison against the well-known Java ABM toolkit MASON is also presented.

OvS refers to techniques for discovering the parameters of a complex model by optimizing one or more objective functions, which can only be computed by running a simulation. Due to the high dimensionality of the search space, the heterogeneity of the parameters, and the stochastic nature of the objective evaluation function, optimizing

such a simulation is extremely computational demanding. This dissertation discusses methods for exploiting parallel/distributed systems' computational power to improve the efficiency and effectiveness of OvS strategies. Specifically, three frameworks are presented that differ for their underlying computing system architecture adopted: i) heterogeneous – where CPU and GPU are used to execute simulations in a distributed system composed of a heterogeneous node in terms of hardware and software, ii) homogeneous – the computing system is composed of homogeneous nodes where are used MASON (simulation library) and ECJ (optimization library) software for elaborate the OvS process, and iii) cloud computing – the computing system is a MapReduce cluster running over the cloud.

Contents

Chapter 1

Introduction

1.1 Computational Science

Over the past decades, the scientific methodological approach has changed. In science, physics, social sciences, biomedical, engineering research, defense, national security, and especially in industrial innovation, problems are increasingly approached from a computational perspective. Computational science [1], also known as Scientific Computing (SC), is an established and growing field that uses advanced computing and data analysis to study complex real-world problems. SC aims to address problems using the predictive capability to support traditional experimentation and theory using a computational approach to problem-solving. This discipline combines computational thinking, modern computational methods, hardware, and software to address problems, overcoming the limitations of traditional methods. Advances in SC allow R&D scientists and engineers to tackle problems that are too complex to be reliably predicted from a theoretical standpoint and/or too dangerous or expensive to reproduce in the laboratory. The Figure 1.1 shows the definition of SC by dividing it into three main 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. *Computing infrastructure* that supports both the science and engineering problem solving and the developmental computer and information science.

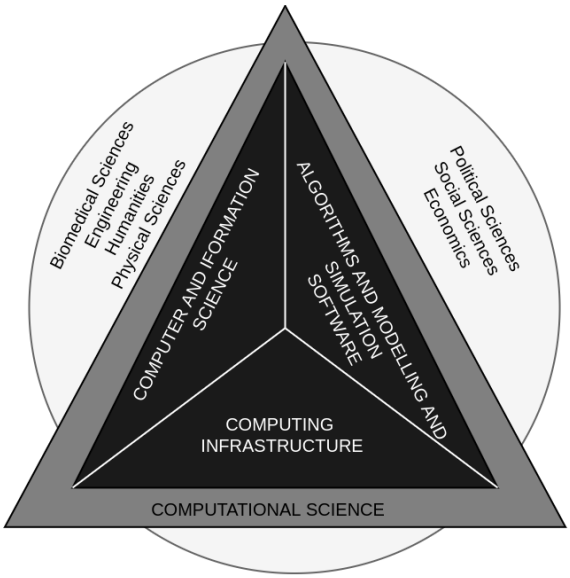

Figure 1.1: Computational Science areas.

A fundamental aspect of Computational Science is that it allows the analysis and study of complex real systems through mathematical models and simulations. A class of models particularly interesting takes the name of Agent-based Models (ABM). These models are useful in the simulation of actions and interactions of autonomous agents (both individual or collective entities such as organizations or groups) with a view to assess their effects on the system as a whole. An ABM consists of three components: agents, relations, and rules. The agents model a population; the relations define potential interactions among agents; the rules describe the behaviour of an agent as a result of an interaction.

The enormous success obtained by these models led to the realization of several tools for agent-based simulations, but the only requirement that never changed was the need 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.

The rapid growth of ABM models has led to a growing need for computational resources. As a result, parallelization techniques and computation distribution on HPC or Cloud systems had to be exploited. Further increasing the need for additional computational power was the need to optimize these ABM simulations. One feature of simulation is that one can easily change the parameters of a simulation model and observe the system performance under different sets of parameters. Therefore, it is natural to find the set of parameters that optimizes the system performance to produce results as close to reality. The Optimization via Simulation (OvS) process allows optimizing an objective function (which maps the real behavior of the complex system) by executing a large number of simulations. The OvS was immediately shown to be a non-trivial process that required both a high number of computational resources and high execution time. From the Computer Science point of view, in the SC field, the challenge is to improve the current status of methods, algorithms, and applications to enhance support for SC in terms of efficiency and effectiveness of the solutions.

As described in [1], the most important goal that should guide our research is *Scalability*. Our solutions, software, models, algorithms, systems should be scalable according to the problem itself. This requirement is not only at the level of software development of architectures and frameworks but also includes design solutions for a problem in the SC domain. Scalability is a frequently-claimed attribute of a system or solution to face complex problems using computer systems. Despite the centrality of Scalability in software development, its definition is not generally accepted, as described in [2]. Nevertheless, we can use

some of its principles to design scalable solutions for the SC field better. According to this idea, it is possible to state that Computational Science should be scalable at different software and hardware levels.

From the point of view of computational scientists, the realization of correct and high-performance applications is very complicated and challenging [3]. This is also emphasized in [4], a survey on the practice of SC at Princeton University. The survey reveals the difficulty that computational scientists have in interfacing with combinations of numerical, general-purpose, and scripting languages. The survey also reveals the difficulty of computational scientists in developing high-performance applications due to the absence of specific skills in parallel and distributed programming. Finally, the survey also shows the need for high-performance tools to optimize models to improve the quality of the solutions.

The purpose of all the contributions presented in this dissertation is to simplify the work of computational scientists by providing languages and tools that allow them to write effective and efficient applications taking full advantage of the scalability offered by modern computing techniques. These contributions fall into several areas of SC:

1. *Scalable scientific workflow*. The development of a Domain-Specific language for exploiting the computational power of Multi-Cloud Computing systems.

2. *Efficient Parallel Agent-based Simulation*. Software solutions to develop massive Agent-Based Simulations.

3. *Scalable Optimization via Simulation*. Software solutions to develop distributed OvS processes on High-Performance Computing (HPC) as well as cloud infrastructures.

A detailed description of the contributions of this thesis will be provided in Section 1.4.

To better understand the rest of the thesis, the next section introduces the concept of Scalability, some basics of parallel and distributed computing until we get to answer the question: "*How to measure the scalability of an application?*"

1.2 Scalability

The scalability requirement measures the capability of the system to react to the computational resources used. Scalability can be seen in several forms: speed, efficiency, high reliable applications, and heterogeneity [5]. 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 to solve a problem. In the following, we denote a system with a large number of nodes and dedicated hardware architecture as a supercomputing system. The availability of supercomputing systems has become much affordable day by day. Also, Cloud Computing systems, which offer many high-performance resources at a low-cost, are an attractive opportunity in the SC field. As described in [6] 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 moderate or large. For instance, a search engine may use a hash table or balanced tree data structure to index pages 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 North American's telephone numbering scheme, 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 telephones numbers.

The Scalability of a system can be analyzed according to different properties:

- speed: an increasing 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.

To measure how efficient an application is, we can analyze how the application behaves when using increasing numbers of computational resources (cores, processors, threads, machines, CPUs, etc.). Before discussing how to measure a parallel or distributed application's efficiency, it is necessary to briefly discuss parallel and distributed background.

1.2.1 Parallel and distributed background

An accepted classification of the state of the art of computing divides the history of computing into four eras: batch, time-sharing, desktop, and network [7]. Today, 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 system.

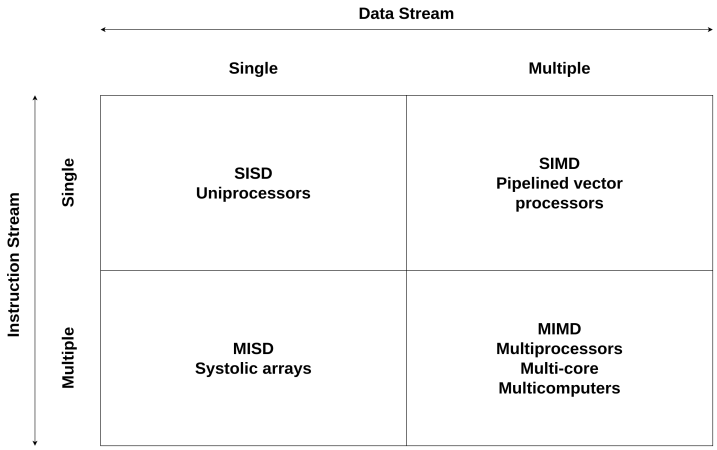

Figure 1.2: Flynn's Taxonomy

To understand parallel computing, the first concept that is essential to know is the classification of computer architectures. An important and well know classification scheme is the Flynn taxonomy. Figure 1.2 shows the taxonomy defined by Flynn in 1966. The classification relies on the notion of a stream of information. A processor unit could accept two types 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:

- SISD, Single-Instruction Single-Data streams: operations are performed sequentially. This is the classic von Neumann architecture;
- SIMD Single-Instruction Multiple-Data streams: architectures composed of many processing units that simultaneously execute the same instruction but work on different data sets. SIMD systems are primarily used to support specialized parallel computing. The most famous examples is the vector supercomputers, used for particular applications (where mainly work on large matrices);
- MISD Multiple-Instruction Single-Data streams: multiple instruc-

tion streams (processes) work simultaneously on a single data stream;

- MIMD Multiple-Instruction Multiple-Data streams: multiple instructions are executed simultaneously on several different data items. Computer clusters fall under this classification.

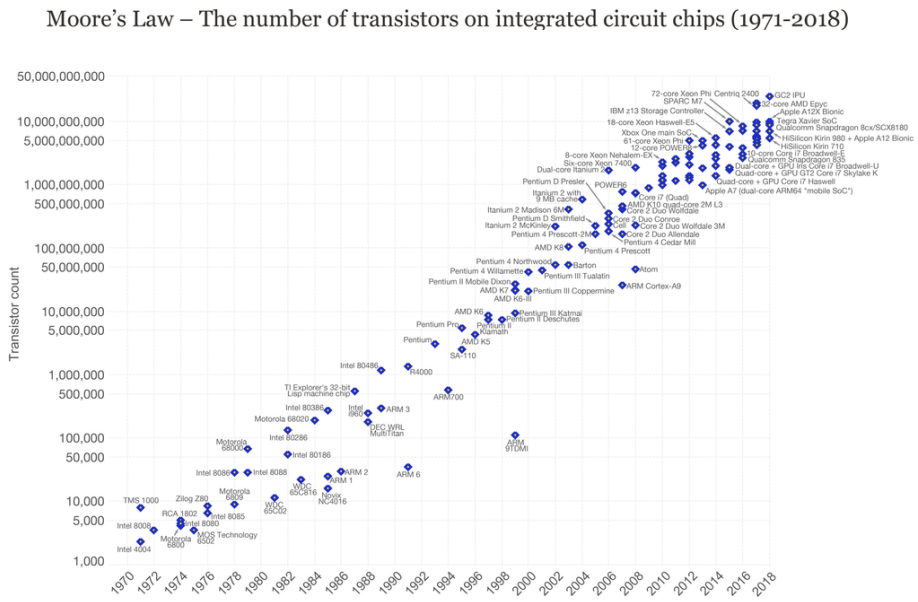

Figure 1.3: The number of transistors on integrated circuit chips from 1971 to 2018

Moore's law. Historically, the way to achieve better computing performances has depended on hardware advances. Now, this trend is permanently changed, and parallel and distributed computing is the only way to achieve better performance. The Figure 1.3 depicts Moore's Law, 1965, which clearly explains this idea. Moore's Law shows that the count of CPU and RAM transistor doubled each year. Nonetheless, due to physical limits and heat emission, this trend has ended around

2008, stabilizing the speed processors. The actual trend of CPU vendors is to increase the number of cores to have better performance. In other words, this means that each science has to face problems exploiting parallel and distributed computing to increment the complexity of the problems or improve the performances.

Parallel computing architecture

This section describes different approaches to do parallel and distributed computing, depending on system architectures. Parallel computing architectures are historically categorized into 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:

- Symmetric multiprocessing (SMP), In this model, multiple processors use the same memory. This is the easiest 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 limit of this architecture is the bus bandwidth that may represent a performance bottleneck.
- Multi-core. This is the model adopted by modern processors that employ multiple cores in a single processor. This architecture allows using a single processor as an SMP machine.
- Multi-computers. This architecture is basically the distributed computing architecture. The computers are connected across a network, and the single processor can access only its local memory space while interactions 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.

Computational models

This section describes two theoretical computational models that are independent on the parallel computing architectures used. These mod-

els aim to measure the quality of a solution using a parallel computing approach. Starting from these two models, it is also possible to describe the concept of scalability for the parallel computing architectures described before.

Equal duration model. These models 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 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 ($\mathrm{S}(n)$) of a parallel system, as the ratio between the time to execute the whole computation on a single processor (t_s) and the time obtained exploiting n processors unit (t_m).

$$S(n) = \frac{t_s}{t_m} = \frac{t_s}{\frac{t_s}{n}} = n \tag{1.1}$$

The previous definition is not enough to describe the speed obtained. One needs 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 becomes:

$$S(n) = \frac{t_s}{t_m} = \frac{t_s}{\frac{t_s}{n} + t_c} = \frac{n}{1 + n\left(\frac{t_c}{t_s}\right)} \tag{1.2}$$

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

$$\xi = \frac{S(n)}{n} = \frac{1}{1 + n\left(\frac{t_c}{t_s}\right)} \tag{1.3}$$

The value of the efficiency ranges between 0 and 1. Again, this model is not realistic because it assumes that a given task can be divided, in "equal" sub-tasks, among n processors. On the other hand,

real algorithms contain some serial parts that cannot be divided among processors. Furthermore, a parallel algorithm is also characterized by some sub-tasks that cannot be executed concurrently by processors. These sub-tasks includes synchronization or other special instructions and are named critical sections. 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 a fraction f of the given task is not dividable into concurrent sub-tasks and the remaining fraction $1 - f$ is assumed to be dividable in concurrent sub-task. Like the previous model, the total time to execute the computation on n processors is $t_m = f \times t_s + (1 - f) \times \left(\frac{t_s}{n}\right)$. In this case, the speedup becomes:

$$S(n) = \frac{t_s}{ft_s + (1 - f)\left(\frac{t_s}{n}\right)} = \frac{n}{1 + (n - 1)f} \tag{1.4}$$

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

$$S(n) = \frac{t_s}{ft_s + (1 - f)\left(\frac{t_s}{n}\right) + t_c} = \frac{n}{f(n - 1) + 1 + n\left(\frac{t_c}{t_s}\right)} \tag{1.5}$$

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

$$\lim_{n \to \text{inf}} S(n) = \lim_{n \to \text{inf}} \frac{n}{f(n - 1) + 1 + n\left(\frac{t_c}{t_s}\right)} = \frac{1}{f + \left(\frac{t_c}{t_s}\right)} \tag{1.6}$$

According to equation 1.6, it is worth noticing 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

$$\xi = \frac{1}{1 + (n - 1)f} \tag{1.7}$$

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

$$\xi = \frac{1}{f + \left(\frac{t_c}{t_s}\right)} \tag{1.8}$$

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

Parallel computation laws

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

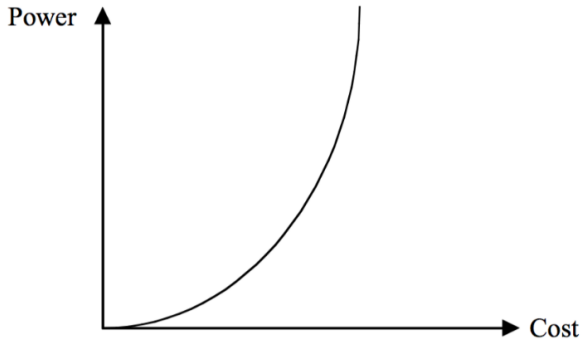

Figure 1.4: Power-cost relationtship according to Grosch's Law.

Grosch's law. H. Grosch, in the 1940s, postulated that the power of a computer system P increases in proportion to its cost C, $P = K \times C^s$ where s and K are positive constant. Grosch postulated further that the value of s would be close to 2. The Figure 1.4 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 most HPC and heterogeneous distributed systems.

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 1.4, 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. Amdahl's principle states that the maximum speedup factor is given by:

$$\lim_{n \to \text{inf}} S(n) = \frac{1}{f} \tag{1.9}$$

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

$$\lim_{n \to \text{inf}} S(n) = \lim_{n \to \text{inf}} \frac{n}{1 + (n-1) \times f(n)} = n \tag{1.10}$$

This contradicts Amdahl's law considering that it 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.

1.2.2 Measuring parallel performance

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

$$SS = \frac{t_s}{(n \times t_m)} \times 100 \tag{1.11}$$

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 resource consumption of an application. In 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:

$$WS = (\frac{t_1}{t_n}) \times 100 \tag{1.12}$$

1.3 The State of Art of Scalability in Computational Science

The SC field has been massively studied in the last decades to find more and more effective and efficient solutions in solving complex real-world problems by exploiting the concept of high-performance computing. In the following, a state-of-art of parallel languages and scalable tools in the field of SC is shown.

1.3.1 Parallel language for computational science

Parallel and distributed languages have been actively investigated for decades. The results are commonly concentrated on parallel machines, high-performance systems, and distributed computing infrastructures, as described in [8]. In the following, several languages and frameworks that are suitable for developing scalable applications in the SC research area were presented:

- *Fortran* is a programming language designed for numeric computation and scientific computing. It is widely used in scientific fields (such as numerical weather prediction, computational fluid dynamics, and computational physics) and quite popular for high-performance computing applications.

- *R* [9], and its libraries implement a wide variety of statistical and graphical techniques, including linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, and others.

- *Python* [10] is an interpreted high-level programming languages for general-purpose programming. Python features a dynamic type system and automatic memory management. Python is most used in SC thanks to the following libraries: *SciPy*, a collection of mathematical algorithms and convenience functions built on the Numpy extension of Python; *scikit-learn* for machine learning; *scikit-image* for image manipulation; *Pandas* for DataFrames.

- *Julia* [11] is a high-level, high-performance dynamic programming language for numerical computing. Julia provides a sophisticated compiler as well as distributed parallel execution, numerical accuracy, and an extensive mathematical function library.

- *Limbo* [12] is a programming language intended for applications running distributed systems on small computers. Limbo supports modular programming, concurrent programming, strong type-checking at compile- and run-time, inter-process communication over typed channels, automated garbage collection, and simple data type.

- *Chapel* [13] is a portable programming language designed for productive parallel computing on large-scale systems. Its design and implementation have been undertaken with portability in mind, enabling Chapel to run on different environments like multicore desktops and laptops as well as commodity clusters,

and the cloud, in addition to the high-end supercomputers for which it was designed.

- *Cilk* [14] is a C/C++ extension designed for multithreaded parallel computing. Cilk is a C/C++ extension that supports nested data and task parallelism. The principle behind the design of the Cilk is that the programmer is responsible for exposing the parallelism while the run-time environment decides how to actually divide the work between processors.

- *Swift* [15] is a featured data-flow oriented coarse-grained scripting language, which is designed for scientists, engineers, and statisticians that need to execute domain-specific application programs many times on large collections of file-based data. Swift is written on top of *Java Virtual Machine* in such a way that each Swift program (or workflow) is automatically parallelized at compilation time and executed concurrently. Swift is primarily used to manage calls of external functions written in C, C++, Fortran, Python, R, Tcl, Julia, Qt Script, or executable programs. *Swift/T* [16] is the high-performance computing version of Swift languages, in which the Swift programs are translated into MPI based programs to be executed on HPC systems. Swift and Swift/T provide set-up on cloud IaaS[1].

- *OpenMole* [17] offers tools to run, explore, diagnose and optimize numerical models, taking advantage of distributed computing environments. OpenMOLE comes with a graphical user interface (GUI) to write scripts around your model. These scripts will use OpenMOLE methods to explore your model and distribute its executions on High Performing Computing (HPC) environments, with only a few lines of code. OpenMOLE built-in methods are designed to explore your model and answer several questions regarding optimization, sensitivity, robustness, and diversity of your model capabilities. Multiple environments are available to delegate your workload. The Multi-thread permits you to execute the tasks concurrently on your machine, the SSH one to execute

[1] swift-lang.org/tutorials/cloud/tutorial.html

tasks on a remote server through SSH. You can also access a wide variety of clusters like PBS/Torque, SGE, Slurm, Condor, or OAR. You can also use EGI to execute tasks on the EGI grid.

1.3.2 Agent-based simulation

In the following are presented several existing tools - commonly designed either as a framework or a library (or both) - for developing and running ABM simulations with a particular emphasis on their peculiarities, as described in [18]. According to the underlying architecture, ABM software tools can be easily classified into two categories: software for sequential computing architectures and software for distributed computing architectures.

ABM tool	Source Language	Applications Language	Computing Platform	Application Domain	License
SWARM [19]	Java, Objective-C	Objective-C, Swarm code, Java	Personal computer, Workstation, Large-scale scientific computing clusters and HP supercomputers	Simulation of complex adaptive systems in social or biological sciences	Open source, GPL, Free
StarLogo [20]	Java / YoYo	StarLogo scripting	Desktop computer	Simulation in social and natural sciences, education	Closed source, Clearthought Software License version 1.0, Free
NetLogo [21]	Scala	NetLogo language	Desktop computer	2D/3D simulation in social and natural science, teaching/research	Open source, GPL, Free
REPAST [22]	Java / C#	Java; C#, C++, Lisp, Prolog Visual Basic.Net, Python scripting	Desktop and vast-scale distributed computing clusters	Simulation of social networks and integrates support for GIS, genetic algorithms	Open source, BSD, Free
MASON [23]	Java	Java	Desktop computer, Workstation	General multi-purpose 2D/3D simulation	Open source, Academic Free License version 3.0
FLAME [24]	C	Graphical user interface, visualiser and validation tools	Laptop, Workstation, HPC supercomputers	General multi-pupose simulation	Open source, GNU Lesser General Public License, Free
REPAST-HPC [25]	C++ with MPI	Standard or Logo-style C++	Large-scale distributed clusters and HP supercomputer	Simulations in computational social sciences, cellular automata, complex adaptive system	Open source, BSD, Free
D-MASON [26]	Java	Java	Desktop computer, Workstation, Clusters, Cloud architectures	General multi-purpose 2D/3D simulation	Open souce, Apache License version 2.0
FLAME-GPU [27]	C for CUDA OpenGL	C-based scripting and optimized CUDA code	Laptop, Workstation, HPC	3D simulation for emergent complex behaviours in biology/medical domains with multi-massive amount of agent on GPU	Open source, FLAME GPU Licens Agreement, Free

Table 1.1: ABM frameworks/software comparison.

Table 1.1 describes the most important frameworks and libraries for ABM simulations according to the simulation engine programming

language, user programming language, computing platform, application domain, and release license. The first six rows of Table 1.1 summarise as many frameworks suitable for sequential computing simulations and that are fully described by [18].

- *SWARM* [19] is a multi-agent software platform for the simulation of complex adaptive systems. In the SWARM system, the basic unit of simulation is the *swarm*, a collection of agents executing a schedule of actions. SWARM provides object-oriented libraries of reusable components for building models and analyzing, displaying, and controlling experiments on those models.

- *StarLogo* [20] is an agent-based simulation language. It is an extension of the Logo programming language [28] designed for education. StarLogo Nova is the latest version, and this version is under development by the MIT Scheller Teacher Education Program.

- *NetLogo* [21] is a multi-agent programmable modeling environment. NetLogo is designed for education, and it is used by students, teachers, and researchers worldwide. NetLogo is built on top of Logo programming language [28]. NetLogo offers an extension, called BehaviorSpace, that can exploit the parallelism of the machine by running more simulations at the same time with different parameter settings.

- *Repast* [22] is a free and open-source agent-based modeling toolkit. Repast offers an extremely flexible hierarchically nested definition of space, including the ability to do point-and-click and modeling and visualization of 2D environments, 3D environments, networks including full integration with the JUNG network modeling library, and geographical spaces including full Geographical Information Systems (GIS) support. Repast offers a concurrent multi-threaded discrete event scheduler.

- *MASON* [23] is an open-source agent-based modelling simulation toolkit. MASON is designed to be general-purpose, to be efficient, and flexible. MASON models are fully separated from

visualization and are entirely serializable to disk. MASON models are duplicable (run several instances of the simulation with exactly the same parameters produce exactly the same results, even on different machines). MASON includes a high-quality random number generator and supports GIS simulation.

- *FLAME* [24] is a framework to create agent-based models that can be run on high-performance computers (HPCs). The philosophy of FLAME is to specify an agent-based model as you would specify software behavior, as ultimately, the execution of the model will be in software. The behavior model is based upon state machines, which are composed of a number of states with transition functions between those states. There is a single start state, and by traversing states using the transition functions, the machine executes the functions until it reaches an end state.

Several frameworks are devoted to developing large-scale simulations and provide good performance exploiting parallel/distributed computing architectures. The bottom part of Table 1.1 shows three frameworks for developing ABM simulations in distributed and parallel computing architecture.

- *Repast HPC* [25] is the HPC version of Repast that uses schedulers on each independent processes. It is written in cross-platform C++ and can be used on workstations, clusters, and supercomputers. The execution model of Repast HPC requires a complex cluster configuration that provides the MPI ecosystem.

- *DMASON* [26] is a distributed implementation of MASON designed to run over clusters and/or cloud-computing architectures. DMASON provides facilities to execute simulations in distributed environments. However, it introduces complexity in the simulation execution.

- *FLAME GPU* [27] is a high-performance Graphics Processing Unit (GPU) extension to the FLAME framework. It provides a mapping between a formal agent specification with C-based scripting and optimized CUDA code. FLAME GPU requires to

be executed a powerful Nvidia GPU with CUDA support, that often is not available.

1.3.3 Distributed optimization via simulation

In the literature, there are many works that use Optimization via Simulation, and most of the results found present ad-hoc solutions to solve the problem. Such solutions require optimization algorithms. Evolutionary algorithms and related stochastic optimization algorithms are well-established approaches to optimizing many kinds of models and have been used to tune models of neuron behavior [29, 30], agriculture [31] and textile folding [32].

Some general-purpose evolutionary algorithm frameworks exist that offer massively distributed algorithms and configuration options useful for Optimization via Simulation.

- ParadisEO [33], published under a non-commercial license, are libraries for combinatorial optimization that can deal with parallel metaheuristics methods.

- ECJ [34] is an open-source system for evolutionary computation. ECJ can be used for developing evolutionary algorithms and general integration of simulation code on massively parallel systems. ECJ is integrated with Java-based simulation code (for instance, written in MASON).

Only a handful of software tools are available that allow researchers to apply Optimization via Simulation techniques to simulations without needing to implement their framework from scratch.

- EMEWS [35], uses the general-purpose parallel scripting language Swift/T to generate highly concurrent simulation workflows. These workflows enable the integration of external parameter space exploration (PSE) algorithms to coordinate the running and evaluation of large numbers of simulations. These also allow implementing simulation optimization (SO) process easily.

- OpenMOLE [17], provides an execution platform that distributes simulation experiments on HPC environments using a domain-specific language (DSL) that is an extension of the Scala programming language.

- MEME [36], is based on virtual hosts specially prepared for simulation experiments, deployed on the cloud computing infrastructure Amazon AWS.

- OptTek [37] offers several proprietary tools for metaheuristic optimization. OptTek optimization engine is also directly integrated into a number of ABM and simulation tools, for instance, AnyLogic.

- SOF [38] is a cloud-based OvS framework focused on NetLogo and MASON simulations, providing a generic simulator interface for other types of simulations. The focus is on Hadoop clusters, although other distributed architectures appear to be able to be supported.

- Dakota [39] combines a number of optimization, design of experiment, and uncertainty quantification libraries developed by Sandia National Laboratories. Dakota can be used on several machines, from desktops to HPC systems.

1.4 Dissertation Structure

This dissertation discusses Frameworks and Parallel Languages for SC. Chapter 2 provides the results about FLY Language, a novel Domain-Specific Language that aims to exploit the computation power of multi-cloud infrastructures in a simple way for domain experts that doesn't have a strong parallel and distributed background. Chapter 3 describes Rust-AB, a discrete events simulation engine designed to be a ready-to-use ABM simulation library suitable for the ABM community. Chapter 4 presents three tools for the implementation of OvS processes that

exploit different computing system architecture: heterogeneous, homogeneous, and cloud computing. Finally, Chapter 5 presents a summary of the results.

Parallel language for computational science

Chapter 2 describes the FLY Language, a Domain-Specific Language for Scientific Computing on Multi-Cloud environment. FLY is a parallel workflow scripting imperative language inspired by functional language peculiarities. FLY provides implicit support for parallel and distributed computing paradigms. FLY perceives a cloud computing infrastructure as a parallel computing architecture on which it is possible to execute some parts of its execution flow in parallel. Thanks to this feature, with FLY it is possible to write SC workflows and run portions of them on different cloud infrastructures and SMP architectures simultaneously, achieving extreme scalability without the complexity of managing all the underlying computing infrastructures and computing distribution.

The FLY Language is presented in the following two papers:

[40] Cordasco G., D'Auria M., Negro A., Scarano V., Spagnuolo C. (2020) "FLY: A Domain-Specific Language for Scientific Computing on FaaS". In: Schwardmann U. et al. (eds) EuroPar 2019: Parallel Processing Workshops. Euro-Par 2019. Lecture Notes in Computer Science, vol 11997. Springer, Cham.

[41] Cordasco, G, D'Auria, M, Negro, A, Scarano, V, Spagnuolo, C. "Toward a domain-specific language for scientific workflow-based applications on multicloud syste". Concurrency Computat Pract Exper. 2020.

Chapter 2 also presents a methodology to realize OvS processes exploiting the computational power provided by the Function-as-a-Service (FaaS) models of cloud providers. This methodology allows executing concurrently the massive number of simulation execution (needed by the OvS process) over a multi-cloud system straightforwardly and transparently using the FLY Language.

Agent-based simulation

High-performance ABM simulations are built upon performance-critical operation, and interactions exhibit multiple levels of concurrency. Implementing an efficient framework for the development of ABM simulations is extremely challenging, and the choice of the implementation language is a crucial aspect to consider. Chapter 3 presents Rust-AB, an ABM simulation library using the Rust Language. Rust is a multi-paradigm system programming language with performance comparable with C. Its main feature lies in its memory model, designed to be both memory and thread-safe. This feature makes it extremely interesting in the development of a parallel library for ABM simulations. The Rust-AB simulation library is presented in the following paper:

[42] Antelmi A., Cordasco G., D'Auria M., De Vinco D., Negro A., Spagnuolo C. (2019) "On Evaluating Rust as a Programming Language for the Future of Massive Agent-Based Simulations" In: Tan G., Lehmann A., Teo Y., Cai W. (eds) Methods and Applications for Modeling and Simulation of Complex Systems. AsiaSim 2019. Communications in Computer and Information Science, vol 1094. Springer, Singapore.

However, the Rust-AB library presented in [42] is sequential. Chapter 3 also shows its parallel version, highlighting the changes made and the performance achieved.

Distributed optimization via simulation

Chapter 4 describes Heterogeneous Simulation Optimization (HSO), a framework for developing OvS process in a heterogeneous computing system. The HSO architecture is based on the Master/Worker paradigm. It was designed to exploit the computational power of several hardware architectures, for instance, General-purpose CPUs and/or GPUs, as well as exploiting different programming languages available on the heterogeneous system. HSO is presented in the following paper:

[43] Cordasco G., D'Auria M., Spagnuolo C., Scarano V. (2018) "Heterogeneous Scalable Multi-languages Optimization via Simulatio". In: Li L., Hasegawa K., Tanaka S. (eds) Methods and

Applications for Modeling and Simulation of Complex Systems. AsiaSim 2018. Communications in Computer and Information Science, vol 946. Springer, Singapore.

Chapter 4 also present a way to implement automated and distributed ABM parameters and agents behavior calibration. This methodology marries two tools popular in their respective fields: the MASON agent-based simulation toolkit [23], and the ECJ evolutionary optimization library [34]. This methodology is presented in the following two papers:

[44] Matteo D'Auria, Eric O. Scott, Rajdeep Singh Lather, Javier Hilty, and Sean Luke. 2020. "Distributed, Automated Calibration of Agent-based Model Parameters and Agent Behaviors". Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1828–1830.

[45] D'Auria M., Scott E.O., Lather R.S., Hilty J., Luke S. (2020) "Assisted Parameter and Behavior Calibration in Agent-Based Models with Distributed Optimization". In: Demazeau Y., Holvoet T., Corchado J., Costantini S. (eds) Advances in Practical Applications of Agents, Multi-Agent Systems, and Trustworthiness. The PAAMS Collection. PAAMS 2020. Lecture Notes in Computer Science, vol 12092. Springer, Cham.

Finally, Chapter 4 shows an implementation of an OvS process that uses the SOF framework [38, 46] to solve the Cruise Itinerary scheduling problem, an attractive real logistic problem. This use case is presented in the following paper:

[47] M. Carillo, M. D'Auria, F. Serrapica, C. Spagnuolo, C. Caligaris and M. Fabiano, "Large-scale Optimized Searching for Cruise Itinerary Scheduling on the Cloud," 2019 5th International Conference on Optimization and Applications (ICOA), Kenitra, Morocco, 2019, pp. 1-6.

Chapter 2

FLY: A Domain-Specific Language for Scientific Computing on Multi-cloud Systems

2.1 Introduction

The cloud computing paradigm [48], since the beginning, has reshaped the horizon of computing systems. It becomes the skeleton of modern computing systems by offering subscription-based services that follow a pay-as-you-go model. Various service models have followed, starting from IaaS, which provides the ability to request virtual machines, to Software-as-a-Service, which allows you to take advantage of applications hosted on the cloud. In recent years, a new service model called Function-as-a-Service (FaaS) has been emerging, allowing small snippets of code to be executed transparently on the cloud infrastructure without knowing where and when the code is executed.

From the beginning of Cloud Computing, it was clear that it would represent an opportunity in the SC field. It allows us to develop and run extremely scalable applications while maintaining a low cost compared to HPC applications, as presented in [49].

Commonly, the development of computational science applications

requires a combination of general-purpose languages or parallel languages/frameworks (see Section 1.3.1 for more details). However, computational science applications are typically computationally expensive and require the computational power of distributed systems (clusters or HPC). Recently cloud companies are providing specialized services that are of fundamental importance in realizing high-performance scientific applications. An example can be the MapReduce [50] programming paradigm formed as a cluster of machines on which the Apache Hadoop framework is present. Although cloud providers provide highly scalable solutions, migrating scientific applications to IaaS or PaaS architecture is a huge and complex task that can hide serious cost considerations. These preclude scientific application developers from taking full advantage of cloud computing's scalability and affordability in their application domain.

Nowadays, the need for several simple requirements such as availability, cost reduction, or special functionality has led to the shift from single-user private clouds to multi-tenant clouds [51]. The main advantage of adopting multi-cloud strategies in application development is undoubtedly to take advantage of the huge amount of computing resources provided simultaneously by multiple cloud providers. However, there is an additional level of management complexity, which requires specific skills.

This work aims to reconcile Cloud and High-Performance Computing by providing a powerful, effective, and pricing-efficient tool for the development of scalable workflow-based scientific computing applications [52, 53, 46], on different FaaS platforms, eventually, at the same time (as a multi-cloud system), through the design and implementation of the FLY Domain-Specific Language (DSL). FLY is *powerful* because it enables to exploit the computing capabilities of different cloud providers at once, in a single application, and, then, the most efficient solutions can be merged together. FLY is *effective* because it consists of a user-friendly programming language that frees the programmer from the management and configuration of several complex computation systems. Finally, FLY is *pricing-efficient* because the programmer becomes conscious of the maximum computing costs based on the prices provided by various cloud providers. In this

way, the programmer also has the possibility to choose the service that provides the best value for money, based on the characteristics of the computation that is going to perform.

The next section contains a brief discussion about Domain-Specific language.

2.1.1 Domain-specific languages

Domain-Specific Languages (DSLs) provides a notation tailored toward an application domain based only on the concepts and features relevant to the domain. DSLs enable to express solutions at the same level of abstraction of the problem domain. This abstraction can help shift the development of applications from software developers to a broader group of domain experts, who have a deeper understanding of the domain while having less technical expertise. Furthermore, DSLs are much easier to learn, given their limited scope. DSLs have specific design goals that contrast with general-purpose languages: DSLs are much more expressive in their domain and should exhibit minimal redundancy. A well-designed DSL should exhibit the following three peculiarities [54]:

- Provides a direct mapping to the artifacts of the problem domain.
- Uses the problem domain's vocabulary to improve communication between developers and domain experts.
- Abstracts the underlying implementation. A DSL must not contain accidental complexities that deal with implementation details.

There are many discussions about the advantages and disadvantages of DSL:

- *Pro*. Domain-specific languages have different expressive power compared to general-purpose languages, but they can significantly shorten the time for the development of an application. DSLs improve the correctness of the developed application and the communication between the domain expert and the programmer.

DSL can be used as a mechanism to protect software systems as intellectual property and be a powerful tool for creating self-documented code. DSL allows combining multiple programming paradigms and lowering syntactic noise.

- *Cons*. Regardless of the lower final cost of the overall development, a higher starting price of the application development is often pointed out as a disadvantage. Developing an application that involves building an appropriate DSL is a hard process that requires programmers to be language experts. In such cases, the creation of DSL requires complete knowledge of domain constraints. Debugging and unit testing is hard to perform when using a DSL. DSLs can lead to language cacophony. Proper selection of DSLs and fair usage is crucial.

DSLs can be categorized as external or internal [55]. An External DSL is like making a completely new language from beginning to end. The language itself is separate from the development language and requires many programming concepts found in developing a general-purpose, high-level language. Typical examples of External DSL are languages like SQL, CSS, and HTML. Most of them are bound to a particular technology or infrastructure. Often External DSLs are interpreted or translated through code generation tools into GPL code. The advantages of external DSLs include loose specification and minimal or no following of common standards. In such a way, developers can express the domain artifacts in a compact and useful form. An internal DSL is embedded in the main language. Internal DSLs are a particular way of using GPL providing domain friendly syntactic sugar to the existing API, using underlying programming language constructs.

Developing a DSL means developing a compiler that can read text written in that DSL, parse it, process it, and eventually interpret it and generate/translate the code. With more details, DSL development requires the following phases:

- *lexical analysis*, in which the code is partitioned in *token*, where each token is a single atomic element of language;
- *syntactic analysis*, which checks that the identified tokens form a

valid statement in the language. This phase produces the Abstract Syntax Tree (AST), which is a representation of the syntactic structure of the programs;

- *semantic analysis*, which checks that the assignment of values is between compatible data types (type checking). This phase keeps track of identifiers, their types and expressions, and checks whether identifiers are declared before their use;

- *code generation*, which generates the machine code or code in another language.

DSL developers may benefit by using a DSL development framework, which helps them to develop each implementation phase. One of these is *JetBrains MPS* [56], which is a tool for designing domain-specific languages based on language-oriented programming. It uses projectional editing, which enables overcoming the limits of language parsers and building DSL editors.

Another popular tool is also *Xtext* [57], it is a framework for the development of programming languages and domain-specific languages. With Xtext, the DSL developers need only to provide a grammar specification for the novel language. Xtext, starting from this definition, provides a full infrastructure, including lexer, parser, the AST model, linker, type-checker, compiler, and editing support for Eclipse and other editors that supports the Language Server Protocol as well as web browsers. However, every single aspect can be customized by the DSL developer.

2.1.2 Cloud computing service models

Cloud computing enables companies to use computing resources as a service (like electricity) rather than buy, set-up, and maintain computing infrastructures in-house. Several cloud computing service-models [58] well known as *Software-as-a-Service (SaaS)*, *Platform-as-a-Service(PaaS)* and *Infrastructure-as-a-Service(IaaS)* have been proposed during the last two decades.

- *Software-as-a-Service (SaaS)*: is a software distribution model in which applications are hosted by cloud providers and made available on the web. SaaS offers implicit scalability provided by the cloud infrastructure. The users use the software, and the application is able to scale according to the cloud computing provider capacity.

- *Platform-as-a-Service(PaaS)*: A paradigm for delivering applications frameworks on the Internet without downloads or installing it. This service-model allows the user to build their cloud applications on top of systems running over the cloud infrastructures.

- *Infrastructure-as-a-Service(IaaS)*: is the outsourcing of computing power required by the customers. These resources involve disk space, hardware, and networking components. Using this model-service is possible to execute: virtual machines over the cloud infrastructure and our databases or messaging services.

The Scientific Computing community may get advantages from cloud computing adoption in their compute-intensive applications and workflows. First of all, it is possible to use IaaS for executing applications on high-performance machines or huge machine clusters. Moreover, a cloud computing provider can offer SaaS or PaaS dedicated domain specific services for scientific and data analysis purposes, as machine-learning services, MapReduce frameworks, data-mining services, and so on. Nevertheless, these kinds of service models do not allow the users to deploy easily functional-partitioning applications, or moreover, write ad-hoc applications that can exploits in their executions the cloud computing high scalability.

In the last years, a new service model named Function-as-a-Service (Faas) [67, 68, 69] is established. This service model answers the need for a new scalable price-effective cloud applications era, providing an easy framework for deploying Extreme-scalable functional-partitioning applications. FaaS allows the developers to not care about administering and installing their servers running their backend applications. FaaS

[1]Amazon AWS Lambda pricing. [2]Microsoft Azure Function pricing. [3]Google Function pricing. [4]IBM Bluemix pricing.

Cloud Infrastructure	**FaaS Service**	**API Languages**	**FaaS Languages**	**Pricing and Limitations**
Amazon Web Services [59]	AWS Lambda Function	Java, .NET, Node.js, PHP, Python, Ruby, Go, C++, REST	JavaScript, Java, Python, Go, C#.	1 M functions and 400.000 GB/s of execution time free per month[1]. The execution time of a single function is limited at 300s.
Microsoft Azure [60]	Azure Function	.NET, Java, Python, Go, Node.js, REST	C#, F#, JavaScript, Java	1 M functions and 400.000 GB/s of execution time free per month[2]. The execution time of a single function is limited at 300s.
Google [61]	Google Function	REST, RPC	JavaScript	2 M functions, 1 M seconds of execution and 5 GB of network traffic free per month[3]. The execution time of a single function is limited at 540s
IBM Bluemix/Apache OpenWhisk [62] [63]	Action	REST	JavaScript, Python, Java, PHP, Swift, Docker and native binaries, Go	5 M of functions and 400.000 GB/s of execution time free per month[4]. The execution time of a single function is limited at 600s.
Fission [64]	Fission function	REST	C#, Go, JavaScript, PHP, Python	
Fn Project [65]	Fn Function	REST	Java, Go,Ruby, Python, PHP, JavaScript	
kubeless [66]	Kubeless Function	REST	Python, JavaScript, Ruby, PHP, Go, .NET, Ballerina	

Table 2.1: Cloud Computing infrastructures API and FaaS programming languages fragmentation.

could be seen as another application on PaaS. The main difference between these models is the scalability feature of FaaS. FaaS is an implicit scalable framework. It allows the user to execute functions written in different languages over the cloud infrastructure without care

which server is running the code. The second advantage of FaaS is the price-effective feature. Each function execution has associated a price proportionally to the time e and computation resources need to execute the function.

Commonly, a FaaS service-model architecture is event-triggered, which means users must deploy functions on the cloud infrastructure before they can execute them. A function is executed in response to a new event on the cloud infrastructure (a new record in a database, a message on a queue, etc.). Table 2.1 shown the most famous Cloud Computing infrastructures (open-source and private companies) that provide the FaaS service-model. Table 2.1 presents the supported programming languages for the cloud API and FaaS backend for each cloud infrastructure. Furthermore, the last column describes the execution limitations for each function and the price plan provided by the cloud company (if it is present).

2.2 FLY Language

FLY has been designed in order to enable the domain developers (i.e., domain experts with limited knowledge about complex parallel and distributed systems) to develop their applications exploiting data and task parallelism on a FaaS architecture. This is achieved by a rich language that provides domain-specific constructs that allow the developers to easily interact with different FaaS computing backends, using an abstraction of the cloud provider's services. FLY has been designed from the scientific computing community perspective in which scientists are interested in exploiting cloud computing easily, rather than rely on a general purposes language or dedicated languages API. The FLY principal design goals are:

- *expressiveness,* in deploying large-scale scientific workflow-based applications;
- *programming usability,* writing programs for FLY should be straightforward for domain experts, while the interaction with the cloud environment should be completely transparent; the users do not need to know the cloud providers services;

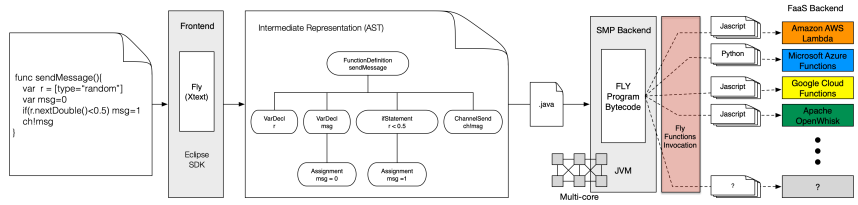

Figure 2.1: FLY compilation workflow.

- *scalability,* either on SMP architectures or cloud computing infrastructures that support FaaS.

FLY provides implicit support for parallel and distributed computing paradigms and memory locality, enabling the users to manage and elaborate data on a cloud environment without the effort of knowing all the details behind cloud providers API. A FLY program is executable either on an SMP or a Cloud infrastructure (supporting FaaS) without a deep knowledge of the underlying computing resources. FLY is translated in Java code and is able to automatically exploit the computing resources available that better fit its computation requirements. The main innovative aspect of FLY is represented by the concept of FLY *function*. A FLY function can be seen as an independent block of code that can be executed concurrently. FLY is designed as an enhanced scripting language. It is a sequence of instructions and a number of FLY functions invocations. FLY functions can be executed in sequential mode, parallel on an SMP, or on a FaaS backend. The language provides programming constructs for function definition, execution, synchronization, and communication. Communication among different environments/backends is obtained through some virtual communication path named *channels.* Along these lines, FLY has been designed as an enhanced scripting language and is composed of a sequence of standard instructions integrated with a number of FLY functions invocation, which communicates via channels.

Figure 2.1 depicts the FLY compilation workflow. On the left side, the FLY program is given in input to the compiler (written using XText).

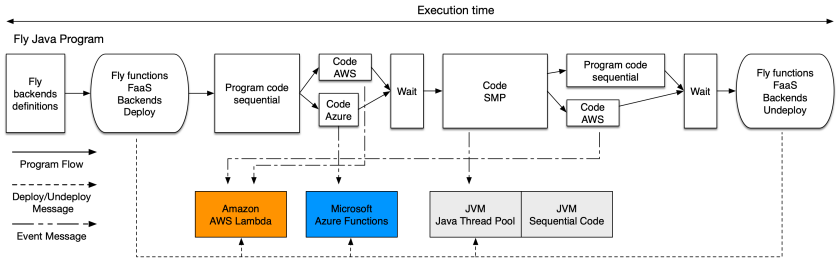

Figure 2.2: FLY execution workflow.

The intermediate AST representation is translated into a Java program. Each FLY function is translated into different executable codes (one for each backend). Therefore FLY provides compiled functions code that can be executed on each cloud infrastructure backend (see the right side of Figure 2.1).

In detail, in Figure 2.2 we show a general execution flow of a FLY program along the execution time. First of all, the program initializes all the backends required by the FLY code, and *deploys* the generated code on the corresponding backend. We notice that the FLY functions are already compiled when the main FLY program is executed, thereby avoiding run-time compilation overheads. After these initialization steps, the main program is executed following the FLY code instructions. Each time the fly keyword is used, the program generates *events* on the corresponding SMP and/or FaaS backend, in order to execute the FLY functions. FLY supports synchronous and asynchronous execution models.

Before presenting the FLY language design, Listing 2.1 shows a simple example of a FLY program, which computes the PI estimation through the Montecarlo Method on Amazon AWS Lambda backend. Briefly, the PI Monte Carlo estimation algorithm generates random points and counts the number of points inside a circle of diameter 1.0. Given the sum of points, it computes the estimation of PI as $\frac{S*4.0}{N}$, where S is the number of points inside the circle and N is the total

number of generated random points.

Listing 2.1: PI Montecarlo Estimation on Amazon AWS

```
1  var local = [type="smp",threads=4]
2  var aws = [type="aws",profile="default", access_key="
       ...",secret_key="...",region="us-east-2",language="
       python3.6",concurrency=1000,memory=128,seconds=300]
3  var ch = [type="channel"] on aws
4  func hit(i){
5  var r = [type="random"]
6     var x = r.nextDouble()
7     var y = r.nextDouble()
8     var msg=0
9     if((x*x)+(y*y)<1.0){msg=1}
10    ch!msg on aws
11 }
12 func estimation(){
13    var sum = 0
14    var crt = 0
15    for i in [0:10000] {
16     sum += ch? as Integer
17     crt += 1
18     }
19    println "pi estimation: "+ (sum*4.0)/crt
20 }
21 fly hit in [0:10000] on aws thenall estimation
```

The FLY PI code defines, at line 1, a new SMP environment, and line 2, a new Amazon AWS FaaS backend. An important aspect to note is that an SMP environment must always be declared since a part of a program's execution takes place locally. Line 3 declares a new channel on the environment `aws` that allows the program to communicate with FLY functions and also FLY functions with other functions. At line 4, a FLY function `hit` is defined, it uses a pseudo-random generator (initialized using the clock time) to estimate a point inside the circle of diameter 1.0. While at line 10, a new message is sent on the channel `ch`, the message value is 1 if the point is inside the circle and 0 otherwise. Moreover, at line 12, a second function `estimation` reads a sequence of integers number from the channel `ch` and write on the standard output the estimation of PI, computed according to the sum of the messages. At line 21, 10000 `hit` functions

are executed synchronously on the `aws` backend. When all functions terminate, the function `estimation` is called on the SMP backend. It should notice that each FLY functions cannot use variables declared outside the function scope, excepts for variables of type channel (see Section 2.2.1).

2.2.1 Language definition

FLY is a parallel workflow scripting imperative language inspired by functional language peculiarities. FLY has a statically checked type system, however, it allows the users not to use types during variable declaration. FLYconsiders that variable types cannot change during the program execution. Types are automatically inferred when the variable is first assigned. For this reason, FLY enables runtime type casts as in Java or C#, programmers can not care about types until it is not used in particular cases (for instance, when a variable is a function parameter or a message on a channel, see paragraph 2.2.2 and 2.2.1). FLY supports inter-process (and inter-FLYfunctions) communications using channels according to communicating sequential processes (CSP) definition. The FLY syntax and concepts are a fusion of different languages such as Java, JavaScript, Python, and R, it ensures familiarity with the most powerful and famous general purposes/data science languages.

FLY provides what a programmer expects from a scripting language: expressions, relational expressions, boolean operations, and code comments as in Java (using the `//` or `/* block comments */`). Moreover, FLY is DSL for scientific computing it provides several domain-specific constructs for parallel/distributed task/data-based parallelism, in the next sections are explored the FLY concepts, types, constructs, executions models, and libraries extensions that we have defined for our scientific scalable computing language.

Data models and types. FLY provides a rich set of data-types, inherited by Java. From booleans, integers and reals (double point precision floats) to strings are named basic types. Table 2.2 shows the FLY types, together with their literal notations. Variables and functions are identified by an *ID* literal. To define a new variable, the user has to

Type Name	Description
ID	`(a..z|A..Z|$) (a..z\|A..Z\|$|_|0..9)*` literal definition for instance variable name or function name.
Basic Types	
Integer	`(..9)+`
Float	`(..9)+ '.' (..9)+`
Boolean	`'true' | 'false'`
String	`" ( \\ . | !( \\ | " ) " )* | ' ( \\ . | !( \\\| ') ' )*`
Array	Collection of homogeneous basic types.
Domain Types	
Object	Collection of basic types elements.
Environment	SMP (local) or FaaS backends.
Channel	CSP structure for communicate between functions and main process.
Random	Generator of pseudo-random numbers.
File	I/O operations on files.

Table 2.2: FLY Literates and Types

use the word var and provide the variable name, eventually assigning the variable an initial value. Variable type is automatically inferred at the first assignment (when a variable is declared, it is not needed to initialize them, FLY can assign the type when it is used). Notice that variable type cannot change during the execution time, differently from other languages like JavaScript.

```
var ID = Integer|Float|String|Object|File|Array|
    Environment
```

FLY defines the notion *Range*, it is a range of values between two integers values in natural numbers ordered. Ranges are useful in loops and FLY functions executions model.

```
[Integer|ID:Integer|ID]
```

A FLY Array is a collection of homogeneous basic types laid on a tabular form having one, two, or three dimensions. The declaration of an array is described in the following:

```
var ID =  Integer|Float|Boolean|String[N] // array
    declaration
```

```
var ID =  Integer|Float|Boolean|String[N][M] // 2-D
    Matrix declaration
var ID =  Integer|Float|Boolean|String[N][M][K] // 3-D
    Matrix declaration
```

where N, M, and K denotes the length (i.e. number of elements) of each array dimension.

The language provides several domain types shown in the second half of the Table 2.2. Six domain types are defined that allow the users to access the memory and interact with computing backends easily.

Object domain type. A FLY object is a heterogeneous collection of basic and/or domain types elements, it is a map able to store key-value couple elements.

```
var ID={ID=VALUE,ID=VALUE,...}  (e.g., var m={i=0,s="
    text"})
```

FLY objects can work as an array or a map. The value of an element can be access by using two different notions the key or the position. Obviously, it is admitted to use both methods to access the structure. However, it is forbidden access by position to an element added by key and vice versa. When the user valorizes a key/position not present in the structure, a new element is created to assign the value to the key/position.

```
name.ID //where name is the Object variable name
name[Integer|ID]
name.ID = VAL // if ID it is not available, it is
    create a new key ID with value VAL.
```

Environment domain type. This is the most important FLY domain type. It is an abstraction of an execution environment. It provides the ability to interact with a FaaS cloud provider or SMP system. For example, when FLY functions are executed on a particular backend, we need a mechanism to refer to the execution environment on which execute the functions, environment object allow to specialize the execution of several backends.

For declaring a new environment, is used a syntax like an object definition, but using the square brackets ([...]). Additionally, the user has to define several parameters that characterize a backend.

```
var ID = [type=("smp"), threads=Integer] //
    declaration for a SMP environment
var ID = [type=("aws|aws-debug|azure"),(cloud-
    provider-access-parameters), region=String,
    language=String,
concurrency= Integer, memory=Integer(MB), seconds=
    Integer(s)] //declaration for a FaaS environment
```

The first parameter is *type* that defines the supported backend. The most simple backend is *smp*, and it allows the user to exploit SMP architecture. The second parameter is *threads*, an integer value that defines the number of the concurrent threads used for the parallel evaluation of the FLY functions. Instead, the declaration of a FaaS environment has a series of further parameters. The declaration of a FaaS environment, on the other hand, is a bit more articulated. First of all, it provides the definition of the various parameters for access to the cloud provider. Since these parameters are different depending on the chosen cloud provider, they have been categorized into *(cloud-provider-access-parameters)*. The *region* parameter defines the geographic region relative to the data center of the cloud provider used. The *language* parameter defines the language in which the serverless functions must be used. The *concurrency* parameter indicates the number of concurrency functions that you want to use. The *memory* and *seconds* parameters respectively define the maximum amount of memory and the maximum execution time of the single serverless function.

Communication statement. The Channel type allows the synchronization and communication between FLY functions and/or the main program. Channel follows the Communicating sequential processes (CSP) definition. A new channel is defined on an environment and can be used for the communication between functions executing on the same backend or with the main program. Channels are blocking message queues. When the program sends a message on a channel, the routine immediately ends. Instead, when it receives a message,

the execution is blocked until a new message has not arrived on the channel.

```
var ID = [type="channel"] on ID //the second ID
   refers and environment variable name.
```

In order to send a message on a channel it is used the character '`!`', `ch!VAL` this instruction send a message VAL on the channel ch, while for reading a message is used the character '`?`', `x=ch?` this instruction read a message from the channel ch and assign the value to the variable x.

In detail, channels use the network to communicate with cloud environments. For this reason, for sending/receiving messages, a serialization mechanism is used. FLY defines the serialization for objects, files, and basic types. It is not allowed to send messages containing environment, channel, and random objects. However, it is not convenient to send a message containing a file, but it can be more efficient to elaborate the file locally and send only portions to FLY functions. In this version of the language definition, the serialization mechanism uses a simple algorithm that transforms objects, files, and basic types in formatted string easily to be deserialized using the JSON format.

Randomness statement. The Random type is a pseudo-random number generator object. A random object allows the user to generates a sequence of pseudo-random number calling the functions `nextInt()` and `nextDouble()`, that respectively return a random integer and a double-precision real between 0 and 1. The random object can be initialized by an integer value (the random seed), given as parameters to the object constructor. We plan to provide an initialization mechanism that allows the user to use a defined sub-sequence of a random number given by a particular random number sequence (this can help write programs that do not have statistical error).

```
var ID = [type="random"]
var ID = [type="random", seed=Integer]
```

File object. The File object is the abstraction of a file in FLY. The language supports two file formats identified by the *type* parameter. The

first supported file type is “dataframe” which refers to the CSV format, while the second is “file” which refers to the format TXT. Moreover, the file object is defined by the parameter *path*, the file system path or reference to the file, and by the separator *sep*, which is an optional parameter defined for CSV files.

The language provides two access methods to file, depending on where the file is stored: *local* and *remote*. When a file is accessed in local memory, it is available locally, and the parameter path defines the local file system path. FLY also allows access to a remote file, available on the cloud infrastructure, providing an environment object where the file is available. For instance, consider that a file is stored on Amazon S3 service[1] the path parameters defines the Amazon bucket URL. An interesting feature of the language is that it allows the automatic upload of a local file to a cloud provider storage mechanism. This is possible simply by declaring a file with a local path but specifying that you want to use it on the cloud through the on clause.

```
var ID = [type="file|dataframe", path=String, sep=
    String] \\local access
var ID = [type="file|dataframe", path=String, sep=
    String] on ID \\remote access
```

FLY provides particular attention to the CSV data managing them as a Dataframe (similar to R language dataframe). The memory is seen as a matrix structure, allowing the user to access to row and columns and used dedicated operations as querying, filtering, random access, etc...

2.2.2 Control structures

Conditional control. FLY provide the *if-else* statement. The behavior of this statement is the same of other languages as Java.

```
if (Boolean Expression)
    BlockExpression/Expression
else
    BlockExpression/Expression
```

[1] Amazon AWS Simple Storage Service

Iterative controls. FLY provides the *foreach* and *while* loop statement. In many languages, the loops syntax, and semantics are not natural and require useless complexity. We have simplified the FLY loops, providing a simple behavior according to the data structure used. Two kinds of *foreach* loops can be used in FLY. The former used the Range definition and allows the user to loops in a range of integer values. The second mode allows the user to iterate over a FLY object or file. Iterate over an object allows the user to access each element of an object. At each loop, the value of the iteration variable is updated with a new FLY object contains two fields: the key (or position) of the object, identified by the ID *key*, and the value, identified by the ID *value*. While iterate over a File Object, create a new FLY object that contains a portion of the file.

```
for ID in Range
    BlockExpression/Expression

for ID in Object|File
    BlockExpression/Expression
```

The *while* statement loop behavior is the same of others languages as Java.

```
while (Boolean Expression)
 BlockExpression/Expression
```

Execution control structures

Functions. The FLY language core concept. Functions are quite different from other scripting languages and follow a functional programming inspired definition. We named functions as FLY function, and it is defined as a code block that can be executed concurrently. A FLY function and can be seen as a task or independent job of our program. For declaring a function is used the word func. Each function can specify a set of input parameters by defining the IDs of the parameters (omitting parameter types). Functions may return a value using the word returns, if a function does not return any value, the value of the last instruction is used as the return value.

```
func ID (ID, ID, ID, ..)
```

```
BlockExpression
```

FLY functions have a private scoping, which means that only function parameters and local variables are visible in the body of the function. The state of the parameters is passed by copying the value, and it is considered immutable. However, functions can avoid this limitation using channels. An outside declared channel can be directly used in the function. This also applies to environment objects. Notice that the FLY language does not ensure that operations are admitted: if a function is executing on a backend B, the function must use only channels defined on the backend B. Functions called by passing parameters values are sequentially executed where they are called. For instance, it is possible to invoke a function in other functions that are executing on a backend, and the execution is made on the same backend. In order to execute functions concurrently FLY provide the fly statement described later (this statement it is not admitted in the body functions), see paragraph 2.2.2.

Types casting. FLY provides support for types casting as in Java and C#. Several cases need the casting operation. For instance, if we need to use a function parameter in the function body, it is required a type casting operation. Types casting operation is admitted on basic and domain types, but it is forbidden on environment and channels. This has to do with the fact that the environment and channel objects cannot be a parameter of functions and cannot be sent on a channel.

```
ID as TYPES
```

Parallel/distributed statement. The definition of FLY functions is the consequence of the explicit parallel execution model of FLY. The language provides the word fly that allow the user to execute concurrently a set of functions (the number of concurrently functions is defined by the backend support and by the user needs, see paragraph 2.2.1). The fly statement can be seen as a loop iteration, in which iterations concurrently run. As described before, for the *foreach* iteration control statement, in FLY is possible to iterate over a range of integer values, an Object, a File, or an Array and for the fly statement is the same (by

using the word in is possible define that). To be used in the fly construct, functions must take only one input parameter (the iteration value) that is automatically passed to the function execution.

```
fly ID in Range|Object|File|Array on Environment then
    ID thenall ID
```

In the second part of the fly statement, an execution environment is defined using the word on. Finally, the fly statement support two kinds of function callbacks, declared using the word then and thenall. The *then* callback is executed for each FLY function executions, instead the *thenall* callback is executed after all FLY function execution.

The above FLY execution model uses FLY functions, which take only one parameter (the iteration value). FLY also supports a single execution mode, in which the FLY function can be called by passing parameter values and executed on an environment.

```
fly ID(VAL,VAL,..) on Environment then ID thenall ID
```

Moreover, FLY explores synchronous and asynchronous execution models. The previous construct defines the synchronous mode, in which the main program waits for all functions termination. It is possible to execute functions asynchronously using the word async before the fly construct.

```
var ID =  async fly ID in Range|Object|File|Array on
    Environment then ID thenall ID
ID.wait()
ID.waitall()
```

The *async* statement returns a special FLY object, named *async-object*, that allows the user to control and interact with the asynchronous execution. The *async* fly constructor invocation immediately returns the control to the main program and the execution can continue. The user can control the status of the asynchronous functions invoking the method `status()` on the *async-object* and can wait the termination of all functions using the method `waitall()` or wait the termination of the first function by invoking the method `wait()`.

Other features. FLY is a language for Java Virtual Machine language (because it is translated in Java code), and for this reason, it inherits all

String and Math operations allowed in the Java language. However, FLY functions can be executed on several backends that may not support Java code. FLY compiler is able to translate these operations also in other languages supported by the different backends. The language documentation provides the complete list and description of math and string operations.

FLY also provides timing operation. The user can compute the execution time using the method `time()`. The method returns the clock time expressed in milliseconds. Notice that it is allowed to call the time operation only to compute interval of time in the same environment (for instance, in a function or in the main program).

FLY provides the native statement, defined by the keyword native, that enables the programmers to invoke directly native code (Python or Node.js code) in the FLY functions.

2.2.3 FLY compiler

FLY source-to-source compiler is available on a public GitHub repository[1]. An implementation of the language grammar and code generators for the SMP and Amazon AWS and Microsoft Azure FaaS backends have been developed. FLY has been developed in order to generate a Java program, which is able to support several backends. The FLY language compiler is developed using *Xtext* [57] framework, which enables the user to create JVM based DSL. The FLY code is translated in a pure Java program that exploits cloud APIs in order to use FaaS services. Xtext leverages the powerful ANTLR parser, which implements an LL parser. In compiler development, it is known that LL-parsing has some drawbacks: LR parsers are more general than LL parsers, which do not allow left recursive grammars. Moreover, LR parsers are, in general, more efficient. On the other hand, LL parsers have significant advantages over LR algorithms concerning readability, debuggability, error recovery, and as well as they are much simpler to understand. Xtext provides an initial grammar specification, a full infrastructure including lexer, parser, AST model, linker, type-checker, compiler, as well as editing support for Eclipse and/or any editor that

[1] github.com/spagnuolocarmine/FLY-language

	Computing Environments/Backends		
Name	**SMP**	**FaaS**	
		JavaScript	**Python**
Integer	java.lang.Integer	Number	Numbers
Float	java.lang.Double	Number	Numbers
Boolean	java.lang.Boolean	Boolean	Boolens
String	java.lang.String	String	String
Object	java.util.HashMap <String,String>	Object	Dict
Environment	java.util.concurrent. ScheduledThreadPoolExecutor; com.amazonaws.auth.*; com.amazonaws.services.s3.*; com.amazonaws.services.lambda.*; com.amazonaws.services. indentitymanagement.*; com.microsoft.azure. AzureEnvironment; com.microsoft.azure.management.*; com.microsoft.azure.credentials.*;		
Channel	java.util.concurrent. LinkedTransferQueue<String> com.amazonaws.services.sqs.*; com.microsoft.azure.storage.queue.*;	aws-sqs azure-storage-queue	aws-boto3-sqs azure-storage-queue
Random	java.util.Random	Math.random()	Random package
File	java.io.FileInputStream	File I/O	File I/O
Dataframe	tech.tablesaw.api.Table [70]	dataframe-js [71]	Pandas Library

Table 2.3: Code generation mapping of FLY types.

supports the Language Server Protocol as well as web browsers. The LL grammar for FLY language, which provides the complete language definition, is presented in Cordasco et al. [40]. Xtext has also been used to develop a code generator that, given the intermediate AST program representation (the output of the first compilation phase), generates a FLY Java program. The FLY Java program execution flow is shown in Figure 2.2. The code generation phase is the core of our compiler, and it generates different codes according to the backend where the FLY code has to be executed.

The code generation phase is designed to be specialized according to the considered backend:

1. *SMP backend.* A Java Thread Pool is used to implement the backend for the SMP architecture. The FLY main program is

executed as Java code on a JVM, which also executes the SMP backend. Only one assumption has been made about the underlying hardware, and it is that it provides at least 2 physical cores, one to run the main program, and one that acts as an SMP backend. Therefore, the FLY functions are translated in pure Java code. In details, all FLY types are mapped on a particular Java type, as described in Table 2.3, while the FLY functionalities are provided exploiting the Java language.

2. *FaaS backends*. The FaaS computing backends have been developed using the API of each cloud provider, as shown in Table 2.3 exploiting a particular cloud service, according to each provider and computing language API, for each type and functionality of FLY. Our FLY compiler translates each FLY function in Java, JavaScript, or Python languages, automatically according to the computing backend where the function will be executed during the workflow. For each backend and each FLY function, the compiler generates a deploying package containing: the source code and libraries structured according to the destination backend. The FLY deploying phase on each cloud backends exploits the cloud client command-line interface (CLI): AmazonAWS CLI[1] and Microsoft Azure CLI [2]. An important aspect to notice is that we invoke (or trig) the functions by using asynchronous HTTP POST requests, which, according to experimental results, ensures the shortest invoking latency.

Finally, the compiler produces a Java Maven project including all dependencies, the FLY Java class program (named as the FLY source code), and the code of functions. The command `mvn package`, which generates am executable *JAR* file, is used to build the project.

[1] aws.amazon.com/cli

[2] docs.microsoft.com/en-us/cli/azure

2.3 Debugging FLY Applications

The FLY language is a workflow language that enables the execution of functions on remote/distributed backends. The programmers have no control over the execution and memory of the remote computing backend. For this reason, debugging FLY applications is very complex, considering that to analyze functions logs, the programmers have to access each cloud provider used in the computation using specific API of Web portals. Moreover, a possible incorrect execution has a considerable cost. Indeed, assuming that a FLY application generates errors only in some rare cases (i.e., memory requirements or index out of bounds), then according to the FaaS execution model, all incorrectly executed functions must be paid as correct executed ones.

For the reasoning above, FLY provides a particular debugging backend for the Amazon AWS cloud provider. By using this backend, the programmer is able to execute and test the code on a local virtual environment that emulates the Amazon AWS ecosystem. The programmer has to define the AWS backend using a particular environment type `aws-debug`. We developed the AWS debugging environment using Local Stack [72], which provides an easy-to-use test/mocking framework for developing Cloud applications. In particular, the AWS debugging environment enables the programmers to test FLY applications onto the local machine using the same functionality and APIs as the real AWS cloud environment. Local Stack framework exploits the containers virtualization technology to emulate the AWS environment, and for this reason, in order to exploit the AWS debugging environment, the Docker[1] application needs to be installed on the local machine.

2.4 Language Evaluation

We evaluated FLY efficiency analyzing the performance of FLY on the Word Count problem, a popular benchmark for distributed workflow computing framework, which consists of listing the frequencies of different words in a set of text files.

[1] www.docker.com

Listing 2.2: Word Count on Amazon AWS

```
var local = [type="smp",threads=4]
var aws = [type ="aws",user="default", access_key="k1",
    secret_key="k2", region="us-east-2", language="
   python3.6", threads=1000,memory=128, seconds=300]
var ch = [type="channel"] on aws
var dir = [type="file", path="data"]
for f in dir{
    var file = [type="file", path=f] on aws
}
func count(files){
    var counts = {}
    for f in files{
        var file = [type="file", path=f]
        for strs in file{
            var words = strs.split(" ")
            for w in words{
                var word = w.v as String
                if(not counts.containsKey(word))
                   counts[word] = 1
        else
            counts[word]=counts[word]+1
    }}}
    ch!counts on aws }
func reduce(){
    var total_counts = {}
    for i in [0:1000]{ //this values depends on the
       number of concurrent functions requested
        var res = ch? as Object
        for w in res{
            var word = w.k as String
            var tmp = w.v as Integer
            if(not total_counts.containsKey(word))
                    total_counts[word] = tmp
                else
                    total_counts[word] = total_counts[
                       word] + tmp
        }
    }
    for value in total_counts{
        println value.k+ " "+ value.v
    }
}
```

```
39 var t_start = time()
40 fly count in dir on aws thenall reduce
41 println time(t_start)
```

Listing 2.2 shows the FLY code for the Word Count problem. Initially, the input files are loaded on an AWS S3 bucket (lines $5-7$). Lines $8-21$ show the code of the `count` function, which takes as a parameter a list of string (files path), which are the AWS endpoint of the input files on the S3 bucket, and iteratively counts the number of occurrences of each word. Then, a message containing the FLY object that maps each word with the corresponding frequency is sent on the channel `ch` (line 21). Lines $22-38$ provide a `reduce` function that will be executed on the local machine. The `reduce` function reads from the channel `ch` the words frequencies, computed by the FLY functions, and generates a single occurrences list. Finally, line 40 shows the FLY construct to execute the process on the backend AWS, which executes at most 1000 concurrent `count` functions on AWS. Each function receives a balanced subset of the files contained in the local directory *data*. When all FLY functions are completed, the `reduce` function will be executed (synchronization is obtained thanks to the synchronous FLY construct, line 41).

We performed several benchmarks of the Word Count program to explore FLY capabilities in terms of performance and cost-effectiveness. Table 2.4, shows the results obtained running nine different input size (rows) and varying the execution backends (columns). Specifically, we tested three files size: 125, 250 and 500MB, with three cardinalities of files: 500, 1500 and 2000. Therefore we analyzed, in the worst-case, approximately 1 Terabyte of data.

We compared the performance of the sequential executions (FLY Java) against two parallel executions, using 4 (SMP-4) and 64 (SMP-64) cores, and a distributed execution using 1000 concurrent FLY functions at the time (AWS-64). Furthermore, we evaluated the performances of the previous configuration of FLY compared to the execution of the same task on Apache Hadoop[1], a well-know big data analytic framework (H-16). We used the Amazon AWS cloud as computing

[1] hadoop.apache.org

environment, in particular AW-64 exploits a *m4.16xlarge* EC2 instance[1], which provides 64 virtual cores. We measured the total computation time (in hours) without considering the time required to load the files on the computing environment.

We also performed the same experiment using an *m4.xlarge* EC2 instance, which provides 4 virtual cores, obtaining similar results in terms of computational time and cost. There are several possible explanations for this outcome. First of all, the computing time for FLY functions is the same in both configurations. Moreover, the greatest number of cores of AWS-64 are exploited only for the *reduce* functions, which is not a computationally intensive task for this task.

The H-16 configuration refers to the same task executed on an Apache Hadoop cluster machine, running exploiting the Amazon AWS EC2 and Elastic Map Reduce (EMR) services. We used a cluster of 16 computing nodes and one master node for a total of 128 virtual cores. Each node is a *m4.xlarge* EC2 instance, which costs $0.80\$$ hourly plus the cost of EMR running on it that is $0.24\$$ hourly, hence the total cost of the cluster is $16.64\$$ hourly. We notice that this cluster configuration requires, on average, about 20 minutes of cold starting time. For this reason, we added to the cost of each experiment the price to start the cluster, but we present, in Table 2.4, only the time for running the experiment, also without considering the time to load the file on the HDFS.

Table 2.4 shows the obtained speed-up concerning the sequential execution and the total cost (in dollars) considering the cost of the FLY functions and the cost of the EC2 instance machine running the FLY main program. As highlighted in the table, the best performance has been obtained by the backend that exploits FaaS (AWS-64). On the other hand, H-16 is the most expensive experiment, while SMP-4 results as the cheaper execution configuration. As shown, the performance of the FaaS backend is the most effective, compared to the parallel backend and the Apache Hadoop, also considering their cost for running on the Amazon AWS. It is important to notice that we adjust the number of nodes composing the Hadoop cluster machine to be comparable in cost to the configuration execution using FaaS.

[1] aws.amazon.com/ec2/instance-types

Configurations			FLY Java		SMP-4			SMP-64			AWS-64			H-16		
MB	N	Tot. GB	T (h)	C ($)	T(h)	S	C ($)	T(h)	S	C ($)	T(h)	S	C ($)	T (h)	S	C ($)
125	500	$\sim$ 62	0.751	0.2	0.324	2.3	**0.1**	0.016	44.9	0.1	**0.015**	**47.7**	0.2	0.061	12.3	6.6
125	1500	$\sim$ 187	3.036	0.6	1.042	2.9	**0.2**	0.089	34.1	0.3	**0.018**	**166.6**	0.5	0.103	29.5	7.3
125	2000	$\sim$ 250	3.711	0.7	1.351	2.7	**0.3**	0.124	30.0	0.4	**0.018**	**196.1**	1.0	0.106	35.0	7.3
250	500	$\sim$ 125	1.518	0.3	0.670	2.3	**0.1**	0.062	24.3	0.2	**0.018**	**82.9**	0.5	0.560	27.1	6.5
250	1500	$\sim$ 375	4.659	0.9	2.069	2.3	**0.4**	0.177	26.3	0.6	**0.023**	**198.6**	1.5	0.146	29.9	8.1
250	2000	$\sim$ 500	6.060	1.2	2.738	2.2	**0.5**	0.234	26.0	0.8	**0.024**	**250.6**	2.0	0.205	29.6	8.9
500	500	$\sim$ 250	2.911	0.6	1.323	2.2	**0.3**	0.116	25.0	0.4	**0.023**	**121.8**	1.0	0.106	27.5	7.3
500	1500	$\sim$ 750	9.017	1.8	3.981	2.3	**0.8**	0.359	25.1	1.2	**0.034**	**261.3**	2.5	0.302	29.9	10.6
500	2000	$\sim$ 1000	10.918	2.2	5.065	2.2	**1.1**	0.437	25.0	1.4	**0.035**	**307.6**	3.8	0.400	27.3	12.2

Table 2.4: FLY Words Count backends comparison: T (Time, h) hours vs S (Speed-up with respect to FLY Java) vs C (Costs, $).

FLY as a distributed computing framework It is worth mentioning that our experiments on H-16 are consistent with the complete analysis presented in [73]. The authors detail the performance of several benchmarks (such as Word Count, K-means, and so on), on several cloud-based cluster machines. In particular, they exploited an Amazon EC2 cluster composed by 8 *r5.24xlarge* instance type, where each machine is equipped with 96 cores and $768GB$ of memory. The described results provide a good comparison for our purposes. As described by the authors, Hadoop spends about 0.3 hours (18.3 minutes) for counting 1.6TB of data. Analyzing the cost for this experiment, we have that each EC2 instance cost 6.048\$ per hour, while the *Elastic MapReduce* service, running on this kind of instance, costs 0.26\$ per hour, hence the total cluster cost for 0.3 hours is about 14.60\$. We can compare this data with our results on FLY, which computes the word count on 1 TB of data with a cost of 3.8\$. Moreover, our system is a clear advance in terms of performance by also considering the opportunity to exploit multi-cloud, which enables the programmer to easily increase the amount of computational resources by exploiting different cloud providers in the same application. Moreover, the paid price for our solution is extremely cheaper (about 4 times less). We plan to investigate a more detailed comparison by running this benchmark, varying the computing backends on different cloud providers, like Microsoft Azure, and comparing the cost and performance of Apache Hadoop and Apache Spark running on the same cloud provider.

2.4.1 Use cases

In the following, we provide two well-known scientific computing use cases developed in FLY: *supervised machine learning classification* and *sequence alignment*. For both the problems, we identified a well-known algorithm that has been developed and tested using FLY. With more details, we developed a FLY implementation of the linear k-Nearest Neighbors (k-NN) algorithm, which is one of the most used learning algorithms, as well as the Smith-Waterman algorithm, a dynamic programming algorithm for determining the optimal local alignment between two strings according to a gap-scoring scheme.

We tested the performance of k-NN algorithm using the *20 Newsgroups (20NG)* [74] data set, which comprises a collection of approximately 20000 newsgroup documents. The data is organized into 20 different newsgroups, each corresponding to a different topic. The data set is composed of 19300 instances, described by 1006 attributes. We used 9000 instances for the training set, while a combination of instances in the data set plus some synthetic instances has been used for the test sets. For the Smith-Waterman algorithm, we performed the alignment of several synthetic protein sequences using the BLOSUM50 matrix as a gap-scoring scheme.

The experimental case studies evaluation has been performed using six computing environments, as described in Section 2.4. Five environments exploit FLY: Java, SMP-4, SMP-64, AWS-4, and AWS-64. Moreover, we considered a baseline environment represented by a pure Java implementation, running on a single thread machine. In order to perform a cost comparison of each environment, we executed all the tests on AWS using AWS EC2 instances. In particular, we used the r5.large (2 vCPU, 16GB RAM), r5.xlarge (4 vCPU, 32GB RAM) and m4.16xlarge (64 vCPU, 256GB RAM) instances to run respectively the tests that requires 1, 4 and 64 threads. FLY functions have been deployed on AWS Lambda, configured with 3GB of memory. It is worth mentioning that tests performed on different environments always provide the same result (this was expected since we did not modify the algorithms).

k-NN algorithm

We developed several experiments varying the number of testing instances and the computing environment. For each test, we compute the total computation time and its cost. Figure 2.3 presents the performance in terms of strong scalability on the 20NG data set for each computing environment (series). It is worth mentioning that the implementation of the algorithm in Pure Java and FLY provide the same accuracy. Therefore in the following, we compare the different configurations only in terms of computation time. The total computing times are shown in milliseconds (y-axis, log scale), varying the number of instances in the

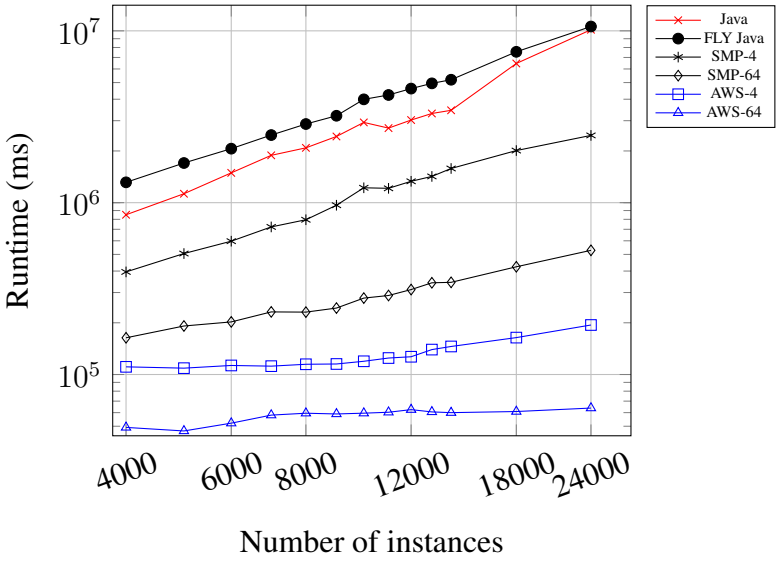

Figure 2.3: FLY Scalability on k-NN.

test set (x-axis, log scale) from 4000 to 24000. As shown in the figure, the best performance is always achieved by the configurations that use the FaaS backend. The pure Java code is slightly better than the SMP using 1 thread (this slow-down represents the overhead due to FLY), but it is outperformed by the SMP configuration with 4 threads. The overall billing cost for the larger test set ranges from ≈ 0.35\$ for the SMP-4 configuration (i.e., the cost of the r5.xlarge instance) to ≈ 0.85\$ for the AWS-64 configuration (i.e., the cost for the m4.16xlarge instance plus the cost for the AWS Lambda functions). We also notice that the pure Java implementation is more expensive (≈ 0.37\$) than the SMP-4 configuration.

Smith-Waterman algorithm

Similarly to the k-NN evaluation, we developed several experiments using the Smith-Waterman Algorithm, varying the length of the sequences to be aligned. Figure 2.4 presents the performance in terms of scalability on the synthetic protein sequences for each computing environment (series). As for the previous case, the best performance is

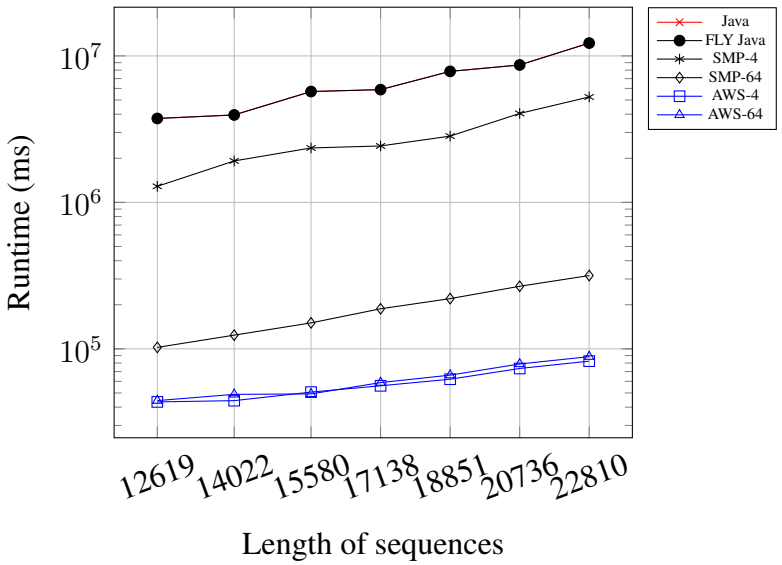

Figure 2.4: FLY scalability on Smith-Waterman.

	k-NN				Smith-Waterman			
	4k	8k	12k	24k	≈12k	≈16k	≈19k	≈23k
SMP-4	1.76	2.61	1.96	6.46	**22.21**	15.49	20.12	14.30
SMP-64	1.06	3.22	3.70	14.55	138.29	149.23	131.19	**154.47**
AWS-4	**0.34**	1.91	3.48	21.85	60.85	76.46	83.28	92.46
AWS-64	0.72	3.46	6.66	**63.58**	56.26	78.26	76.26	94.88

Table 2.5: $\frac{Speed-up}{Cost}$ Comparison.

always achieved by the configurations that exploit the FaaS backend but in this experiment, the capability of the machine that executes the FLY main program does not affect the performance. Moreover, the overhead of FLY with respect pure Java code is negligible. The overall billing cost for the larger test set ranges from $\approx 0.18\$$ for the SMP-4 configuration to $\approx 1.56\$$ AWS-64 one. The Java pure implementation cost is $\approx 1.12\$$. Overall the FaaS backend always provides a speed up to the performance that ranges from $17\times$ (k-NN with 4000 instances) to $160\times$ (k-NN with 24000 instances).

Speed-up vs cost comparison

Table 2.5 provides a comparison in terms of the ratio between the speed-up obtained and its extra cost, respect to the cost of the environment used for the pure Java implementation. The table shows that, except for a small number of instances, the ratio is always greater than 1 meaning that the benefit of using FLY in terms of speed-up is always greater than their extra costs.

2.5 Exploring Function-as-a-Service for Optimization via Simulation using FLY

Simulation models are the most used and expressive way to analyze real-world systems that are too complex to be studied analytically or too risky and expensive to be tested experimentally [38]. A simulation model is a complex mathematical model characterized by several parameters that define its characteristics and behaviors. One of the significant problems in model design is finding the configuration of parameters that makes the simulation execution as meaningful and realistic as possible. Over the years, the scientific community studied different techniques and methodologies to face-off this problem. Optimization via Simulation (OvS) is enjoying the most success, both in terms of performance and accuracy. OvS is a computationally expensive process because both the number of parameters and the values each parameter may assume are generally vast. Moreover, for each configuration, a simulation must be run, resulting in tons of executions.

This work aims to present a methodology that allows to fully exploit Serverless Computing, also called FaaS, in the OvS processes, claiming to be flexible, easy to use, and cost-efficient. The approach stems from the OvS process's subdivision into two main phases: parameters optimization and simulation execution. According to this, we identified two fundamental components that are the optimizer and the executor. While the optimizer takes care of the optimization process, the executor is responsible for the simulation execution, which can be largely improved using FaaS. One of the key benefits of the proposed methodology is

the simplified pattern that it uses. Identifying different components allows us to use the most appropriate based on the context and the application. Moreover, the integration of Serverless Computing inside the OvS process removes the need to configure the system that runs the simulation, speed-up the execution, and facilitates the developer's job.

We show the advantages of the proposed methodology through a real case study. Specifically, we used OvS techniques to solve the Customer Allocation (CA) problem. The CA problem consists of planning a distribution network that maximizes customer satisfaction and, at the same time, minimizes the cost and the time of a product's transportation. To solve this problem, we used the simulation optimization OPTNet framework [75], developed by ACTOR company [76] as the optimizer and an SC workflow written in FLY language [40, 41] as the executor.

2.5.1 Serverless computing methodology for optimization via simulation

OvS consists of an iterative process that starts with the execution of a simulation with some initial parameters. A specific function evaluates the results obtained by this execution, and then the process uses the feedback to tune the parameters or select new ones for the next simulations. The entire OvS process can be divided into two principal phases: the parameters optimization and the simulation execution. The latter is a compute-intensive and time-spending task because of the high number of configurations generated in each iteration, along with the time needed for the execution itself.

The optimization model views the simulation as a black box and only provides them the parameter values. Then it uses the simulation results for the optimization phase. [77].

In this work, we propose a methodology that stems from the division mentioned above, identifying two main components: the optimizer and the executor. The optimizer handles the optimization process, it generates the input parameters for the simulation execution based on the previous iteration results and manages the termination. On the other hand, the executor takes care of the execution of the simulation. It uses the configuration provided by the optimizer as the parameter values

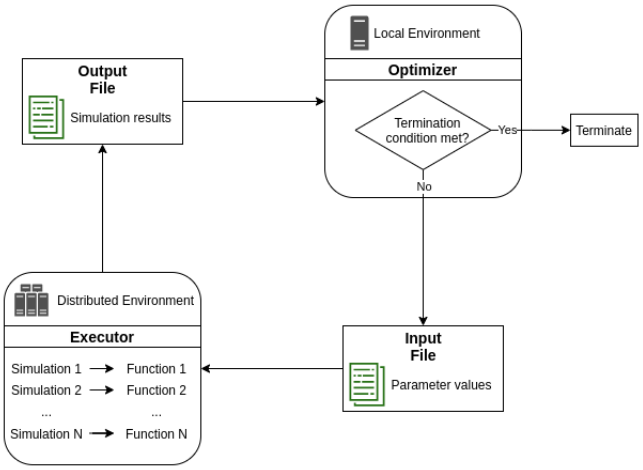

Figure 2.5: Optimization via Simulation process over Function-as-a-Service architecture

and then collects the results. The first advantage of our methodology is that the two components do not need to communicate directly. Instead, the data is exchanged using simple files avoiding any compatibility and dependability issues. Specifically, the optimizer writes several simulation configurations on the input file. The executor reads the input file, uses it to set up the simulations, and writes the results on the output file. The optimizer reads the output file using the data for the optimization phase and starts another iteration if needed. The entire cycle is shown in 2.5.

The components' independence makes the methodology flexible and adaptable, allowing any technology for its implementation. The choice is totally based on the context, and the only needed characteristic is the ability to read and write files, clearly a trivial one.

Workflow distribution approach

The critical part of an OvS process is the simulation phase because each configuration requires a simulation, demanding many computing resources and time. The executions' independence permits to parallelize the workflow to decrease the time needed but, on the other hand,

involves the necessity of a powerful computing system. The integration of Serverless Computing removes this necessity and fully devolves the management to the cloud provider [78]. Figure 2.5 depicts the parallelization technique used. The optimizer provides several configurations, and the executor can run multiple simulations in parallel, assigning a function to each configuration. The function-based development model allows to increase the system scalability, optimize the workload, the execution time, and the costs. However, building an application using Serverless Computing is not that simple, especially when the developer is a domain expert and not a computer scientist. The development of the executor component using FLY permits to overcome these difficulties. First of all, as specified by Section 2.2.1, FLY abstracts any interactions with the cloud provider, providing an easy and effective language independent from the specific cloud provider. In the second place, the FLY program structure makes simple the management of the different simulation functions and file reading and writing. Specifically, a FLY program is composed of the main program and one or more FLY functions. The main program takes care of the whole orchestration managing the initialization of the cloud environments, functions deployment, and the functions' invocations together with results collection. File reading and writing happens here. A FLY function corresponds to a Serverless function and can implement the simulation logic. The proposed methodology offers even more benefits using FLY as the language for the executor. Indeed the simulator designer does not need to have high programming or cloud computing experience. Instead, he can quickly write its simulation with a simple programming language. FLY Language allows the designer to focus on the application logic besides the technicalities needed to execute the application on the cloud, granting to exploit the Serverless Computing power. Even more, a FLY program can be run on several cloud providers, in a multi-cloud manner, with minimal changes to the code.

2.5.2 Use case: Customer Allocation problem

In this section, we describe an implementation of an OvS process for solving the Customer Allocation problem using the methodology de-

scribed in Section 2.5.1. The use case aims to explain, evaluate, and demonstrate the use and the benefits of our approach. Specifically, we implemented an architecture composed by the SO framework OPTNet [75], developed by ACTOR company [76], as the optimizer and a Serverless workflow written using the FLY language [40, 41] as the executor. We tested it using Amazon Web Services (AWS) as the cloud environment to evaluate scalability, efficiency, and costs obtained using our methodology.

OPTNet - simulation optimization framework. OPTNet is a flexible Simulation Optimization framework written in Java and developed by the ACTOR company in 2006. It allows the design, the simulation, and the analysis of a supply chain, its flows, and inventory for cost-cutting purposes. OPTNet aims to maximize computational performances reducing hardware costs. OPTNet can deal with large corporations' business complexity, but it can also scale down for mid-size companies' needs. The design of OPTNet is specific to manage large quantities of data, easily integrated with ERP systems. It can be installed on both on-premise or on a cloud environment, implementing the Software as a Service model. The core of OPNet is a math-optimization engine for transport planning, fleet scheduling, and vehicle routing that provides different supply chain and transportation solutions. Some examples are distribution from a single warehouse, last-mile, and door-to-door distribution for postal and many others. It is also user-friendly, including a web-based and mobile user interface [75].

Customer Allocation problem. One of the most critical activities in logistics is the planning of the product distribution network. The objective is to minimize the total logistical cost, expressed as the sum of activation and deactivation, storage, transport, and customer service costs. The logistics network can be seen as a network of nodes that are production and distribution sites and customer areas, connected by edges, describing idealized transport routes between pairs of nodes. The solution is represented by defining the customer nodes that each production or distribution node must serve. The entire process is called Customer Allocation (CA) problem [79] and represents one of the most

important applications of optimization, in which the goal is to find the best routes for a fleet of vehicles visiting a set of locations, where the best means routes with the least total distance or cost.

Given the CA problem's inherent difficulty, the approach taken is to divide it into several instances of the well-known Capacitated Vehicle Routing Problem (CVRP), each of which includes only one production or distribution site. Once the CVRP solutions are obtained, they are used to reconstruct the complete CA solution. The Capacitated Vehicle Routing problem involves several vehicles with a limited carrying capacity that need to pick up or deliver items at various locations. The items have a quantity, such as weight or volume, and the vehicles have a maximum capacity that they can carry. The problem is to pick up or deliver the items for the least cost while never exceeding the vehicles' capacity. CVRP is known to be an NP-hard problem, so it is difficult to solve this problem directly when the problem size is large.

Implementation details. The solution for the Customer Allocation problem is built using a Simulation Optimization approach. Specifically, a Genetic Algorithm (GA) is implemented. Initially, the algorithm generates naive solutions without optimizations and tries to improve them at each iteration through casual modifications.

Clearly, the mechanism described above is well suited to be implemented with the proposed methodology exploiting the possible subdivision in optimization and execution phases. In detail, the optimizer generates different solutions, i.e., a group of parameter values, and writes them on the input file. The executor reads the input file and uses the parameters to run the simulations. Then the results are collected and written on the output file. The optimizer uses the output file to tune the parameters and start a new iteration.

The implementation uses OPTNet as the optimizer while a FLY program plays the role of the executor. FLY main program takes care of the orchestration part: reads the input file and launches a specified number of functions using the configurations read. A FLY function implements the simulation logic, i.e., the CVRP program. Upon functions' completion, the main program collects all the results and writes the output file. Figure 2.6 shows the CVRP FLY program execution's

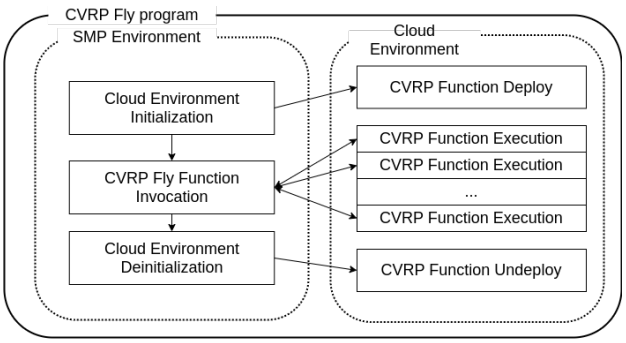

Figure 2.6: FLY execution workflow for Capacitated Vehicle Routing problem.

workflow at a very high level.

The FLY Language characteristics provide many advantages to this kind of workflow because the initialization of the cloud environment, the deployment of the function, and the actual concurrent functions invocations are transparent from the user point of view or, at least, easily implemented thanks to specific language features.

Listing 2.3: CVRP FLY Program.

```
var local = [type="smp", nthread=4]
var aws = [type="aws", profile="default", access_key =
    "", secret_key="",
region="", language="python3.6", concurrency=1000,
    memory=3000, seconds=300]
/* omitted code for space reasons */
func vrpc(data) {
    /* omitted code for space reasons */
    for instance in data {
        for d in demands_it {
            if (d.v > vehicle_capacity){
                split_demands[d.k] = instance[3] as
                    Integer + j
                j++
            }
        }
        /* omitted code for space reasons */
        for d in demands_it {
```

```
            native <<<
            if d['k'] in split_demands.keys():
                demands[i] = vehicle_capacity
                demands[split_demands[d['k']]]=(d['v']-
                    vehicle_capacity)
            else:
                demands[i] = d['v']
            >>>
            i++
        }
        /* omitted code for space reasons */
func estimate() {
    for i in [0:1000] {
        var e = ch ? as Object
        for r in e{
            println("Instance Name: "+r.k)
            println(r.v)
        }
    }
}
/* omitted code for space reasons */
fly vrpc in test on aws thenall estimate
```

The functions developed in FLY are based on the CVRP algorithm provided by the *Google Optimization tool* [80], written in C++ but also supplied with a Python wrapper used in our experiment. A brief snippet of the CVRP FLY program can be seen in Listing 2.3, while the complete code is available on GitHub [81].

Evaluation

Experiment settings. The approach described above, composed of OPTNet as the optimizer and a FLY program as the executor, is being tested using AWS services to evaluate scalability, efficiency, and costs. A benchmark is proposed between the proposed methodology and the sequential version of the same solution, developed using only OPTNet. Specifically, we tested several GA configurations using one iteration loop with 10 facilities and 15000 customers, varying the number of CVRP instances generated, starting from 2468 up to 12340. All the experiments have been performed on an AWS EC2 *m4.16xlarge* machine with 64 cores and 256GB of memory. The FLY program has been

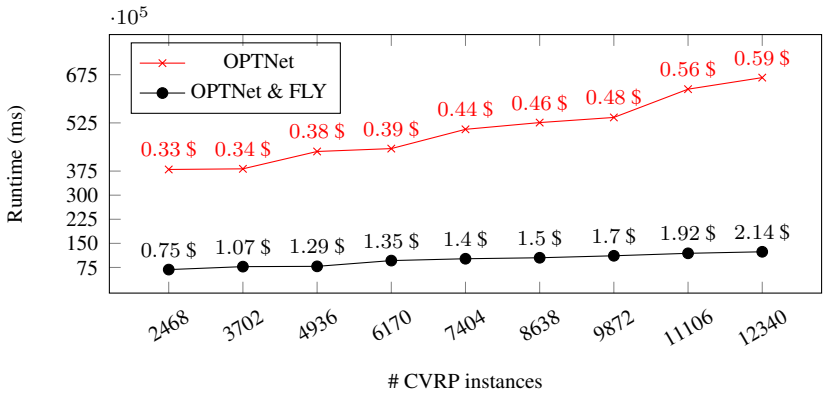

Figure 2.7: OPTNet vs OPTNet & FLY running time and price comparison.

configured to execute 1000 concurrent functions, each with at most 3GB of memory using Lambda functions, the AWS SC service. It is important to note that we used the same AWS EC2 instance for running both the experiments.

Results. Figure 2.7 depicts the obtained performance in our experiments, varying the number of CVRP instances computed. The x-axis shows the number of CVRP instances, while the y-axis shows the total running time in milliseconds. Furthermore, near each point, the corresponding price in dollars ($) is shown.

The benchmark analysis reveals that the proposed methodology provides better performance, i.e., the shortest execution time, as expected. However, the most relevant outcome regards scalability. Indeed, as the number of CVRP instances increases, the total execution time undergoes minimal growth. On the contrary, looking at the performance of the solution implemented with the sequential version of OPTNet, the line rises steeply. This result demonstrates that the methodology allows to fully exploit the computational power provided by Serverless Computing, granting high scalability.

Concerning the costs, the use of AWS Lambda service makes our approach more expensive, as we expected because of the Cloud billing model.

Wrapping up, the combined use of OPNet and FLY results in a speed-up consistently greater than $\times 5$ with respect to the sequential version of OPTNet, while the price increases at most 3 times. We can assert that solving the CA problem using our methodology provides scalability and efficiency, significantly reducing the whole OvS process's computational time. Furthermore, the integration of FLY Language for the development of the executor component brings even more advantages in terms of usability and ease of use.

Chapter 3

Agent-based Simulation

3.1 Introduction

SC uses computer simulation to investigate solutions and study complex real-world problems. Identifying fundamental rules that govern complex systems and developing predictive models to describe the evolution of real phenomena are challenging tasks that can improve the design of approaches and methodologies in many research fields from social sciences to the life sciences, from economics to artificial intelligence [82]. The analysis of real systems has revealed several interesting emergent behaviors both in terms of structural features [83] and dynamic behaviours [84]. However, a full understanding of the dynamic behavior generated by complex systems is extremely hard and requires innovative study methodologies. Computational scientists have proposed the analysis of these phenomena through the exploitation of simulations based on Agent-Based Model (ABM). The success of computational sciences has led to increasing demand for computation-intensive software implementations. Hence, the need to improve the performance of ABMs simulations - successfully adopted in many sciences [85] - in terms of both size (number of agents) and quality (complexity of interactions). Complex ABMs very often require the continuous computation of global data during the simulation [86]. In such cases, the problem consists of ensuring good performance and a high-level of effectiveness in simulation modeling.

However, frameworks for distributed simulations are not able to compute global information efficiently (for instance, the total number of agents that satisfy a given property) [53]. The computation of a global parameter represents a bottleneck for distributed simulations, which jeopardizes the performance due to the communication overhead. In such cases, the use of a parallel and/or sequential simulation framework provides better performances [87]. Moreover, to achieve performance, distributed simulations often require expensive hardware that is usable only by distributed computing experts. Providing efficient and effective software for developing ABM simulations in sequential computing allows the user to effortlessly execute simulations. This makes simulations more suitable for a "what-if" scenario, where the user needs to frequently change the simulation parameters and rapidly observe the results. High-performance ABM simulations are built upon performance-critical operation, and interactions exhibit multiple levels of concurrency. Implementing an efficient framework for the development of ABM simulations is extremely challenging, and the choice of the implementation language is a crucial aspect to consider. It is common to use a language like C to gain performance, as it enables the programmer to exploit low-level memory operations (e.g., deallocating memory) thanks to its low level of abstraction (e.g., no object-oriented support). On the other hand, the usage of such languages turns out to be quite difficult, especially for domain experts with limited knowledge of computer programming and systems.
In this work, we exploit Rust as a programming language for the next generation of ABM simulation. Rust is a multi-paradigm system programming language with performance comparable to C. Its main feature lies in its memory model, designed to be both memory and thread-safe. This aspect can be recognized as the core advantage of using Rust over languages like C++ and Java as it allows the user to write correct code, particularly in the presence of concurrency and parallelism. We will describe Rust's key concepts in Section 3.1.2.

3.1.1 Agent-based models

Agent-based models are a class of computer models in which entities (referred to as agents) interact with each other and their local environment to observe the rising of complex macro-phenomena.

There is no universal definition of agents. The diversity of their application makes it difficult to extract characteristics common to all possible types of agents. From the modeling standpoint, some features are common to most agents. These are briefly presented:

- **autonomy**: can make independent decision processing information and exchanging it with other agents;
- **heterogeneity**: it is possible to have different agents in the same ABM. For example, if we are modeling a city, we can have agents representing people and other agents representing cars, buildings, animals, and all other objects that we can imagine. Obviously, each of these types of agents has it's characteristics and behavior;
- **active**: because agents exert independent influence in a simulation.

This list is not exhaustive or exclusive because agents may possess different characteristics within an application, and for some applications, some characteristics are more important than others. Each agent in an ABM has rules that affect their behavior and relationship with others and/or their environment. These rules are typically based around *if-else* conditions with agents carrying out an action once a specified condition has been satisfied. More generally, rules can be any algorithm that expresses an agent's response in terms of other agents and the surrounding environment. In ABM, as the agents, also the environment plays a critical role. The environment defines the space in which agents operate, serving to support their interaction with the environment and other agents. An environment can be abstract (empty space) or a Geographic Information System (GIS) layer representing real geographic information. It is common to find ABMs with more than one environment to map several layers of information related to the real environment. Environments can be static or dynamic, changing during time.

There are three claimed advantages of the agent-based approach:

- ABMs capture emergent phenomena in a bottom-up approach.
- ABMs provide a natural environment for the study of certain systems.
- ABMs are flexible, particularly to the development of GIS models.

3.1.2 Rust background

Rust is a multi-paradigm system programming language, designed initially at Mozilla Research in 2009. Rust first stable release was launched in 2015, and since 2016 it figures as the *most loved programming language* in the yearly Stack Overflow Developer Survey. The Rust compiler is a free and open-source software dual-licensed under the MIT License and Apache License 2.0. The reasons why Rust is so widely used must be sought in its design principles. Rust, in fact, guarantees both memory and thread safety, thanks to its rich type system and its ownership model.

Ownership. Ownership is Rust's central feature: memory is managed through a system of ownership that the compiler checks at compile time. This means that there is no need for a garbage collector that constantly looks for no longer used memory. In addition, Rust programmers do not have to explicitly allocate and free the memory. Ownership is translated into practice with the following concept: each value in Rust has a variable called its *owner*. The owner is unique, and when it goes out of scope, the value is dropped.

References and borrowing. These two concepts are strictly related to Rust's ownership model. As in other programming languages, a given variable `x` can be passed either by *value* or by *reference*. When a value is passed by reference, it can be passed either by immutable reference using `&x` or by mutable reference using `&mut x`. The `&x` syntax creates a reference that refers to the value of `x` but does not own

it. For this reason, the value it points to will not be dropped when the reference goes out of scope. Similarly, the signature of the function uses `&` to indicate that the type of the parameter `x` is a reference. Using references as function parameters is known as *borrowing*. References, as Rust variables, are immutable by default. If a reference to a variable `x` needs to be modified, it has to be declared as *mutable* using `&mut x`. The benefit of having this restriction is that Rust can prevent data races at compilation time.

Furthermore, the Rust compiler guarantees that *dangling* references, i.e., a pointer that references a location in memory that may have been given to someone else, will never happen. Every reference in Rust has a *lifetime*, which is the scope for which that reference is valid. Most of the time, lifetimes are implicit and inferred, but they must be annotated when the lifetimes of references could be related in a different way. The main aim of lifetimes is to prevent dangling references.

Rust object-oriented programming Rust is a programming language influenced by many programming paradigms, including object-oriented (OO) programming (OOP). Therefore it shares certain common characteristics with OO languages:

- *Rust Objects.* Rust enables the definition of objects using structures, enums, and `impl` blocks. A `struct` is a custom data type that packs together multiple related values that make up a meaningful group. As it happens with `structs`, `enums` can be defined to hold generic data types in their variants. The `impl` keyword is primarily used to define implementations on types.

- *Encapsulation* means that the implementation details of an object are not accessible to code using that object. Rust defines the `pub` keyword to let the programmer decide which modules, types, functions, and methods should be public. By default, anything else is private.

- *Inheritance* is a mechanism whereby an object can inherit from another object's definition, thus gaining the parent object's data and behavior without having to define them again. Rust does not

allow the user to define a struct that inherits the parent struct's fields and method implementations.

- *Polymorphism* means that your multiple objects can be substituted for each other at running time if they share certain characteristics. In Rust, this feature is enabled through *traits*. A `trait` tells the Rust compiler about functionalities a particular type has and can share with other types. Traits can be used to define shared behaviors in an abstract way, in which a type's behavior is defined by the methods we can call on that type.

3.2 Rust-AB: Programming Agent-based Models in Rust

Rust-AB is a discrete events simulation engine designed to be a ready-to-use ABM simulation library suitable for the ABM community. To reduce the learning curve and simplify its usage, we adopted the same modular and standard architectural layout of the Java library MASON, based on the Model-View-Controller design pattern.

More in detail, a MASON simulation is made up by three fundamental players:

- the simulation agents, specified by the Java interface `Steppable`;
- the simulation scheduler, defined by the `Scheduler` object;
- the simulation state, represented by the `SimState` object.

The implementation of a MASON simulation has to extend the `SimState` object, while its agents are represented through a Java class, which implements the `Steppable` interface.

Even though Rust-AB resembles MASON in its architecture, we have re-engineered the simulation engine to exploit Rust's peculiarities. Furthermore, Rust-AB has been designed to provide the programmers an easy and standard simulation framework for developing ABM, thus enabling easier adoption of a new language as Rust.

The current version of the Rust-AB simulation engine library is released under MIT license on a public GitHub repository [88]. Section 3.2.1 describes Rust-AB architectural concepts and functionalities.

3.2.1 Rust-AB architecture

Agent. An `Agent` is the most important concept of Rust-AB. According to the OO model of Rust, an agent is a `trait` of a Rust `struct`, which means that every Rust struct implementing the trait `Agent` is considered a simulation agent. Similarly to the MASON toolkit, the Agent implementation must provide a `step` method where the agent logic should be placed.

Schedule. Being Rust-AB a discrete event simulation engine, the `Schedule` is its core object as it provides all functionalities to manage a simulation according to event-based scheduling. It provides the same interface defined by MASON.

The simulation proceeds by scheduling the agents time-by-time. A schedulable agent is a Rust struct that implements the Rust-AB trait `Agent` and the Rust trait `Clone`. To obey the Rust programming model, the scheduler has to mandatory clone the agents before each simulation step.

The scheduler works as a priority queue (FIFO), where the agents are sorted according to their scheduled time and a priority value - an integer. The simulation time - a real value - starts from the scheduling time of the first agent. At each discrete simulation step, all agents scheduled in the current simulation time perform a simulation step according to their scheduling priority. The scheduler provides two scheduling options:

- `schedule_once` inserts an agent in the schedule for a specific simulation step. The scheduling time and the priority are given as parameters. The priority is used to sort all agents within the same simulation time.

- `schedule_repeating` acts like *schedule_once*, with the difference that the agent will be scheduled for all subsequent simulation steps.

The schedule provides the `step` method, which allows executing one simulation step. In this way, the programmer can easily design his/her simulation by looping for a certain number of steps or for a given amount of CPU time.

Location2D. `Location2D` is a Rust-AB `trait`, defining a Rust struct exposing a position in a 2-D space. An agent can be placed in a field struct, thus enabling the programmer to easily model agents' neighborhood interactions. Every Rust struct that implements this trait can be placed in a Rust-AB field. A position in a 2-D space is modeled as a Rust-AB struct `Real2D`. A given `Location2D` implementation must provide two functionalities: i) `get_location`, that provides the current position in the space - a Real2D, and ii) `set_location`, which allows to move an object.

Field2D. `Field2D` is a sparse matrix structure modelling agent interactions on a 2-D space. The `Field2D` structure is parameterized on a given type implementing: `Location2D` Rust-AB trait, and Rust `Clone`, `Hash`, and `Eq` (equivalence relation) traits. `Location2D` defines the structure on which the field operates, while the remaining traits allow a more efficient implementation of the field functionalities.

It is worth mentioning that the field structure is useful not only for the agents but for any kind of Rust type that implements the described traits. This designing aspect allows the programmer to model interactions with any kind of simulation environment easily.

The `Field2D` structure provides the following methods:

- `set_object_location` inserts/updates an object in a field in a given position.
- `get_neighbors_within_distance`, returns a vector of objects contained in the circle centered at a given position with

a radius equal to the distance parameter. An optimized radial searching method is used to compute the neighborhood.

- `get_object_location`, returns the position of a Location2D object.
- `get_objects_at_location`, returns a vector of objects stored in a given position.
- `num_objects`, returns the total number of objects stored in the field.
- `num_objects_at_location`, returns the number of objects at a given `Real2D` object position.

Simulation state. `SimulationState` is the state of a Rust-AB simulation. A Rust-AB simulation is composed of an agent definition (i.e., a Rust struct that implements the trait *Agent*), a Rust-AB scheduler instance (declared for the agent implementation), and a set of fields and variables. The simulation logic is implemented in the step function of the agent. For this reason, the programming environment must provide a mechanism to access the simulation state from the agent's step function.

The simulation state is defined using a Rust struct, containing all fields and variables. The simulation state is defined using a Rust struct, containing all fields and variables. To access this struct, the programmer must declare the struct itself as a local variable inside the simulation's main() function and give it as an argument to the agents' step function. This procedure is better described in Section 3.2.2.

Limitations. The main design limitations of Rust-AB are due to the basic Rust OOP model and its memory model.

The first limitation concerns the multi-agents capabilities of Rust-AB: the current version of Rust-AB does not support multiple definitions of an agent. Nevertheless, it is still possible to implement a multi-agents model by defining different behaviors in the same agent

definition. The second limitation lies in the fact that the field environment can only accommodate objects of the same type. To model interactions between objects of a different type, it is necessary to use multiple field environment instances (one for each type).

3.2.2 A case study: the *Boids* simulation

To analyze the effectiveness and efficiency of Rust-AB, we implemented a well-known ABM on which we performed several benchmarks varying the model scale parameters. The performance of Rust-AB has been compared against the MASON toolkit running the same ABM.

We developed the Boids model [89] by C. Raynolds (1986), which is a steering behavior ABM for autonomous agents simulating the flocking behavior of birds. The agent behavior is derived by a linear combination of three independent rules:

- *Separation*: steer in order to avoid crowding local flock-mates;
- *Alignment*: steer towards the average heading of local flock-mates;
- *Cohesion*: steer to move towards the average position (centre of mass) of local flock-mates.

We developed the Rust-AB Boids model following the same strategy adopted for the *Flocker* MASON simulation, which implements the same model. First, we defined the agent code and its logic by implementing the `Agent` trait. Then, we defined the simulation state by providing the simulation parameters and the environment definitions. Finally, the main simulation function is defined, where the scheduling policy for agents and the fields initialization are provided.

Agent definition

A Rust-AB agent is a struct containing all the local agent data. For our purposes, we defined a new struct named *Bird* that emulates the concept of a bird in a flock. As stated in Section 3.2.1, a Rust-AB agent has to implement the traits `Agent`, `Eq` and `Hash`.

According to the model specification, at each simulation step, every agent has to compute three steering rules according to the position of its neighboring agents. For this reason, all the agents are placed in a Rust-AB *Field2D* environment. As a consequence, the agent definition must implement the trait *Location2D*, as well as the traits *Clone* and *Copy* (that can be automatically computed using the Rust macro `#derive[(_)]`). The steering behavior model can be implemented by storing the position of the agent in the previous and current simulation steps. The agent position can be modeled using a `Real2D` Rust-AB struct. To easily develop the trait *Hash*, a unique identifier is stored in the agent. Listing 3.1 shows the Rust-AB agent struct definition.

Listing 3.1: Rust-AB Agent.

```
1 #[derive(Clone, Copy)]
2 pub struct Bird{
3     pub id: u128,
4     pub pos: Real2D,
5     pub last_d: Real2D,
6 }
```

The agent logic is defined in the step function. We designed the agent logic using three sub-functions defined in the agent implementation. Listing 3.2 shows the agent implementation code. Lines $1-8$ define the object `Bird` by providing a constructor and three functions: `avoidance`, `cohesion`, and `consistency`, which implement the steering model rules. Each function takes as input parameter a reference to a vector of agents (the current agent neighbourhood) and returns a new `Real2D`, which is the force computed according to the position of flock-mates. Lines $10-13$ show the implementation of the `Location2D` trait, which enables to place the agent in the `Field2D` environment. Lines $15-24$ define the implementation of the traits `Hash` and `Eq`. Lines $29-51$ implement the agent `step` function describing the agent logic, which simulates the steering behavior of the model. The agent computes its neighborhood (line 30) and, using the sub-functions, evaluates its new position. The computed position is then used to update the status of the environment (line 50).

Listing 3.2: Rust-AB Agent Implementation.

```
impl Bird {
    pub fn new(id: u128, pos: Real2D, last_d: Real2D)
        -> Self {
        Bird {id, pos, last_d}
    }
    pub fn avoidance (self, vec: &Vec<Bird>) -> Real2D
        {..}
    pub fn cohesion (self, vec: &Vec<Bird>) -> Real2D
        {..}
    pub fn consistency (self, vec: &Vec<Bird>) ->
        Real2D {..}
}

impl Location2D<Real2D> for Bird {
    fn get_location(self) -> Real2D { self.pos }
    fn set_location(&mut self, loc: Real2D) { self.pos
        = loc; }
}

impl Hash for Bird {
    fn hash<H>(&self, state: &mut H) where H: Hasher,
    { self.id.hash(state); }
}

impl Eq for Bird {}

impl PartialEq for Bird {
    fn eq(&self, other: &Bird) -> bool { self.id ==
        other.id }
}

impl Agent for Bird {
    type SimState = BoidsState;

    fn step(&mut self, state:&BoidsState) {
        let vec = state.field1.
            get_neighbors_within_distance(self.pos,
            10.0);

        let avoid = self.avoidance(&vec);
        let cohe = self.cohesion(&vec);
        let rand = self.randomness();
```

```
35          let cons = self.consistency(&vec);
36          let mom = self.last_d;
37
38          let mut dx = COHESION * cohe.x + AVOIDANCE *
                avoid.x + CONSISTENCY * cons.x + RANDOMNESS
                * rand.x + MOMENTUM * mom.x;
39          let mut dy = COHESION * cohe.y + AVOIDANCE *
                avoid.y + CONSISTENCY * cons.y + RANDOMNESS
                * rand.y + MOMENTUM * mom.y;
40
41          let dis = (dx * dx + dy * dy).sqrt();
42          if dis > 0.0 { dx = dx / dis * JUMP; dy = dy /
                dis * JUMP; }
43
44          self.last_d = Real2D { x: dx, y: dy };
45          let loc_x = toroidal_transform(self.pos.x + dx,
                WIDTH);
46          let loc_y = toroidal_transform(self.pos.y + dy,
                WIDTH);
47
48          self.pos = Real2D { x: loc_x, y: loc_y };
49          drop(vec);
50          state.field1.set_object_location(*self, Real2D
                { x: loc_x, y: loc_y });
51      }
52  }
```

Model definition

We define the Boids simulation state by declaring a new struct `BoidsState`. Listing 3.3 shows the code of the *BoidsState* struct (lines $1-9$). According to the model and the agent definitions, we defined the agents' interactions through the `Field2D` environment. For this reason, the state struct contains only a *Field2D* declaration.

Listing 3.3: Rust-AB Simulation State.

```
1  pub struct BoidsState {
2      pub field1: Field2D<Bird>,
3  }
4
5  impl BoidsState {
```

```
6       pub fn new(w: f64, h: f64, d: f64, t: bool) ->
            BoidsState {
7           BoidsState { field1: Field2D::new(w, h, d, t),
                }
8       }
9   }
```

The main simulation function is shown in Listing 3.4. At line 3, a new Rust-AB `Schedule` is defined, while at line 5 the local variable state is declared, this models the simulation state. From line 6 to 13 a given number of agents are randomly initialised, placed in the `Field2D` (line 11), and scheduled using the `schedule_repeating` method (line 12). At line 15 the schedule step is called for a given number of times.

Listing 3.4: Rust-AB Main Simulation Function

```
1   fn main() {
2       let mut rng = rand::thread_rng();
3       let mut schedule: Schedule<Bird> = Schedule::new();
4
5       let mut state = BoidsState::new(WIDTH, HEIGTH,
            DISCRETIZATION, TOROIDAL);
6       for bird_id in 0..NUM_AGENT {
7           let r1: f64 = rng.gen();
8           let r2: f64 = rng.gen();
9           let last_d = Real2D { x: 0.0, y: 0.0 };
10          let bird = Bird::new( bird_id, Real2D {x: WIDTH
                * r1, y: HEIGTH * r2,}, last_d,);
11          state.field1.set_object_location(bird, bird.pos
                );
12          schedule.schedule_repeating(bird, 0.0, 0);
13      }
14      for _ in 0..STEP {
15          schedule.step(&mut state);
16      }
17  }
```

Results

We performed several tests to assess Rust-AB ability to run simulations with different model scale properties. As benchmark simulation, we

used Rust-AB *Boids* compared with MASON and NetLogo *Flockers*. Both simulations implement the same model.

All experiments have been performed on a desktop machine equipped as follow: $1 \times$ CPU i-7-8700T 12×2.40 MHz; 16 GB of RAM; Ubuntu Linux 18.04 LTS; Oracle Java Virtual Machine 1.7; Rust 1.31.

We evaluated several simulation configurations by changing the simulation environment and the number of agents. The experiments were conducted in different settings: i) *constant agent density*, varying both numbers of agents and the dimensions of the simulation field; ii) *constant field size*, chancing only the number of agents; iii) *constant number of agents*, varying only the dimensions of the simulation field. The agent density of a simulation field can be easily computed by $\frac{w \times h}{A}$, where w and h denotes respectively the width, and the height of the simulation field and A denotes the number of agents.

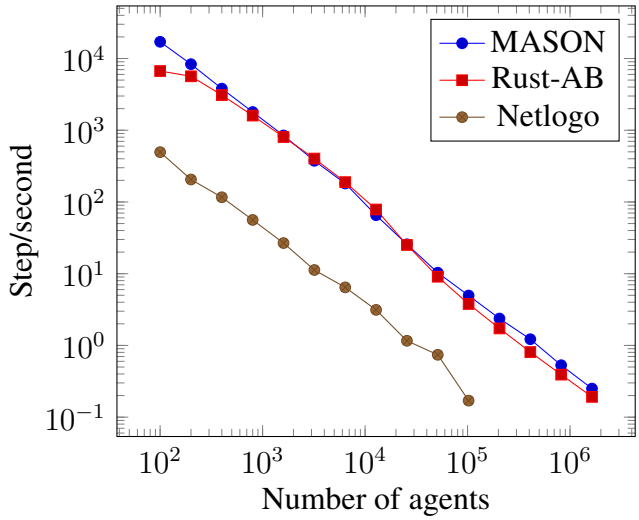

Figure 3.1: Rust-AB performance comparison. Constant Agents Density

Constant agent density. In these experiments, we tested the simulation engine's ability to simulate an increasing number of agents while maintaining the same scaling proprieties. Results are depicted

in Figure 3.1. The x-axis shows the number of agents - ranging from 100 to 1638400 - while the y-axis shows the performance in terms of average simulation step per second (log scale), during a 10 minutes of simulation. As shown in the plot on the left, Rust-AB and MASON obtain almost the same performance when the agent density is constant. On the other hand, the performance of NetLogo is always significantly smaller than the other simulators. It is worth highlighting that NetLogo was not able to execute the last three experiments due to memory requirements.

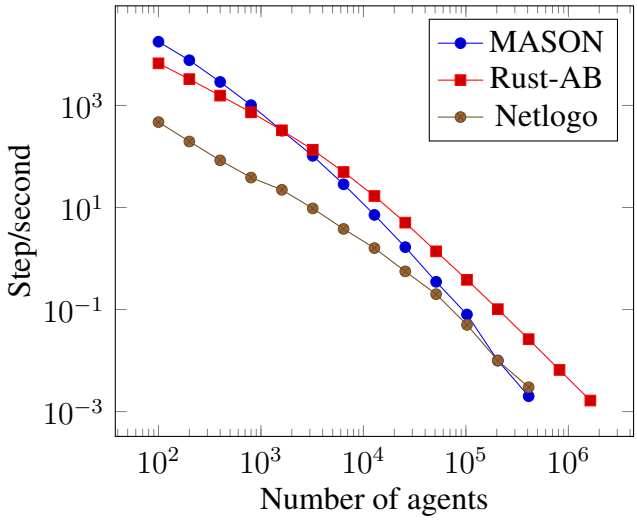

Figure 3.2: Rust-AB performance comparison. Constant Field Size

Constant field size. We evaluated the simulation engine's ability to simulate an increasing number of agents lying on a field of fixed size (200×200). Increasing the number of agents implies increasing the agent density and, consequently, the computational cost of computing the neighborhood and the new position of each agent. Figure 3.2 shows how Rust-AB simulation scales much better than MASON and NetLogo simulations. In particular, MASON achieves the same performance of Rust-AB when the number of agents is low. However, at the increasing of the number of agents, and consequently of the total computational

load, Rust-AB performs better. This behavior may be due to Rust's ability to efficiently manage a high computational workload, mainly thanks to its memory system. Further analysis is needed to assess this hypothesis. On the contrary, NetLogo initially provides the worst results, while at the end, its performance is comparable to MASON. Again it is worth mentioning that neither MASON and NetLogo were able to execute the last two simulation configurations.

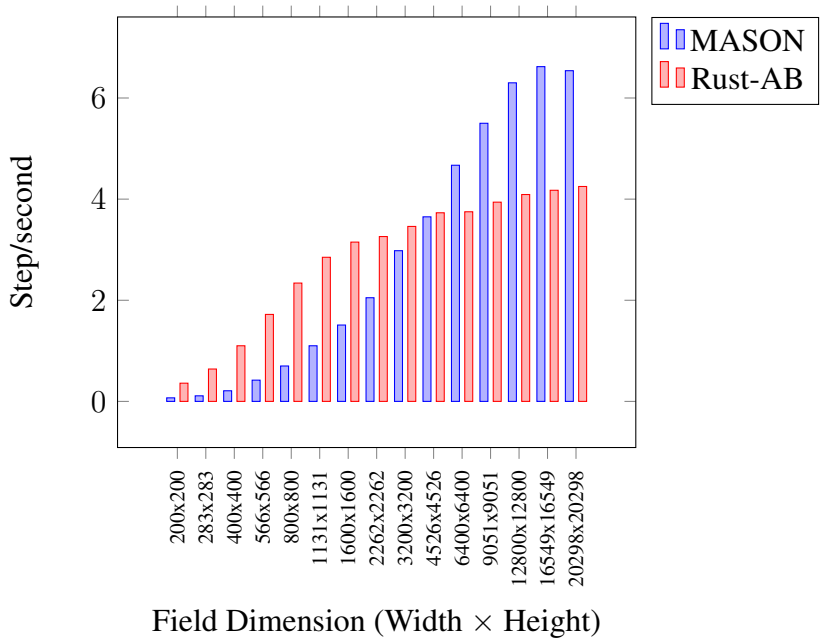

Figure 3.3: Rust-AB vs MASON: Constant Number of Agents.

Constant number of agents. With these tests, we evaluated the simulation engine's ability to simulate a constant number of agents (102400) varying the field dimension. Figure 3.3 presents the results. The x-axis describes the field size - ranging from 200×200 to 20298×20298 - while the y-axis represents the performance obtained in terms of average simulation step per second. As shown in the plot, Rust-AB outperforms the MASON until the density of the field becomes small enough to decrease the total computation load. These results can be

motivated by the same explanation given in the previous paragraph. When the density of the simulation field is small, the MASON library outperforms Rust-AB slightly due to the cost of double buffering used in the Rust-Ab Field2D implementation. For this experiment, we do not show the results of NetLogo as it was not able to run the simulation using the experiment configurations.

3.3 Parallelize Rust-AB

The need to have an effective and efficient agent simulation library capable of handling a massive amount of agents drove us to take full advantage of the multi-threaded architecture of modern computing systems. As described in Section 3.2.1 the `Schedule` provides all the functionalities to manage the simulation according to the event-based scheduling. As a result, it is the first place to apply parallel computing techniques. The parallel version of the Schedule uses threads to execute several events (agents) simultaneously. However, multiple agents' parallel execution shows a criticality due to concurrent access to the `Field2D`. The `Field2D` structure comprises several data structures to store agents and objects on the field and efficiently access them, so we need to modify these internal structures to allows thread-safe operation. These data structures are modified to exploit the double buffering and sharding approaches. As the name implies, the double buffering scheme divides the state of a data structure into two, one for reading and one for writing. This division leads to the need to have a point at which these structures' states must be synchronized. Given the stepwise nature of an agent-based simulation, the end of a step represents an excellent synchronization point. The introduction of double buffering improves concurrent access to the `Field2D` field but brought an increase in memory requirements due to double state handling.

Listing 3.5: Rust-AB Schedule.

```
pub struct Schedule<A:'static + Agent + Clone + Send>{
    pub step: usize,
    pub time: f64,
    pub events: Mutex<PriorityQueue<AgentImpl<A>,
        Priority>>,
    pub pool: ThreadPool
}
```

3.3.1 Parallel Schedule

In the parallel version of *Schedule* a `pool` field has been added. This field is a ThreadPool of the Rayon crate.

The `step()` method of the `Schedule` divides the agents to be processed equally into N batches, where N is the number of threads used (the user specifies this value). These batches are then assigned to the threads within the `pool` to be processed.

Each time a thread processes an agent, it also checks if it needs to reschedule it for the next step, maintaining these agents in a list. When a thread has finished processing its assigned batch, it requests the lock on the event queue, reschedules the agents if necessary, and releases it.

Double buffered Field2D

The original implementation of `Field2D` used Hashmaps to store agents and objects inside the field and efficiently access them. In the parallel version of Rust-AB, such HashMaps have been replaced by `DBDashMaps`, a modified version of the concurrent `DashMap` [90] remodeled to take better advantage of the double buffering technique.

The `DBDashMaps` retains the same sharding mechanism as `DashMap` to significantly reduce the contention that occurs when the number of threads begins to grow. In detail, `DashMap` divides the underline HashMap into several shards (`Box<[RwLock<HashMap<K,V,S>>]>`), each of these shards is a HashMap protected by a lock. This division does not affect correctness. When an operation on `DashMap` occurs, a hash function is used to locate the correct HashMap on which

to operate. As long the number of shards is big enough, the probability of two or more threads waiting for the same should be small enough. The problem related to the `DashMap` is that read and write operations must request a lock on a shard to be performed. This is needed for the writing operation, but we can prevent the lock request on the reading operation. In `DBDashMaps`, we applied the double buffering approach to the sharding mechanism used in `DashMap`. Thus, in `DBHashmap`, we have a collection of writing shards synchronized by a lock (`Box< [Mutex<HashMap>] >`) and a collection of reading shards without synchronization (`Box< [HashMap] >`).

The main methods provided by the DBDashMap are as follows:

- `insert()`: to insert new entries into the DBDashMap, operates on the writing shards;
- `remove()`: to remove an entry from DBDashMap, it works on writing shards;
- `get()`: to read an entry in the DBDashMap, it operates on the reading shards, without the need for synchronization;
- `update()`: exchanges shard pointers and clear the writing shard. At the end of the execution of this method, the reading shards will contain all the entries made since the last call of update();

Notice that, in this new version of Rust-AB, the double buffering is exploited both in the sequential executions and in the parallel ones.

3.3.2 Parallel evaluation

We evaluated the parallel version of Rust-AB using the Boids simulation described in section 3.2.2. An interesting aspect is the possibility of using the same simulation code presented in Section 3.2.2, and executing it in a parallel or sequential manner. This ability has been made possible through the Cargo Features mechanism. Listing 3.6 shows the command to run the Boids simulation in parallel by adding the *parallel* feature and specifying the number of threads desired by the *nt* feature.

Listing 3.6: Command to run boids simulation in parallel.

```
cargo run --release --features parallel --example boids
    --nt N
```

Instead, Listing 3.7 shows the command to run the boids simulation in a sequential manner.

Listing 3.7: Command to run boids simulation in sequential.

```
cargo run --release --example boids
```

To evaluate the performance of the parallel version of Rust-AB, we performed a Strong Scalability analysis. All experiments were performed on a c5.24xlarge machine on Amazon EC2 with the following configuration: 96 vCPU; 192 GB of RAM; 10 Gbps of bandwidth; Ubuntu Linux 20.04 LTS, OpenJDK 11; Rust 1.48.

Strong Scalability. Two strong scalabiliy analyses were performed, one using 500000 agents and the other using 1000000 agents by varying the number of threads used from 1 to 96. The results obtained by Rust-AB were compared with ideal values.

Figure 3.4 shows the results obtained by Rust-AB, to run the *Boids* simulation with 500000 agents. The x-axis shows the number of threads used - ranging from 1 to 96- while the y-axis shows the speedup obtained from Rust-AB to perform 50 simulation steps. The figure shows a notable increase of the performances according to the increase of the number of threads, with a speedup of $\times 10$ with 96 processors. Figure 3.5 shows the results obtained with Rust-AB to execute the *Boids* simulation with 1000000 agents. As before, the x-axis shows the number of threads used - ranging from 1 to 96- while the y-axis shows the speedup obtained from Rust-AB to perform 50 simulation steps. From the figure, we can see that although the number of agents is considerably higher, with 96 threads, we manage to obtain a speedup of $\times 14$.

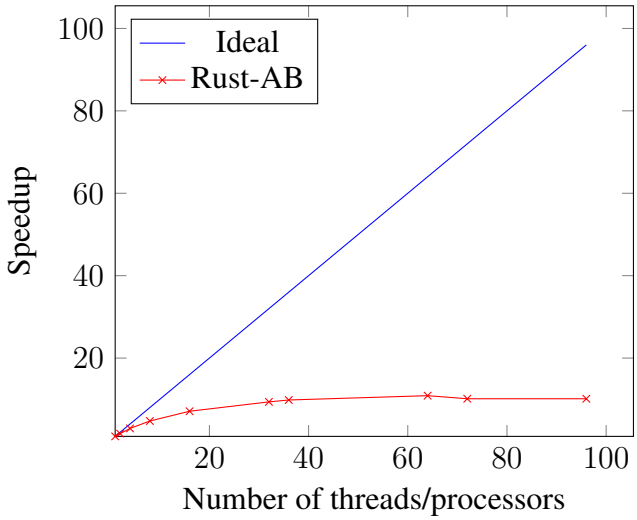

Figure 3.4: Strong Scalability Analysis, Boids Model with 500000 agents for 1,2,4,8,16,32,36,64,72,96 threads.

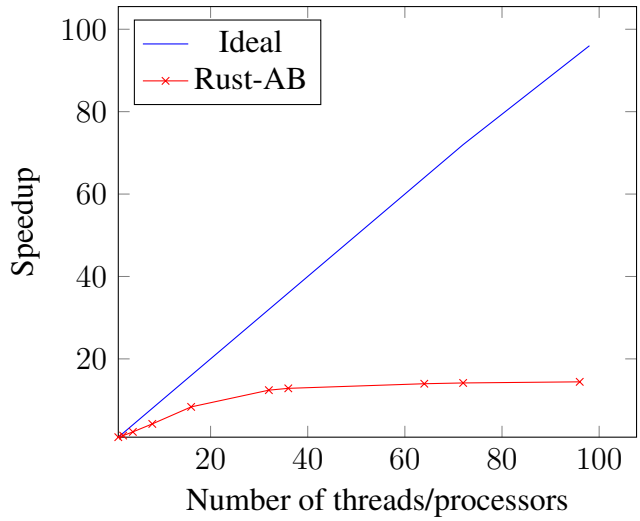

Figure 3.5: Strong Scalability Analysis, Boids Model with 1000000 agents for 1,2,4,8,16,32,36,64,72,96 threads.

Chapter 4

Distributed Simulation Optimization

4.1 Introduction

Complex system simulations continuously gain relevance in business and academic fields as powerful experimental tools for research and management, particularly for Computational Science.

Simulations are mainly used to analyze too complex behaviors to be studied analytically or risky/expensive to be tested experimentally [91, 92].

The representation of such complex systems results in a mathematical model that includes several parameters. To obtain results close to reality, the need to tune a simulation model arises, which results in finding the optimal values of the parameters that maximize the effectiveness of the model. Considering the multi-dimensionality of the parameter space, their heterogeneity, and the irregular and stochastic nature of the objective functions, finding the optimal configuration of the parameters is not an easy task and requires a lot of computational power. Optimization via Simulation (OvS) strategies are designed to ascertain model parameters that minimize (or maximize) specific criteria (one or many), which can only be computed by running a simulation. Although these techniques aim to reduce the computational load and time required, these are still exaggeratedly high. As a result, the need for tools that

leverage the computational power of parallel and distributed systems to improve the effectiveness and efficiency of OvS strategies has become essential.

This chapter discusses methods for exploiting parallel and distributed systems' computational power to improve the efficiency and effectiveness of OvS strategies. Specifically, three frameworks are presented that differ for their underlying computing system architecture adopted:

- *Heterogeneous* - Heterogeneous Simulation Optimization (HSO) framework allow using CPUs and GPUs to execute simulations in a distributed system composed of a heterogeneous node in terms of hardware and software.

- *Homogeneous* - The computing system is composed of homogeneous nodes where are used MASON (simulation library) and ECJ (optimization library) software to elaborate the OvS process.

- *Cloud computing* - The computing system is a MapReduce cluster running over the cloud to elaborate an OvS process to solve the Cruise Itinerary Scheduling problem.

4.1.1 Optimization via Simulation

Optimization via Simulation problem, also called Simulation Optimization (SO), described in [93, 94], is defined as follows

$$\min g(\boldsymbol{x}), \boldsymbol{x} \in \Theta, \tag{4.1}$$

where $g(\boldsymbol{x}) = E[Y(\boldsymbol{x}, \xi)]$ is a single objective function defined by the expected value of a random variable $Y(\boldsymbol{x}, \xi)$. The random variable $Y(\boldsymbol{x}, \xi)$ represents the result of a simulation process, in which $\boldsymbol{x}$ a set of simulation parameters (configuration) and ξ corresponds to the randomness of the process (the random seed). Θ is the set of feasible configurations. The distribution of the variable $Y(\boldsymbol{x}, \xi)$ is unknown a priori, and it is estimated running several simulations with the same configuration $\boldsymbol{x}$ but using different random seeds.

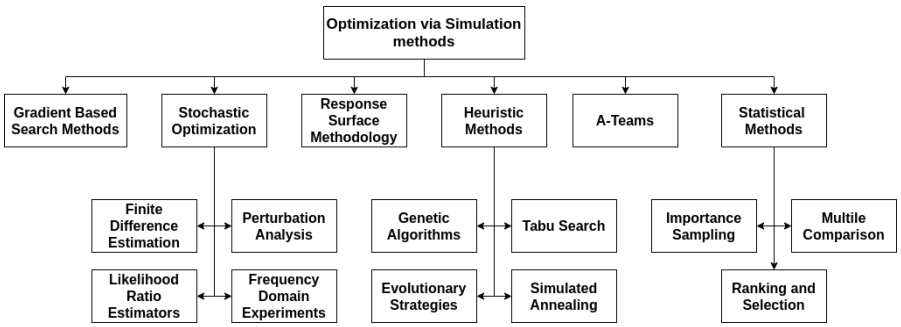

Figure 4.1: Methods in Optimization via Simulation.

Understandably, there are many methods suggested for OvS, and the primary methods are displayed in Figure 4.1. However, most developers preferred heuristic search methods in their solutions. Heuristic search algorithms provide goods and reasonably fast results on a wide variety of problems [93]. The authors mention some important heuristic algorithms. These include genetic algorithms, evolutionary strategies, simulated annealing, simplex search, and tabu search [93].

Optimization via Simulation is an iterative process and typically works as follow:

1. An initial set of parameter values is chosen and one or more replication experiments are carried out with these values.

2. The results are obtained from the simulation runs, and then the optimization module chooses another parameter set to try.

3. The new values are set and the next experiment set is run.

4. Step 2 e 3 are repeated until either the algorithm is stopped manually or a set of defined finish conditions are met.

This general procedure seems to be very clear and simple, but its implementation is much more complicated, as different simulation platforms and selected algorithms have to be used.

4.2 Heterogeneous Scalable Multi-Languages Optimization via Simulation

Heterogeneous Simulation Optimization (HSO) is a framework for developing OvS processes in a heterogeneous computing system. We designed our architecture to exploit the computational power of several different hardware architectures, for instance, General-purpose CPUs and/or GPUs, and exploit different programming languages available on the heterogeneous system.

According to the OvS problem definition, we assume the following:

- simulation process takes more time than the optimization process, that means the computational cost to generate the configurations to be evaluated is significantly lower than the computational cost to compute the objective function (running a set of simulations);
- the number of configurations to be evaluated is large. A huge number of simulation processes are needed (also considering that since simulations are not deterministic, several simulation runs are required for each configuration to be analyzed).

Heterogeneous computing overview. In a Heterogeneous Computing system, multiple, possibly dissimilar, computational machines cooperate to solve a problem. If well configured, such systems allow full utilization of the computational power of each element in the system.

A widely used definition, presented in [95], of the heterogeneous computing network model is a set $H = \{M_1, M_2, \ldots, M_m\}$ of m distinct machines, where each machine can communicate with any other. The model considers any computing paradigm (parallel, sequential, dataflow, etc.). The computation is divided into tasks, and each machine can execute one task at a time. Dependencies among tasks are described through a task graph. A task graph for a particular computation P is a directed acyclic graph $G_P = (V, E)$, whose nodes $V = \{t_1, t_2, \ldots, t_n\}$, are the tasks and whose edges are intertask dependency that is (t_i, t_j) is in E if and only if t_i must be executed before t_j due to data dependencies or tasks synchronization. Each task is

an atomic computation, meaning that when a task is executed on one machine, no communication takes place with any other machine.

A mapping $\Pi : V \rightarrow H$ associates each task to a machine in H. Notice that if Π maps all tasks to the same machine, we have sequential computation. Given a computation P of n tasks on a system H composed by m machines, there are m^n possible mappings.

4.2.1 OvS in a heterogeneous computing model

OvS process is composed of several optimization rounds and is executed using the Master/Worker paradigm [96]. In each round, the Master node invokes the optimizer that generates a set of configurations $\boldsymbol{X} \subseteq \Theta$ to be analyzed according to the results of previous simulations. The system at this point builds a mapping Π, that assigns a subset $X_j \subset \boldsymbol{X}$ to each machine $M_j \in H$. Each machine $M_j \in H$ executes for each element in the subset X_j a certain number of simulations using CPU or GPU (according to their hardware and software capabilities). When a machine ends its computation, it sends back to the Master node the results, and the process starts again until the optimizer decides to end the process.

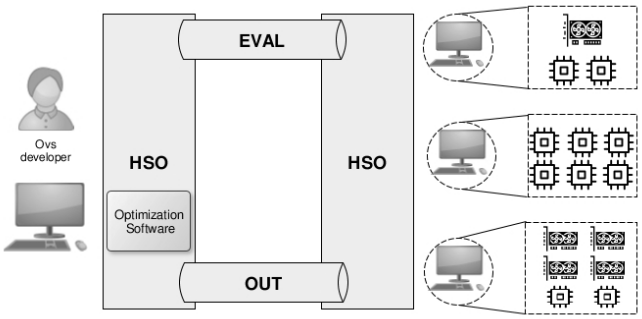

Figure 4.2: HSO Architecture.

Figure 4.2 depicts the proposed architecture named Heterogeneous Simulation Optimization (HSO). HSO is based on the Message Passing paradigm. The Master of the system builds two different messages

queues named EVAL and OUT. In the EVAL queue is sent, by the optimizer process, the set of configuration to be evaluated $\boldsymbol{X}$. In contrast, the machines send the results of the simulations (the value of Y) in the OUT queue. The system takes the values in the queue EVAL and generates the mapping Π, according to the underlying set of machines H. The optimizer is executed on a machine that is not involved in the computation. When the optimizer process needs to execute the evaluation of some configurations $\boldsymbol{X}$, it sends to HSO a message containing the values that will be processed. The computational machines use the simulation software provided by the OvS developer and written for both CPU and/or GPU to run the simulation with the chosen parameters.

The HSO framework can change the mapping Π for each OvS round according to the execution time of the previous optimization round or optimize some computational criteria. We plan to introduce learning algorithms to compute an optimal mapping for the underlying heterogeneous system. This could also be useful in a Cloud Computing scenario, in which we could change the number of CPUs dynamically and/or GPUs involved in the OvS process. Dynamically changing the dimension of the system H, we can optimize the total cost (on the Cloud Computing infrastructure) of the computation or the completion time of an optimization round.

4.2.2 Heterogeneous Simulation Optimization (HSO) framework

HSO was developed in the C language, and its execution model follows the Master/Worker paradigm. The software is fully implemented, and it is available on a public GitHub repository[1]. We used OpenMPI [97] library, an implementation of the standard Message Passing Interface (MPI), for its features in managing the processes spreading over the system. MPI is also used to exchange information between the master node and worker nodes about the number of CPUs and GPUs available on each worker.

[1]Heterogeneous Scalable Multi-Languages Optimization via Simulation GitHub repository, https://github.com/isislab-unisa/hso

HSO is designed to support two levels of heterogeneity. The first is the hardware heterogeneity. The second level of heterogeneity concerns the ability of HSO to support a wide range of programming languages for both simulation and optimization. We have used the ZeroMQ[1] distributed messaging library for the communication between the master of HSO and the optimizer process. ZeroMQ is an embeddable networking library but could be seen as a simple concurrency framework. The library enables the developer to use in-process, inter-process, TCP, and multicast communication through message passing. For our purposes, ZeroMQ is suitable for heterogeneous computing since it supports many programming languages such as C++, C, Clojure, Erlang, Go, Haskel, Java, Node.js, Objective-C, PHP, Python, Scala, and many others. This allows OvS process developers to write their optimizer or import the optimizer they are used to running in any supported language, providing a high heterogeneity level. HSO is designed for general executable and can run any simulation software. For this reason, the simulations on CPU and GPU are executed by defining two different *Bash* scripts which, respectively, run a simulation on a CPU or a GPU.

The OvS developer needs to load on each worker, in H, the simulation software, and provide a script to be used for starting a simulation. The provided script takes as the first argument the simulation input (a configuration $\boldsymbol{x} \in \Theta$) as a string of values separated by a comma. This approach enables the OvS developer to use any simulation software or library.

Listing 4.1 shows the script that executes a simulation on a CPU using the NetLogo simulator. The script exploits a wrapper tool to execute NetLogo by command line[2]. In this example the simulation takes four arguments (see line 18) that are stored in the variable `param_line` (see line 8). The NetLogo simulation parameters are described in Section 4.2.4.

[1]ZeroMQ distributed messaging library, http://zeromo a70q.org/

[2]High-Performance Dataflow Computing Agent-Based Simulator Wrapper, https://github.com/spagnuolocarmine/swiftlangabm

Listing 4.1: CPU simulation example

```
#!/bin/bash
 set -eu
 if [ $# -eq 0 ]
     then
         echo "No arguments supplied: run ./script.sh <
            params>"
         exit 1
 fi
 param_line=$1
 IFS=','
 read -ra PARAMS <<< "$param_line"
 cd ../example/Zombie/Simulation_Netlogo
 /usr/local/java/bin/java -Xmx1536m -XX:-UseLargePages
    -jar target/netlogo-1.0-wrapper.jar \
     -m resources/models/JZombiesLogo.nlogo \
     -outfile counts.csv \
     -runid 1 \
     -s 250 \
     -trial 1 \
     -i human_count, ${PARAMS[0]}, zombie_count, ${
        PARAMS[1]}, human_step_size, ${PARAMS[2]},
        zombie_step_size, ${PARAMS[3]} \
     -o human_count\
```

When HSO starts, the master node waits to receive initialization messages from the workers and the optimizer process. The initialization messages are:

- `INIT OUT::out address:port::`, this message is sent from the optimizer to HSO, the message informs HSO about the IP address and port to be used for the communication using the ZeroMQ library (this message enables to use the optimizer and the HSO on different networks);

- `INIT CPU::cpu model::`, this message informs HSO about the script to be used for executing a CPU simulation;

- `INIT GPU::gpu model::`, this message informs HSO about the script to be used for executing a GPU simulation;

- `INIT NUM::num cpu::num gpu::`, this message is sent by the optimizer to HSO and informs HSO about how many CPUs and GPUs (according to the available CPUs and GPUs in the system) are available for the OvS process.

As described above, the HSO uses two queues (`EVAL` and `OUT`) for the communication between the optimizer and the HSO. Two kinds of messages are sent on the queues:

- `INPUT::`$param_1$`;`$param_2$`;...;`$param_N$`::N::`, this message contains a set of simulation configuration, and is sent by the optimizer on the `EVAL` queue. Each value $param_i$ is a configuration as a list of values separated by a comma, corresponding to the values of each simulation parameter. HSO generates the mapping Π using a Round-robin strategy (other strategies can be easily implemented). All inputs are divided among the workers in equal portions and in circular order. Each configuration is executed by a CPU or a GPU. We notice that when a GPU is used, also the corresponding CPU is busy. For this reason, if a GPU simulation is assigned, it is not possible to assign on the same machine also a CPU simulation;

- `OUTPUT::`res_1`;`res_2`;...;`res_N`::`, this message contains the output values of the simulation process, each res_i is the output value of a requested configuration.

4.2.3 Usage of HSO

As described before, the OvS developer must define the scripts for executing CPU and GPU simulations and distribute them on all the available machines. The optimizer can be written in any language supported by the ZeroMQ library. Listing 4.2 shows the code of a dummy optimizer, written in Java language. The optimizer uses the HSO to analyze two random configurations in two optimization rounds.

Lines $14-21$ show the configuration code of the ZeroMQ, while line 26 and 27 show the code for sending HSO optimizer initialization messages. Lines $30-38$ show the code to initialize CPU and GPU

HSO simulation software. Finally, lines $40 - 69$ show the code of the random optimization process. In each optimization round, a random configuration **x** is computed and sent to the HSO to be evaluated. At the end of each round, the results are printed on the console.

Listing 4.2: Dummy Java Optimizer

```
//other imports are omitted
 import org.zeromq.*;
 import org.zeromq.ZMQ.Socket;
 public class Optimizer {
    private static ZMQ.Context context;
    // HSO Queues
    private static Socket eval, out;
    static String IP="127.0.1.1";
    static String PORT_EVAL="2222";
    static String PORT_OUT="2223";

    public static void main(String args[]) throws
        Exception {
        // ZeroMQ initialization
        context = ZMQ.context(1);
        String path_eval = "tcp://"+IP+":"+PORT_EVAL;
        System.out.println(path_eval);
        String path_out = "tcp://"+ Inet4Address.
            getLocalHost(). getHostAddress() +":"+
            PORT_OUT;
        eval = context.socket(ZMQ.PAIR);
        eval.connect(path_eval);
        out = context.socket(ZMQ.PAIR);
        out.bind(path_out);

        //Init HSO with Optimizer IP
        System.out.println("INIT_OUT::" +path_out +"::"
            );
        System.out.println(("INIT_OUT::" +path_out +"::
            ").length());
        eval.send(String.valueOf(("INIT_OUT::" +
            path_out +"::").length()),0);
        eval.send("INIT_OUT::"+path_out+"::",0);

        //Init HSO with cpu and gpu simulation model
        String cpu_model = "INIT_CPU::cpu_script.sh::";
```

```
String gpu_model = "INIT_gpu::gpu_script.sh::";
String n_acc = "INIT_NUM::1::1::";
eval.send(String.valueOf(n_acc.length()),0);
eval.send(n_acc,0);
eval.send(String.valueOf(cpu_model.length()),0)
    ;
eval.send(cpu_model,0);
eval.send(String.valueOf(gpu_model.length()),0)
    ;
eval.send(gpu_model,0);

byte[] res;
String result;
String[] tmp,array_res;
StringBuilder sb = new StringBuilder("");
int round = 2;
Random r = new Random();
int N = 10;
while(round > 0)
{
    String x = r.nextInt()+"," +r.nextInt() +";
        " +r.nextInt() +"," +r.nextInt();
      //x denotes a configurations
    sb.append("::INPUT::"+x+"::2::");
    String params = sb.toString();
    eval.send(String.valueOf(params .length())
        ,0);
    eval.send(params,0);
    //Evaluate the configuration using HSO
    out.recv(0);
    res = out.recv(0);
    result = new String(res);
    tmp = result.split("::");
    array_res = tmp[1].split(";");
    int num = 0;
    for(int i=0;i<N;i++){
        for(int j=0;j<2;j++){
            System.out.println (Integer.
                parseInt(array_res[num]));
            num++;
        }
    }
    round--;
```

```
        }
        eval.close();
        out.close();
        context.term();
        System.exit(0);
    }
}
```

4.2.4 Use cases

We have studied two social science diffusion problems using different optimization strategies. The purpose of these use cases is, on one side, to show how to use different optimization strategies in HSO, and on the other side, to provide examples of the HSO programming languages heterogeneity.

In the first use case, we have simulated a *Spread of Information* process using a SIR ABM compartmental epidemiology model and genetic algorithm to optimize the simulation parameters. This use case is presented in [35]. The authors use the EMEWS framework to evaluate the OvS process on an HPC system. The OvS process is developed using a Python genetic algorithm for the optimization, and a Repast simulation, for the simulation. We developed the same OvS process, using a NetLogo simulation and a FlameGPU simulation, and we have tested it on a heterogeneous cluster machine.

The second use case is a *Spread of Influence* process, which models the social influence diffusion process in a social network. Starting from the theoretical diffusion model defined in [98] we developed a CPU based simulation, written in the C language, and a GPU based simulation, written using the Nvidia CUDA [99]. For the optimization strategy, we have used a Java implementation of the Optimal computing budget allocation (OCBA) approach, which has been presented in [100].

Spread of information

This use case is named "Zombie"[1] and aims to show the HSO ability to reuse code already written in other contexts. The ABM simulation

[1] HSO Zombie use case, https://github.com/isislab-unisa/hso/tree/master/example/Zombie

model is an instance of the SIR compartmental epidemiology model in which each of the agents is moving in a 2-dimensional environment and have three different behaviors: *S* for susceptible, *I* for infectious, and *R* for recovered (or immune).

The agents interact with each other and change their behavior according to simple rules. If the agent is in the state *S* change to *I* when interacts with an agent in the state *I*, after some time, an agent in the state *I* changes its behavior to *R*. We used a variant of the SIR model that uses only the behaviors *S* and *I*. The agents in the state *S* represent the humans while the agents in the state *I* represent the zombies.

The CPU simulation has been written using NetLogo[1], while we have developed the same simulation using FlameGPU, to exploit the computational power of GPUs.

Simulations parameters are:

- *human count*, an integer value that is the initial number of agents in the state *S* (number of humans);
- *zombie count*, an integer value that is the initial number of agents in the stat *I* (number of zombies);
- *human step size*, a real value that is the velocity of the humans;
- *zombie step size*, a real value that is the velocity of the zombies;

The output of the simulation is the number of humans that are survived during a fixed amount of simulation time. In this OvS use case, we assume that we are interested in maximizing the number of humans that can survive at the end of the simulation.

The optimization process uses a Python implementation of a genetic algorithm, written using the Distributed Evolutionary Algorithms Python library[2]. We have modified the original implementation, presented in [35], to be suitable for HSO. Hence, the OvS process is written in Python and starts from an initial set of configurations $\mathbf{X}$. During the first step, the HSO simulates the initial configurations. After this

[1]Zombie: simulation Netlogo, https://github.com/isislab-unisa/hso/tree/master /example/Zombie/Simulation_Netlogo

[2]Distributed Evolutionary Algorithms in Python, https://github.com/deap

initial step, it executes the genetic algorithm and generates a new set of configurations $\mathbf{X}$. The OvS process continues until a certain number of epochs (optimization rounds) is reached.

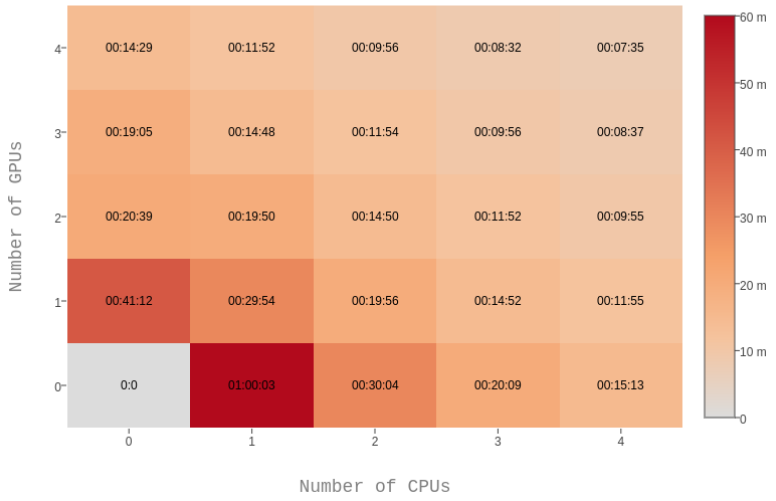

Figure 4.3: OvS Spread of Information use case performance over 4 CPUs and 4 GPUs for 10 optimization round.

Spread of influence

The second use case aims to study the spread of influence in social networks (see [98] and references quoted therein). For instance, in the area of viral marketing [101] companies wanting to promote products or behaviors might try initially to target and convince a few individuals which, by word-of-mouth effects, can trigger a cascade of influence in the network, leading to an adoption of the products by a much larger number of individuals. The intent of maximizing the spread of viral information across a network naturally suggests many exciting optimization problems. In our case, we are looking for a fixed-sized initial set of nodes (initial adopters) that maximizes the number of

influenced nodes at the end of the diffusion process. We have used a diffusion model presented in [98]. This model starts from an initial set of individuals lying in a social network and simulating a threshold-based influence diffusion process.

Formally, given a social network defined by a graph $G = (V, E)$, and a set of initial adopters $S \subset V$, at each round i the set of influenced nodes is augmented by the set of nodes u that have several already influenced neighbors greater or equal to u's threshold $t(u)$. The process terminates when two rounds terminate with the same set of influenced nodes. The threshold $t(u)$ quantifies the human tendency to conform, that is, how hard it is to influence node u, in the sense that easy-to-influence individuals of the network have "low" value. In contrast, hard-to-influence individuals have "high" $t(\cdot)$ values. Unfortunately, the threshold values are usually not known a priori. For our purposes, we have studied the problem when the threshold values are unknown, but to perform fair comparisons, the total sum T of the threshold values is known. Consequently, given an initial set S, the diffusion process must be executed many times, randomly selecting a distribution of the threshold values over the network, to correctly estimate the number of activated nodes at the end of the process.

We have developed the diffusion model for CPU, using the C language, and for GPU, using the Nvidia CUDA library. Both simulations take as inputs a graph G, a subset S, the sum of the threshold T, and the number of trials t to be executed. The diffusion process returns the expected number of influenced nodes for a given subset S. For each trial, the process randomly selects the distribution of the threshold values over the network, such that the sum of the values is equal to T.

The OvS process is written in Java and uses the Optimal computing budget allocation (OCBA) approach [100] to find out the best subset S with respect to the threshold sum T. OCBA approach starts from an initial set of configurations, in our case, a certain number of subsets $\mathbb{S}$, that are randomly selected by the optimizer. After that, the OvS process evaluates the subsets using HSO and the diffusion model described above for a fixed number of trials i. At the end of the first evaluation, OCBA is invoked to compute the new number of trials i for each subset $S \in \mathbb{S}$. In other words, the OCBA selects, from the initial set of

subsets $\mathbb{S}$, the best candidates accordingly to the results of the previous evaluations. OvS process continues until the number of subsets $\mathbb{S}$, which has a value of trials greater than zero, is equal to a chosen number of solutions or a fixed number of optimization rounds.

Table 4.1: Social Networks Spread of Influence use case

Network	#Vertices	#Edges	Avg. Degree
egonets-Facebook	4039	88234	22
ca-AstroPh	18772	396160	21
ca-HepPh	12008	237010	20
email-Enron	26692	367662	10
ca-CondMat	23133	166936	8
ca-GrQc	5242	28980	9
ca-HepTh	9877	51971	5

OvS process evaluations

We tested the OvS use cases, described above, on a heterogeneous cluster machine with:

- 14 CPUs: 7 machines with 2 x Intel(R) Xeon(R) CPU E5-2680 2.70GHz and 256 GB of RAM.
- 8 GPUs: 3 machines with 2 x Tesla M2090 and 2 machines with 1 x Tesla M2090.

We performed several experiments, evaluating the first use case, setting the initial number of humans to 1000 and the number of zombies to 400. The human's velocity is set between 1.0 and 10.0, while the zombie velocity varies between 0.1 and 4.0. The optimization process starts from an initial population of 200 individuals and performs 10 epochs (optimization rounds). Overall the system performed $1,284$ simulations.

Figure 4.3 depicts the time (using a color time scale) required to complete the OvS process, varying the number of CPUs (x-axis) and

the number of GPUs (y-axis) involved in the computation. As shown in the figure, the best performance is achieved by the configuration that uses 4 CPUs and 4 GPUs (the most heterogeneous execution setting). The obtained speedup (i.e., the sequential execution time ratio to the HSO execution time) is 5.

We tested the second OvS use case process, on 7 real networks of various sizes from the Stanford Large Network Data set Collection (SNAP) [101] ((see Table 4.1), setting the size of $|S| = 5\%$ of $|V|$ and the $T = |E|$. The number of evaluated subsets is equal to 200, and the number of initial i trials for each subset is equal to 10. We run the experiment for at most 100 optimization round. Overall a total number of $200,000$ simulations have been performed.

As for the previous use case, we have tested the HSO system using at most 4 CPUs and 4 GPUs. Figure 4.4 depicts the obtained speedup for three CPUs and GPUs configurations. The best performance, as shown, is achieved by the heterogeneous configuration for each considered network.

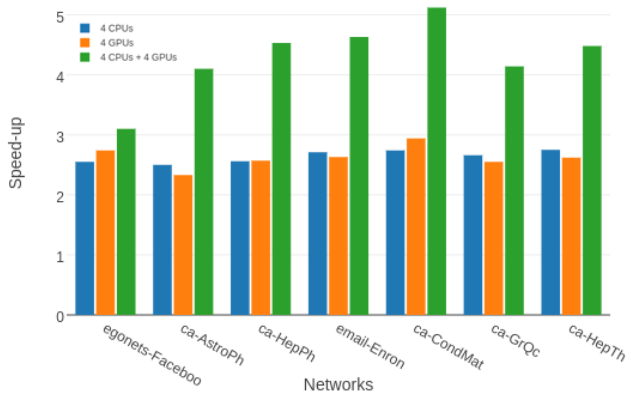

Figure 4.4: OvS Spread of Influence speed-up over 4 CPUs and 4 GPUs.

We also notice that the OvS process ensures good quality results. The hardship to influence a network is proportional to the average degree of the nodes. We experimented on the first three networks (those with the highest average degree) the OvS process, varying the size of

the initial set S. Clearly, the smaller is the initial set S, the harder it is to influence the network. As shown in the Table 4.2, we have tested different sizes of S from 5% to 20% of V.

Table 4.2: Percentage of influenced vertices after 100 OCBA optimization rounds.

Network	**%5**	**%10**	**%15**	**%20**
egonets-Facebook	7.6	16.3	54.2	97.9
ca-AstroPh	7.2	63.4	95.8	98.9
ca-HepPh	11.4	75.1	94.5	97.9

4.3 Assisted Parameter and Behavior Calibration in Agent-Based Models with Distributed Optimization

In an agent-based model, many *agents* (computational entities) interact to give rise to emergent macrophenomena. Agent-based models (ABMs) are widely used in computational biology, social sciences, and multiagent systems. An important step in developing an agent-based model is *calibration*, whereby the model's parameters are tuned to produce expected results. Agent-based models can be challenging to calibrate for several reasons. First, agents often have numerous and intricate interactions, producing complex and difficult to predict dynamics. Second, the agents themselves may be imbued with *behaviors* that need to be tuned: and thus the parameters in ABMs may not just be simple numbers but computational structures. Finally, ABMs are often large and slow, which reduces the number of trials one can perform in a given amount of time.

Despite its importance, ABM calibration is often done by hand using guesswork and manual tweaking, or the model is left uncalibrated because the model's complexity makes it too difficult for the modeler

to perform the calibration! For example, in [102] approximately half of the surveyed models performed no calibration at all.

In this work we consider the task of *automated agent-based model calibration*. We marry two tools popular in their respective fields: the MASON agent-based simulation toolkit [23], and the ECJ evolutionary optimization library [34]. MASON is an efficient ABM simulation tool which can be serialized and encapsulated in a single thread, making it a good choice for massively distributed model optimization, and ECJ has facilities critical to ABM optimization: it can perform distributed evaluation on potentially millions of machines, and it has a wide range of stochastic optimization facilities useful for agent-based modeling.

We will begin with an introduction to the ABM model calibration problem and discuss previous work in model calibration and optimization. We will next provide some background on ECJ and MASON, then present our approach to massively distributed ABM calibration, including examples that provide insight into the breadth of the approach.

4.3.1 Approach

As agent-based models become more common and more detailed, an automated approach to calibration will be increasingly needed. We envision the automation process to work as follows. The modeler first builds the simulation, then assigns values to those parameters he knows or wishes to be fixed. A distributed system then optimizes the remaining parameters as best it can against criteria specified by the modeler. The modeler then examines the results: if they are poor, this could be due to bugs in the model, or insufficient model complexity to demonstrate the modeler's hypothesis, or a hypothesis that is wrong. Accordingly, the modeler revises the simulation and resubmits it to the system to be recalibrated.

To do this procedure, we merged ECJ and MASON to take full advantage of their technical characteristics. To merge them, it was necessary to make changes to both. Without going into implementation details: first, ECJ was modified so that MASON simulations could be used in the evaluation procedure of a candidate solution. Second, MASON was modified to be able to receive the values of model parameters

from outside (that is, from an ECJ process) and to provide the modeler with a way to develop a score function for the simulation (which would then be used by the optimization algorithm).

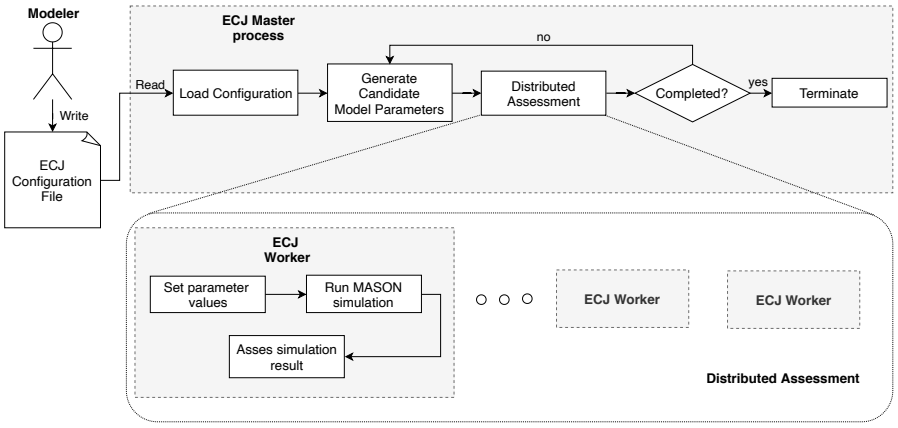

Figure 4.5: Automated model calibration workflow using distributed ECJ and MASON.

Figure 4.5 shows the general workflow of the system. We first define one ECJ process to be the *master*. This process performs the top-level optimization algorithm. When this process has one or more candidate solutions (individuals) ready to be assessed, they are handed to a remote *worker* process. Each candidate solution is simply a set of those agent parameters and behaviors that we wish to test: the worker does this by creating a MASON simulation using those parameters and behaviors, running it some number of times, and assessing its performance. The worker then returns the assessments as the *fitness* (quality) of its tested solutions, and the master uses these results in its optimization procedure.

The modeler can completely customize this procedure if need be, and we demonstrate one such scenario in Section 4.3.5. But if the modeler's optimization needs only involve global model parameters — as is typical for many ABM calibration scenarios — then we provide a simple alternative. The modeler specifies which parameters of interest to optimize, then selects from a few optimization options, and MASON does the rest: it defines the candidate solution representation as a fixed-

length list of the parameters in question, builds the fitness mechanism, creates the evolutionary process, prepares the workers to run the proper simulation and default settings, then sends the candidates to remote workers.

4.3.2 Speedup demonstration

We first show the efficiency of our distributed model calibration facility on a nontrivial ABM scaled horizontally across a cluster of machines. For this demonstration, we use the *Refugee* model drawn from the contributed models in the GeoMASON distribution. This model can take several minutes to run. *Refugee* explores the pattern of migration of refugees in the Syrian refugee crisis. The model demonstrates how population behavior emerges as the result of individual refugee decisions. The agents (the refugees) select goal destinations in accordance with the Law of Intervening Opportunities and these goals are prone to change with fluctuating personal needs.

We calibrated the model using a simple genetic algorithm from ECJ and assessed candidate solutions by comparing the number of arrivals in each city against real-world data gathered from UNHCR and EU agency databases.

Setup. We calibrated over four real-valued parameters in the model. The model ran for 10,000 steps. We used a genetic algorithm with a tournament selection of size 2, one-point crossover, and Gaussian mutation with 100% probability and a standard deviation of 0.01. We ran the models on a cluster of 24 machines, each with Dual Intel Xeon E5-2670@2.60GHz, 24 GB, Intel 82575EB Gigabit Ethernet, Red Hat Enterprise Linux Server 7.7 (Maipo), OpenJDK 1.8.0. Running on these machines were some $N \leq 276$ MASON worker processes.

Results. We performed *strong* and *weak scalability* analysis. Strong scalability asks how much time is needed for a *fixed size* problem given a *variable number of workers*. Weak scalability asks if an *increasingly difficult problem* can be handled in the same amount of time with a corresponding *increasing number of workers*. All the scalability results

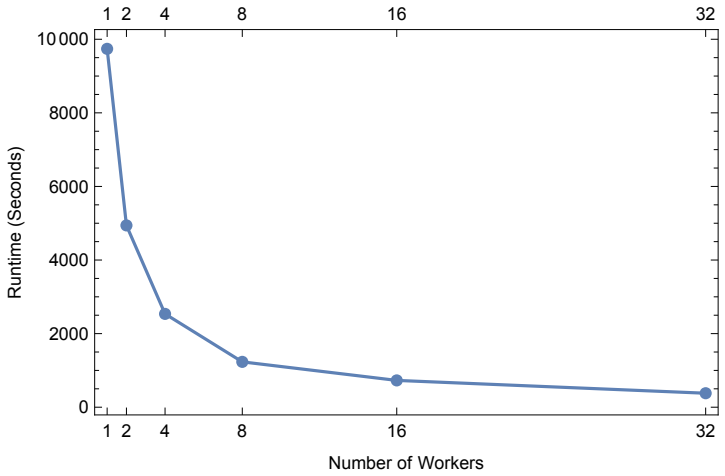

Figure 4.6: Strong Scalability Analysis, Refugee model, for 1, 2, 4, 8, 16, or 32 Workers

are statistically significantly different from one another ($p < 0.01$) as verified by a one-way ANOVA with a Bonferroni post-hoc test.

To do strong scalability analysis we fixed the problem to ten generations, each with 32 individuals, for a total of 320 evaluations. The number of workers was varied from 1 to 32. For 1 to 8 workers we gathered the mean result of ten experiments; for 16 and 32 workers (which ran faster), we gathered the mean result of 20 experiments. Figure 4.6 displays the speedup results. The *strong scalability efficiency* (as a percentage of the optimum) came to 71.88% using 32 workers to solve the problem.

To do weak scalability analysis, we varied the problem difficulty by adjusting the population size such that, regardless of the number of workers, each worker was responsible for four individuals (and thus four simulation runs) per generation. In all cases, the results reflect a mean of ten experiments. For each optimization process the number of generations was fixed to 10 and the population size varied in $\{4, 8, 16, 32, 64, 128, 256, 512\}$, and thus the number of workers varied as $p \in \{1, 2, 4, 8, 16, 32, 64, 128\}$. Figure 4.7 shows the weak scalability results. The *weak scalability efficiency* (as a percentage of the optimum) was 83.18.

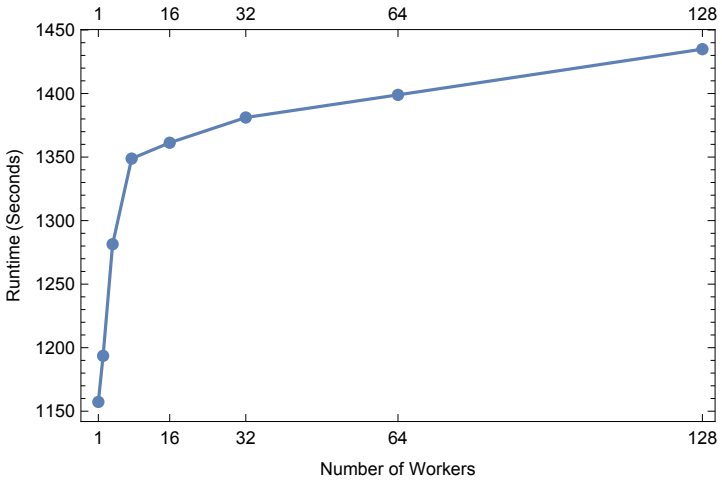

Figure 4.7: Weak Scalability Analysis, Refugee model, for 1, 2, 4, 8, 16, 32, 64, or 128 Workers

4.3.3 Asynchonous evolution experiment

The previous experiment involved a *generational* evolutionary optimization algorithm: the entire population of individuals had to be evaluated on the remote workers before the next generation of individuals was constructed. We next considered an *asynchronous* evolutionary algorithm to improve efficiency when the model runtime varied greatly. An asynchronous evolutionary algorithm only updates the population a little bit at a time, rather than wholesale, and doesn't need to wait for slowly-running models.

The approach works as follows: there are some N workers and a master with a population of size P. The master first creates random individuals, then assigns each to an available worker. When a worker has completed its assessment, the individual is returned and added to the population, and the worker becomes available for another task. When the population has been fully populated, the master switches to a *steady-state* mode: when a worker is available, the master applies the evolutionary algorithm to produce an individual which is then given to the worker to asses. When an individual is returned by a worker, the master selects an existing individual in the population to be replaced by

the new individual.

Setup. We compared the generational genetic algorithm from Section 4.3.2 against an asynchronous evolutionary algorithm using a steady-state genetic algorithm. When replacing an individual in the population, the steady-state algorithm selected the least fit individual. In our experiment, there were 128 workers, and the population size was 128: the generational approach was again run for ten generations, while the asynchronous approach was run until it had evaluated 1280 individuals. We performed twenty experiments per treatment. To simulate varying runtimes in the Refugee model, when a model was to be tested we changed the number of simulation steps at random. 1/4 of the time we halved them, 1/4 of the time we left them as normal, and 1/2 of the time we doubled them.

Results. Asynchronous Evolution had a mean runtime of 293.77 seconds; while Generational Evolution had a mean runtime of 437.77 seconds. These results were statistically significantly different ($p < 0.01$) as verified by a one-way ANOVA with a Bonferroni post-hoc test.

4.3.4 Evolutionary optimization examples

So far we have shown speedup results to demonstrate performance: next, we turn to simple examples of some of many evolutionary algorithms approaches afforded by our facility, to illustrate capabilities of the system and justify their value in an ABM. Accordingly, the remaining demonstrations will be mere proofs of concept, and so will not be accompanied by statistical analysis.

Different evolutionary algorithms. We begin with a demonstration of some different evolutionary algorithms to show breadth. We turn to the *Flockers* model, a standard demo model in the MASON library. This model is a simulation of the well-known *Boids* algorithm [89], where agents develop collective realistic flocking or swarming behaviors.

Flockers has five classic parameters (avoidance, cohesion, consistency, momentum, and randomness) that together define the behaviors

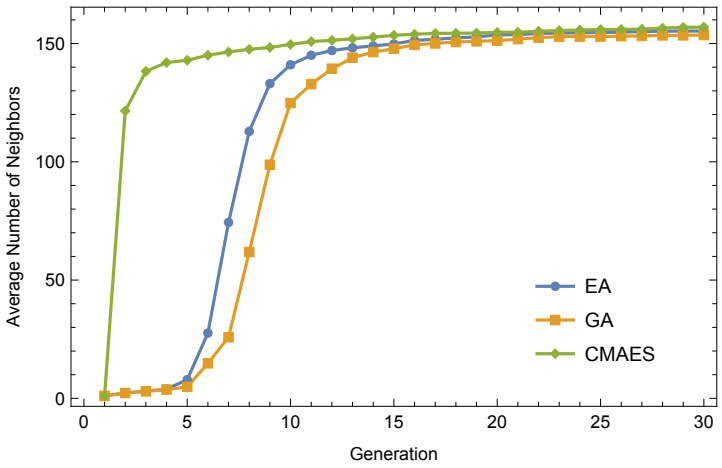

Figure 4.8: Mean best-so-far performance of a Genetic Algorithm, Evolution Strategy, and CMA-ES on the Flockers domain, averaged over 30 runs.

of its agents. We optimized over these parameters and assessed the model performance as the mean number of flockers within an agent's neighborhood, averaged over three trials. This is not a hard problem to optimize: the calibration facility need only maximize cohesion. To show the optimizers at work, we fixed every individual in the initial population to represent the opposite situation (minimal cohesion, maximal values for other behaviors).

We ran for 30 generations using a population size of 276, spread over 276 separate workers. We compared three different evolutionary algorithms: the genetic algorithm as described before; a so-called "$(46, 276)$" *evolution strategy*; and a *CMA-ES* estimation of distribution algorithm with standard parameters. Figure 4.8 shows the performance of these three algorithms on this simple agent-based model: as expected, CMA-ES performs extraordinarily well.

Multi-Objective optimization. Next, we demonstrate our system's ability to optimize problems with multiple conflicting objectives. The classic approach finds a set of solutions that have advantages or disadvantages relative to one another with respect to these objectives. A

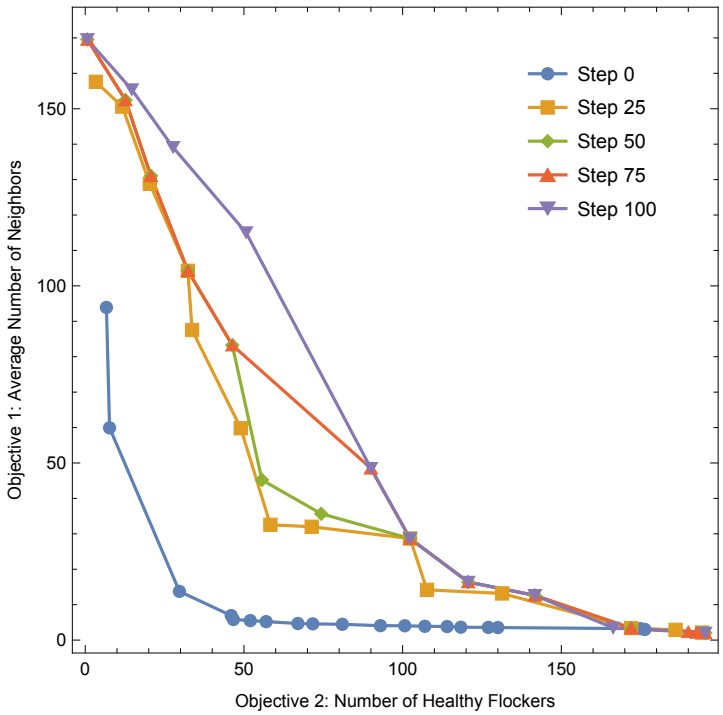

Figure 4.9: Pareto Nondominated Fronts (higher values preferred) for a typical run of the two-objective Flockers domain at generations 0, 25, 50, 75, and 100.

solution A is said to *Pareto-dominate* another solution B if A is at least as good as B in all objectives and better than B in at least one objective. The optimal *Pareto Nondominated Front* is the set of solutions not Pareto-dominated by any other solution.

We extended the Flockers model by introducing an "infection" into the population. Healthy flockers have the same behavior as shown in the previous example, but infected flockers will, with some probability, infect their neighbors or be cured. Our new second objective was to maximize the number of healthy flockers. To do this, flockers must stay as far away from each other as possible, putting our new objective in direct conflict with the first one.

We used the NSGA-II [103] multi-objective evolutionary algorithm with four workers and 100 generations, having 24 individuals per gener-

ation. Figure 4.9 shows the improvement in the Pareto front over time for a typical run.

An aside: coevolution. Though we do not provide demonstrations of them, it is worth mentioning two other capabilities of our system, which may be of value to an agent-based modeler.

In some cases, is ti possible that one might wish to calibrate a model to be *insensitive to* one or more global parameters. For example, we might wish agents to perform migration the same way regardless of rain or shine. One attractive evolutionary optimization approach is *competitive coevolution*. Here we optimize the population A against a second *foil* population B of parameter settings simultaneously being optimized to trip up the first population. Thus while A is trying to be insensitive to B, B is searching for corner cases to challenge A.

A related technique, called *cooperative coevolution*, is a popular way to tackle high-dimensional problems. When the number of parameters to optimize is high, the joint parameter space is exponentially too large to efficiently search. Cooperative coevolution breaks the space into N subspaces by dividing the parameters into N groups, each with its own independently optimized population. Individuals are tested by combining them with ones from the other populations to form a complete solution. The fitness of an individual is based on the performance of various assessed combinations in which it has participated. This reduces the search space from $O(a^N)$ to $O(aN)$, but assumes that the parameters in each group are largely statistically unlinked with other groups.

4.3.5 Optimizing agent behaviors

Agent-based models are unusual in that not only do they have (typically global) *parameters* which must be calibrated but agents with *behaviors* that may benefit from calibration as well. Agent behaviors are essentially programs which dictate how the agents operate in the environment and interact with one another. Unfortunately, it is often the case that the modeler does not know what the proper behavior should be for a

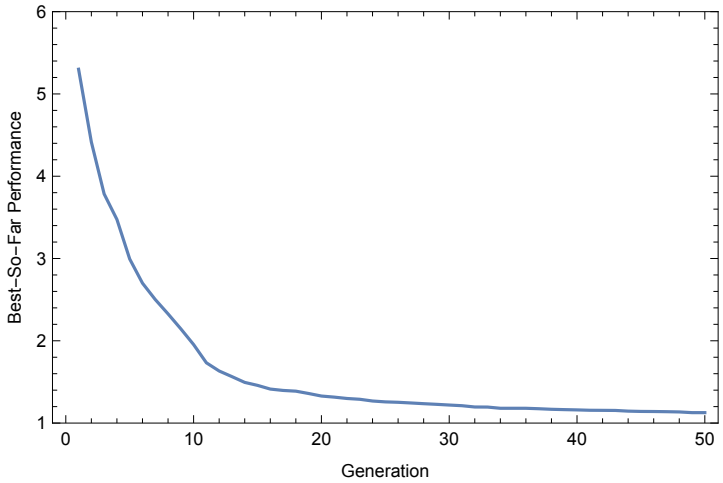

Figure 4.10: Mean best-so-far performance, over 30 runs, of genetic programming on the Serengeti model (lower is preferred).

given agent, or only understands part of the behavior and needs to fill in the blanks with the remainder.

Because we are calibrating agent behaviors and not global model parameters, the modeler must do more than just specify a set of model parameters to calibrate and an optimization algorithm to use. He must also specify the *nature* of the representation of these agent behaviors (in our case below, an array of four parse-trees), and must also write glue code which, when given an individual, *evaluates* its parse trees in the simulation proper.

The evolutionary algorithm community has developed optimization techniques for a variety of agent behavior representations. Out of the box, we can support *policies* (stateless sets of *if→then* rules that determine actions to take in response to the current world situation), *finite-state automata* (as graph structures), *neural networks* (via NEAT), and untyped or strongly-typed "Koza-style" *genetic programming* (or GP) [104]; and provide hooks for a variety of other options.

In this example, we will focus on GP. Here, individuals take the forms of forests of parse trees populated by functions drawn from a modeler-specified *function set*. Functions may have arguments, types,

and arbitrary execution order (like Lisp macros). Parse trees typically impact on behavior through side effects among their functions, or by returning some final result via the root of the tree. Our example is drawn from the *Serengeti* model [105], in which four "lion" agents must capture a "gazelle" in a real-valued toroidal environment. The gazelle uses a simple hard-coded obstacle-avoidance behavior to elude the lions, and can move three times as fast as any single lion. The lions can sense the gazelle and each other. Each lion uses a GP parse tree that, when evaluated, returns a vector indicating the direction and speed the lion should travel at that timestep. Thus the behaviors to be calibrated consist of four different parse trees, one per lion.

We used a GP facility closely following the approach in [105], including its function set (we restricted ourselves to the "name-based sensing" and "restricted breeding" variants as described in the paper). We ran the GP algorithm as described, but with a population size of 5760 spread over 276 workers: each worker thus had 20 individuals per generation. Assessment of an individual's parse trees was performed over 10 random trials. Figure 4.10 shows the mean best-so-far performance of calibrated agent behaviors over 30 runs.

4.4 Large-scale Optimized Searching for Cruise Itinerary Scheduling on the Cloud

The Cruise Itinerary Schedule Design (CISD) problem involves companies for cruising and transportation. The problem consists of defining an optimal permutation for a subset of a given set of ports, representing the cruise itinerary and arrival and departure times for each destination to minimize the total cost, maintaining some constraints defined by the cruising company. The research community has developed several solutions for the CISD problem, exploiting either optimization or simulation.

In detail, a cruise itinerary is a sequence of destinations (the naval ports) visited by a ship during a cruise. Cruises are planned according to weather conditions forecast and historical data considering different geographical zones for each season. The globe is divided into zones

where a ship stays for a certain period (the cruising season). Furthermore, the route of the ships during a season may visit different zones according to a predefined sequence. The outcome of a planning process is the positions of each ship for each day to reach the planned cruising destinations, minimizing the total cost.

Despite the steady growth rate in the cruise industry, the CISD problem is barely present in the academic literature. In [106] is presented a review of cruise industry-academic literature mainly from a marketing and revenue management perspective. In general, it is agreed that very little research has addressed cruise ship problems and [107] explains this lack of research as a consequence of the following data: cruise tourism accounts only for 2% of the whole tourism sector.

More recently [108] have reviewed the cruise industry and highlighted how state-of-the-art research provides empirical and descriptive studies without any optimization-based analysis on cruise shipping-related problems. Furthermore, research by [108] explains why cruise ships have to be considered differently from cargo ships and land transportation and therefore require ad hoc problem resolutions. For interested readers, detailed literature examination can be found in [109].

In this work, we present a solution to the CISD problem that exploits the Optimization via Simulation (OvS) process to find a sub-optimal solution to the problem. We exploit a genetic algorithm in the optimization phase to generate a new candidate set of parameters, while in the second process phase, we developed a heuristic, based tabu search algorithm, which generates candidates itineraries (see Section 4.4.3). We used the Simulation exploration and Optimization Framework for the cloud (SOF) presented in [46] and described in Section 4.4.2, because of the framework ability to exploit the Cloud computing environment easily.

4.4.1 Cruise Itinerary Schedule Design (CISD)

The Cruise Itinerary Schedule Design (CISD) is a planning process where we are interested in maximizing/minimizing some cruise itinerary constraints. In a cruise itinerary, we have to define the route of the ships to reach interested ports with the minimum fuel consumption,

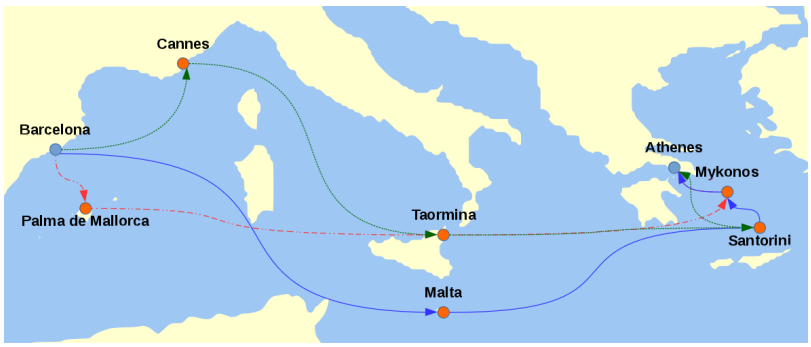

Figure 4.11: Example of CISD problem.

considering the availability of the ports. During the cruising time, a ship can be in a port docked to a pier (or anchorage) or at sea (traveling). This leads in costs or benefits considering the total cruising cost because a ship has a different cost if it is traveling (the amount of fuel needed to operate a route), stay anchored, or stay stopped in a port (the port is characterized by an attractiveness value defined by tourism perspective).

This planning process may be summarized in three phases:

- *Ship Allocation* phase, in which a ship is allocated to a given maritime area in a given seasonal period;
- *Cruise Scheduling* phase, in which, given a planning horizon, the process defines the cruises scheduling
- *Day-By-Day Optimization* phase, that leads to the selection of all the ports of an itinerary, defining the visit sequence, the timing, and the cruise speed for each cruise schedule, considering that every port can not be visited twice except for the embark and debark ports.

These phases could be embedded in a classical Operations Research problem. The first phase is the strategic level, the second phase is the tactical level, and the latter phase is the operational level.

The CISD problem is described formally in [110] as follows. The CISD problem inputs are:

- P, the set of ports with the associated logistic characteristics (dock or anchorage, refueling, embark/debark, turnaround);
- N, the number of cruise liners which operate in the area;
- V, the set of cruising speeds for the cruise liners;
- $c(i, j, v)$, is the fuel cost to travel from the port i to the port j at speed v;
- α, an attractiveness index of each port;
- β, the total cost for fuel bunker, ship maneuvering, and the ship stop in the port.

The CISD problem constraints are:

- restriction fuel type for ports and maritime zones;
- the minimum and maximum number of hours that a ship can stop in a port;
- the time windows for arrival and departure;
- the minimum and maximum: length of an itinerary in nautical miles, number of visited ports for an itinerary, percentage time that a ship can stay in the sea, number of consecutive traveling days, the distance among ports in hours of navigation (according to the ship);
- the cabotage laws is an international law that defines cost constraints for the ships traveling among different countries;
- mandatory/forbidden ports and countries to visit.

The planning objective is defining a sequence of ports to be visited by each ship, defining the scheduling time of arrivals and departures, to minimize the total cost C of the cruise (fuel cost and ports cost) and maximize the commercial attractiveness A of the cruise (the sum of α values of each visited ports). The CISD problem is a two-objective optimization problem [111] and a solution can be found using the *Pareto Efficient Frontier*.

Cruise attractiveness index. From the cruising companies' perspectives, the solution's quality is a trade-off between the cost and the quality of a cruise. From the commercial point of view, the most important successful index of cruising is tourist attractiveness. The evaluation of the attractiveness is a relative measure (a rating score) of the impact on the customers of a cruise according to the characteristics of the visited cities and tourism values as season, historical frequency of visit of a particular port, etc. The α attractiveness index is computed according to different cruising attributes to increase the commercial interest for customers. The considered attributives are: reputation (political stability, safety, etc.); port facilities (infrastructures, distance to the city center, etc.); activities (cultural/natural, food and beverage, shopping, etc.); exclusivity (crowding level, etc.); cruise design (number traveling days, overnights in ports, etc.).

4.4.2 Simulation exploration and optimization framework for the cloud

In [46] the authors introduced *Simulation exploration and Optimization Framework for the cloud* (SOF). This framework enables domain experts and/or policymakers to run and collect results for two kinds of optimization scenarios: parameter space exploration (PSE) and simulation optimization (SO).

Figure 4.12 depicts the SOF work cycle, which comprises three phases: selection, parallel simulation, and evaluation. SOF provides a set of functionality that enables developers to construct their simulation optimization strategy. We designed the framework based on the following objectives:

- *zero configuration:* the framework neither requires the installation nor the configuration of any additional software, only Hadoop and SSH access to the hosting platform are required;
- *ease of use:* the tool is transparent to the user since the user is unaware that the system operates in a distributed environment;
- *programmability:* both the simulation implementation and the

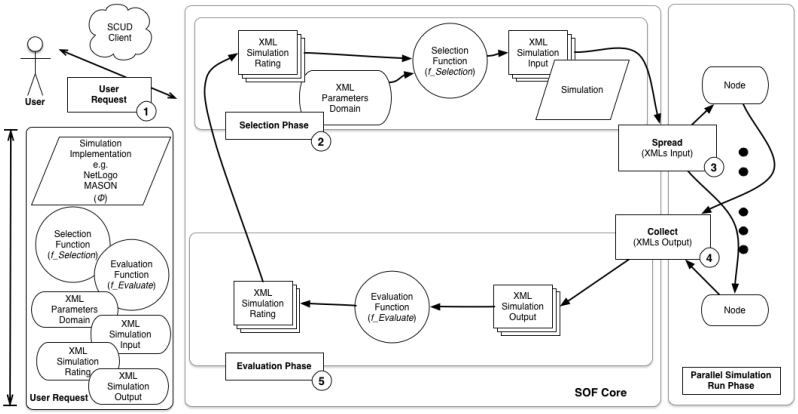

Figure 4.12: SOF Work Cycle.

simulation optimization functionalities can be implemented using different simulation toolkits (MASON, NetLogo, etc.) and/or by exploiting different programming languages supported by the hosting platform;

- *efficiency:* by executing several independent tasks (simulations) concurrently, the framework adequately exploits the resources available on the hosting platforms.

Work cycle. The framework is divided into three functional blocks: *the User Front-end* (Figure 4.12, left); *the SOF core*, which acts as a controller (Figure 4.12, middle); *the computational resources* (Figure 4.12, right).
The User front-end is implemented as a web or a standalone application, through which the user provides the inputs to the system which are: *Simulation Implementation*, *Selection Function*, *Evaluation Function* and are written using any language supported by the cloud environment. According to SOF XML schema, a set of files in XML format defines the Parameters Domain, the Simulation Input, Output, and Rating. The application level of SOF provides a tool to generate the needed files easily. The *SOF Core* is designed to ensure 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 comprises the fundamental block in the SOF architecture that provides the routines for executing and monitoring simulations. It uses Secure SHell (SSH) protocol 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 Hadoop Distributed File System (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 execution of the system, described in Figure 4.12, begins whit a user request. We summarize a SOF loop in the following key phases:

1. ***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.

2. ***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).

3. ***Collect***. When all the simulation executions end, the computation state is synchronized, and the outputs are collected at the SOF core system according to the XML schema defined by the user through a set of messages exchanged between the computational resources and the core system.

4. ***Evaluation Phase***. The system applies the evaluation function to the collected outputs and generates the rating (again in the desired XML format).

After the evaluation phase, the system returns to the selection phase, which, also using the evaluation results obtained during the preceding steps, generates a new set of XML inputs for the next loop. Obviously, the selection function also includes a stopping rule (for instance, an empty set of parameters), which enables the termination of the SO process.

4.4.3 CISD optimization

We have designed and developed a heuristic search algorithm to compute the Day-by-day optimization described in the previous Section 4.4.1. We based our solution on a constructive heuristic that exploits the Tabu search approach [112]. Tabu search algorithm can find out a local optimum by the following steps: compute an initial solution, compute a good neighbor solution by a local search, to improve the value of the objective function. In this kind of search algorithm, it is possible to move the solution to another even if there is no objective function improvement. In this case, the new solution is named *tabu*, and it is stored in a tabu list. In this way, the algorithm can check if a local optimum solution was not already computed. The algorithm has a fixed number of possible tabu solution. The solutions are iteratively computed until a stopping criterion is reached. The algorithm works as follows.

Init phase. First of all, the algorithm tries to merge ports that are close to each other and have the same time windows. To do that, the algorithm performs the following steps.

1. computes d^{max} and d^{min} as the maximum and minimum distances among all ports, respectively.
2. computes pivot distance $\hat{d} = \frac{d^{max} - d^{min}}{PIVOT\_DISTANCE\_RATIO}$ where $PIVOT\_DISTANCE\_RATIO$ is a parameter of the algorithm.
3. for each couple of ports i and j, if distance $d_{ij} \leq \hat{d}$ and time windows of i and j are the same, i and j are considered as the same port.
4. for each couple of ports i and j, compute v^*_{ij} as the best speed (minimum cost) between i and j according to their consumption curve.
5. compute convergence limit $c_{limit} = \frac{tot_p \cdot max_p}{CONV\_LIMIT\_RATIO}$, where tot_p is the total number of feasible ports and max_p is the maximum number of ports that can be visited in the cruise and $CONV\_LIMIT\_RATIO$ is a parameter of the algorithm.

6. creates POP_SIZE initial solution by randomly selecting the first port, where POP_SIZE is a parameter of the algorithm.

Evolution phase. During this phase, the algorithm expands the initial solution until a stopping criterion is met or a blind destination is reached (i.e., no other possible destination are available).

1. for each partial solution up to port i, select all feasible candidates for a successive stop considering a maximum waiting time to enter into the port equals to *MAX_HOURS_TO_ENTER* that is a parameter of the algorithm.

2. for each candidate k, if parameter $USE\_ONLY\_BEST\_SPEEDS = 1$ consider only speed v^*_{ik}, otherwise check if it possible to modify speed in order minimize time window violations.

3. randomly selects next stop among all candidates k.

4. if the selected candidate k^* is a blind destination (i.e., there are no other possible destinations) k^* is inserted in a tabu list.

5. at each iteration checks if at least one stopping criteria is met.

6. destroys and rebuilds from scratch the solution with a probability of *RESET_THRESHOLD* which is a parameter of the algorithm.

Summing up, the algorithm's main parameters, that are iteratively changed for the analysis described in this article, are: *USE_ONLY_BEST_SPEEDS*, *POP_SIZE*, *CONV_LIMIT_RATIO*, *PIVOT_DISTANCE_RATIO*, *MAX_HOURS_TO_ENTER*, *RESET_THRESHOLD*.

Parameter selection optimization process. To discover the parameters configuration that enables to obtain an efficient solution, we have developed the selection function and the evaluation functions for the SOF framework work cycle. The selection function generates a set of parameters. Thereafter the searching algorithm above is executed to generate feasible solutions (according to the current parameters) while the evaluation function computes the cost and the attractiveness of each

solution. The selection function is a pure Java-based genetic algorithm, having a certain number of individuals (*population_size*), which is able to generate a new feasible parameter set for the searching heuristic described above, also exploiting the previous evaluation results. With more details, in the first optimization loop, the algorithm randomly generates a set of parameters. The algorithm takes as input the previous input/output/evaluation results and accordingly generates a new set of parameters in the next steps. The parameter selection process iterates until a stopping criterion is met. In this work, we have used two simple stopping criteria: the optimization loop terminates when the selection function has performed *max_iter* iterations, or the last *max_no_improving_iter* iterations did not improve the solution.

4.4.4 Results

In this section, we describe our results in terms of the quality of the solution obtained by the OvS process using our heuristic tabu search-based algorithm, and we present the computation time of the process varying the number of machines involved in the execution on the AWS cloud infrastructure. We have exploited the Amazon Elastic Map Reduce(EMR) [1] that provides a managed Hadoop framework on the Amazon Elastic Compute Cloud [2] platform, that provides a resizable compute capacity in the cloud. The tests have been performed on a real scenario using 220 ports and 1 ship, and the Mediterranean sea as a maritime area. The other data and parameters range values are computed according to the requirements given by a cursing company.

Quality results We performed 12 different parameters scenario optimization using EMR cluster of 4 EC2 nodes. We changed the input parameters of the selection function varying the three genetic algorithm input:

- $population\_size = 20$;

[1] Amazon Elastic Map Reduce (EMR) `https://aws.amazon.com/en/emr/`

[2] Amazon Elastic Compute Cloud (Amazon EC2) `https://aws.amazon.com/ec2`

- $max\_iter = 20, 30, 40$;
- $max\_no\_improving\_iter = 1, 2, 3, 4, 5, 8, 10, 15, 20$.

The values of $max\_no\_improving\_iter$ for the first 6 tests are obtained by *max_no_improvig_iter* = *max_iter* $\times$ $[0.05, 0.10]$, while for the last 6 tests by *max_no_inproving_iter* = *max_iter* $\times$ $[0.25, 0.50]$.

id test	***population_size***	***max_iter***	***max_no_inproving_iter***	**Executed loops**	**Best loop**	**C**	**A**
1	20	20	1	5	4	218817,36	293
2	20	20	2	11	9	216360,96	292
3	20	30	3	8	5	217938,53	304
4	20	30	3	9	6	218319,12	331
5	20	40	2	-	-	-	-
6	20	40	4	7	3	209174,08	276
7	20	20	5	11	6	207845,81	294
8	20	20	10	20	13	199226,54	352
9	20	30	8	13	10	193345,45	293
10	20	30	15	30	20	209420,05	354
11	20	40	10	20	10	214056,81	328
12	20	40	20	34	14	205052,33	357

Table 4.3: Quality results with different parameter settings. .

The tests are described in the Table 4.3. The selection function generates a set of cardinality $population\_size$ in each iteration. Each generated input is a configuration for the Day-by-Day optimization algorithm. In this experiment we have used the genetic algorithm to change 6 parameters of our algorithm. In particular, the parameters change according to a possible range and are:

- *use_only_best_speed* $\in [0, 1]$
- *pop_size* $\in [40, 45, 50, 55, 60]$
- *conv_limit_ratio* $\in [1, 2, 3, 4, 5]$
- *pivot_distance_ratio* $\in [50, 100, 150, 200]$
- *max_hours_to_enter* $\in [6, 12, 18, 24, 30, 36, 42, 48]$
- *reset_threshold* $\in [0.05, 0.10, 0.15, 0.20]$

Table 4.3 shows the configuration for the selection function and the number of executed optimization loops (Executed loops), the loop that have produced the best results (Best loop), the total cruising cost C and the total value of attractiveness A. As shown in the table 4.3, only the test 5 did not produce a feasible solution before the stopping criteria. The tests 8 and 10 ended because the maximum number of iteration has been reached. The best solution has been achieved by the test 9 and 12. Specifically, the test 9 provides the minimum cruise cost, while the test 12 provides the best attractiveness cruising value.

Performance results. We performed 7 experiments on an EMR cluster to evaluate the scalability of our OvS process for CISD. We exploited a cluster of 32 EC2 instances running the Ubuntu Server 16.04 LTS operating system. Amazon provides different kinds of EC2 instances varying the cost (that is, computed effective computation minutes), the number of CPUs, and the amount of memory. We performed our test with EC2 instance *c4.2xlarge*. The hardware details of this instance type are Xeon E5-2666 v3 processor, 8 vCPUs, and High Network Speed. The cost is 0.398 hourly for Linux on Demand instance and 0.105 hourly for Linux on Spot instance. We present the performance

Nodes	vCPUs	Memory	time(s)	Cost on demand EC2	Cost on demand EMR	Total cost
1	4	15.0 GB	27165	$ 3.00	$ 0.79	$ 3.79
1	8	15.0 GB	23164	$ 2.56	$ 0.68	$ 3.24
2	16	30.0 GB	18695	$ 4.13	$ 1.09	$ 5.22
4	32	60.0 GB	10185	$ 4.50	$ 1.19	$ 5.69
8	64	120.0 GB	5259	$ 4.65	$ 1.23	$ 5.88
16	128	240.0 GB	3610	$ 6.39	$ 1.68	$ 8.07
32	256	480.0 GB	2005	$ 7.09	$ 1.87	$ 8.96

Table 4.4: Time in second and cost ($) required by the 10 loops of the SO process for evaluating 10 input configurations.

results in terms of strong scalability. We have fixed the total amount of computation to 10 optimization loops in which we generate/evaluate 10 input configurations of our algorithm for CISD. We varied the number of nodes from 1 to 32 and, accordingly, we varied the number of vCPU

involved in the computation from 4 to 256.
For each experiment, we compute the total time required to execute the OvS process, the total cost of EC2 instances, the cost of the EMR cluster, and finally, the total cost of the cloud computing infrastructure. All these values are reported in Table 4.4. Results show that the system provides good scalability: Comparing the first and the last test, we have that the time required for the whole computation is reduced by a factor of 13, while the cost of the process is incremented about 2.4 times.

Chapter 5

Conclusion

Computational science is an attractive and active field of research that aims to solve complex problems through a computational approach. Solutions to this problem are computationally expensive and time-consuming. Therefore, the notion of "scalability" becomes central, and the challenge is to improve the current state of solutions in terms of efficiency and effectiveness. However, implementing scalable solutions requires a deep understanding of parallel and distributed computing architectures and paradigms. The cultural background of computational scientists (also called domain experts) does not have these skills. Consequently, they spend most of their time making applications scalable by taking time away from implementing the application logic.

This dissertation discussed frameworks and languages designed to support computational scientists in creating scalable applications efficiently, allowing them to focus on the application's logic without worrying about the implementation aspects of parallel and distributed computing.

One of the main choices when creating computational science applications is the programming language. Many languages support parallel computing constructs for building scalable applications, but they require knowledge of the underlying computing system, which must be managed and maintained. This dissertation discussed FLY, a domain-specific language that aims to reconcile the world of Cloud computing with that of High-Performance Computing. FLY allows the Function-

as-a-Service(FaaS) model provided by different cloud providers to be used as a computing environment transparently and simultaneously in a single application. Using FLY, the domain expert specifies the workflow (the application logic) they want to execute and on which cloud platform(s) they want to execute it (setting only the cloud access credentials and some configuration parameters). Through FLY, complex scientific workflows can be executed in a scalable way. Besides showing the language programming model and the source-to-source compiler, we presented a performance evaluation on a popular benchmark for distributed workflow computing frameworks, WordCount. Furthermore, we showed a collection of three algorithms (Montecarlo evaluation of π, K-Nearest Neighbor, and Smith-Waterman) with an analysis of their performance results on Amazon AWS. The results of these analyses show that considerable speedup can be achieved through FLY at an affordable cost, much less than that of a MapReduce cluster or Virtual Machines provided by Amazon AWS. Finally, an example of integrating a FLY program into an Optimization via Simulation process for solving a Customer Allocation problem is shown. We found that our solution allows us to achieve a significant speedup in execution time, confirming the ability of FaaS to be effective and efficient for developing a large-scale scientific workflow that takes advantage of the OvS techniques. This study's findings have to be seen in the light of a well-known drawback related to FaaS use regarding computational time and memory limits. Although these limits are getting larger and larger, very long-running simulation executions cannot be performed at the moment.

A methodology that is having some success within Computational Science is agent-based modeling (ABM). The success of this tool has led to the creation of several libraries of ABM simulations and the development of simulations with a huge number of agents. This dissertation showed Rust-AB, an open-source library for building massive agent simulations through the Rust language. We implemented an example of Rust-AB simulation, the Boids model, to investigate the sequential performance of Rust-AB in comparison with MASON. The results are promising and exhibit performance-enhancing compared to the MASON toolkit, particularly for simulations with a high agent den-

sity. We performed even a parallel performance evaluation of Rust-AB, always using the Boids model, increasing the number of threads used to simulate 500000 and 100000 agent. We have obtained promising results showing that the parallel version of Rust-AB has a speedup of x14 compared to the sequential version. This speedup can be further improved by making optimal use of Rust's capability.

The growing interest in the development of ABM has led to the need to optimize these models to reflect the reality you want to simulate. The Optimization via Simulation (OvS) collects various methodologies to optimize the simulation parameters efficiently, but it demands a remarkable amount of computational resources. In this dissertation, three tools are presented that exploit the hardware and software features of three different types of distributed infrastructures and Cloud infrastructures. HSO, a tool that uses the power of a heterogeneous distributed system, composed of several CPUs and GPUs, to develop general OvS processes, has been presented. We have evaluated the performance of HSO on two use cases: Spread of Information and Spread of Influence, comparing the OvS process time required to complete different experiments using a heterogeneous system composed of 4CPUs and 4GPUs. In the first use case, we exploited an ABM simulation, written in NetLogo, and a genetic algorithm, developed in Python, to study the problem of Spread of Information. In the second use case, we developed a C based and Nvidia CUDA based simulation to model the Spread of Influence on a social network, and we used a Java OCBA implementation for the optimization phase. In the context of distributed OvS, in this dissertation, we presented a tool that aims to provide an automatic calibration tool for ABM simulations. We developed this kind of tool combining the popular MASON and ECJ libraries and have shown how their combination can produce powerful, fully-featured model calibration facilities with special capabilities of interest to the agent-based modeler. A performance evaluation of several calibration processes using various optimization techniques, like Genetic algorithms, Evolutionary Strategy and CMAES, and simulations, like the Flocker model and the Refugee model, was presented. Finally, a tool for solving the Cruise Itinerary Schedule Desing problem was discussed, which realized a distributed OvS process, through the SOF framework, using a MapReduce clus-

ter provided by Amazon Cloud provider. We evaluated the proposed solution on a real scenario in terms of both the quality of the obtained solutions (attractiveness and costs) and the scalability/cost efficiency on Amazon AWS. Results show that the strategy can identify good solutions and that the cloud infrastructure's use improves the strategy's timing without severely impacting costs.

Bibliography

[1] D. A. Reed, R. Bajcsy, M. A. Fernandez, J.-M. Griffiths, R. D. Mott, J. Dongarra, C. R. Johnson, A. S. Inouye, W. Miner, M. K. Matzke *et al.*, "Computational science: ensuring america's competitiveness," PRESIDENT'S INFORMATION TECHNOLOGY ADVISORY COMMITTEE ARLINGTON VA, Tech. Rep., 2005.

[2] M. D. Hill, *What is Scalability?* Boston, MA: Springer US, 1992, pp. 89–96.

[3] C. Jones, P. O'Hearn, and J. Woodcock, "Verified software: a grand challenge," *Computer*, vol. 39, no. 4, pp. 93–95, 2006.

[4] P. Prabhu, T. B. Jablin, A. Raman, Y. Zhang, J. Huang, H. Kim, N. P. Johnson, F. Liu, S. Ghosh, S. Beard, T. Oh, M. Zoufaly, D. Walker, and D. I. August, "A survey of the practice of computational science," in *State of the Practice Reports*. New York, NY, USA: Association for Computing Machinery, 2011.

[5] H. El-Rewini and M. Abd-El-Barr, *Advanced computer architecture and parallel processing*. John Wiley & Sons, 2005, vol. 42.

[6] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *Proceedings of the 2nd International Workshop on Software and Performance*. New York, NY, USA: Association for Computing Machinery, 2000, p. 195–203.

[7] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2005.

[8] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, 2018.

[9] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, 2008. [Online]. Available: http://www.R-project.org

[10] G. Rossum, "Python reference manual," Tech. Rep., 1995.

[11] J. Bezanzon, S. Karpinski, V. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," in *Lang.*{*NEXT*}, 2012.

[12] D. M Ritchie, "The limbo programming language," *Inferno Programmer's Manual*, 2018.

[13] B. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, 2007.

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, 1995.

[15] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scriptin," in *Parallel Computing*, 2011.

[16] J. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. Foster, "Swift/t: Large-scale application composition via distributed-memory dataflow processing," in *Cluster, Cloud and Grid Comp., 13th IEEE/ACM I. Symposium.*, 2013.

[17] R. Reuillon, M. Leclaire, and S. Rey-Coyrehourcq, "Openmole, a workflow engine specifically tailored for the distributed exploration of simulation models," *Future Generation Computer Systems*, vol. 29, pp. 1981–1990, 2013.

[18] S. Abar, G. Theodoropoulos, P. Lemarinier, and G. O'Hare, "Agent Based Modelling and Simulation tools: A review of the state-of-art software," *Computer Science Review*, 2017.

[19] F. Mahé, T. Rognes, C. Quince, C. de Vargas, and M. Dunthorn, "Swarm: robust and fast clustering method for amplicon-based studies," *PeerJ*, 2014.

[20] M. Resnick, "StarLogo: An Environment for Decentralized Modeling and Decentralized Thinking," in *Conference Companion on Human Factors in Computing Systems*, 1996.

[21] S. Tisue and U. Wilensky, "Netlogo: A simple environment for modeling complexity," in *in International Conference on Complex Systems*, 2004, pp. 16–21.

[22] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, "Complex adaptive systems modeling with Repast Simphony," *Compl. Adap. Syst. Modeling*, 2013.

[23] S. Luke, R. Simon, A. Crooks, H. Wang, E. Wei, D. Freelan, C. Spagnuolo, V. Scarano, G. Cordasco, and C. Cioffi-Revilla, "The MASON simulation toolkit: Past, present, and future," in *International Workshop on Multi-Agent-Based Simulation (MABS)*, 2018.

[24] M. Holcombe, S. Coakley, and R. Smallwood, "A general framework for agent-based modelling of complex systems," in *Proceedings of the European conference on complex systems*, 2006.

[25] N. Collier and M. North, "Parallel agent-based simulation with Repast for High Performance Computing," *SIMULATION*, 2013.

[26] 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," *IEEE 30th I. Parallel and Distributed Processing Symposium*, 2016.

[27] P. Richmond and M. K. Chimeh, "FLAME GPU: Complex System Simulation Framework," in *2017 International Conference on High Performance Computing Simulation*, 2017.

[28] G. N. Abelson H. and R. L., *Logo Manual*, 1974.

[29] E. L. Rounds, E. O. Scott, A. S. Alexander, K. A. De Jong, D. A. Nitz, and J. L. Krichmar, "An evolutionary framework for replicating neurophysiological data with spiking neural networks," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2016, pp. 537–547.

[30] S. Venkadesh, A. O. Komendantov, S. Listopad, E. O. Scott, K. De Jong, J. L. Krichmar, and G. A. Ascoli, "Evolving simple models of diverse intrinsic dynamics in hippocampal neuron types," *Frontiers in neuroinformatics*, vol. 12, p. 8, 2018.

[31] D. Mayer, B. Kinghorn, and A. Archer, "Differential evolution–an easy and efficient evolutionary algorithm for model optimisation," *Agricultural Systems*, vol. 83, no. 3, pp. 315–328, 2005.

[32] D. Mongus, B. Repnik, M. Mernik, and B. Žalik, "A hybrid evolutionary algorithm for tuning a cloth-simulation model," *Applied Soft Computing*, vol. 12, no. 1, pp. 266–273, 2012.

[33] (Accessed Dec, 2020) ParadisEO. http://paradiseo.gforge.inria.fr.

[34] E. Scott and S. Luke, "Ecj at 20: Toward a general metaheuristics toolkit," in *GECCO '19 Companion*, 2019.

[35] J. Ozik, N. T. Collier, J. Wozniak, and C. Spagnuolo, "From desktop to large-scale model exploration with swift/t," in *Proceedings of the 2016 Winter Simulation Conference*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 206–220.

[36] L. Gulyás, A. Szabó, R. Legéndi, M. T., R. Bocsi, and G. Kampis, "Tools for Large Scale (Distributed) Agent-Based Computational Experiments," in *Proceedings of CSSSA*, 2011.

[37] (Accessed Dec, 2020) OptTek metaheuristic optimization. http://www.opttek.com.

[38] M. Carillo, G. Cordasco, F. Serrapica, V. Scarano, C. Spagnuolo, and P. Szufel, "Sof: Zero configuration simulation optimization framework on the cloud," in *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*. IEEE, 2016, pp. 341–344.

[39] (Accessed Dec, 2020) Dakota. https://dakota.sandia.gov.

[40] G. Cordasco, M. D'Auria, A. Negro, V. Scarano, and C. Spagnuolo, "Fly: A domain-specific language for scientific computing on faas," *Lecture Notes in Computer Science*, 2020.

[41] G. Cordasco, M. D'Auria, A. Negro, V. Scarano, and C. Spagnuolo, "Toward a domain-specific language for scientific workflow-based applications on multicloud system," *Concurrency and Computation: Practice and Experience*, 2020.

[42] A. Antelmi, G. Cordasco, M. D'Auria, D. De Vinco, A. Negro, and C. Spagnuolo, "On evaluating rust as a programming language for the future of massive agent-based simulations," in *Methods and Applications for Modeling and Simulation of Complex Systems*. Singapore: Springer Singapore, 2019, pp. 15–28.

[43] G. Cordasco, M. D'Auria, C. Spagnuolo, and V. Scarano, "Heterogeneous scalable multi-languages optimization via simulation," *Communications in Computer and Information Science*, 2018.

[44] M. D'Auria, E. Scott, R. Lather, J. Hilty, and S. Luke, "Distributed, automated calibration of agent-based model parameters and agent behaviors," *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, 2020.

[45] M. D'Auria, E. Scott, R. Lather, J. Hilty, and S. Luke, "Assisted parameter and behavior calibration in agent-based models with distributed optimization," *Lecture Notes in Computer Science*, 2020.

[46] M. Carillo, G. Cordasco, F. Serrapica, V. Scarano, C. Spagnuolo, and P. Szufel, "Distributed simulation optimization and parameter exploration framework for the cloud," *Simulation Modelling Practice and Theory*, vol. 83, pp. 108 – 123, 2018.

[47] M. Carillo, M. D'Auria, F. Serrapica, C. Spagnuolo, C. Caligaris, and M. Fabiano, "Large-scale optimized searching for cruise itinerary scheduling on the cloud," *2019 International Conference on Optimization and Applications, ICOA 2019*, 2019.

[48] A. Shawish and M. Salama, "Cloud computing: Paradigms and tecn." *Inter-cooperative collective intel.: Tech. and app.*, 2014.

[49] G. Cordasco, V. Scarano, and C. Spagnuolo, "Distributed mason: A scalable distributed multi-agent simulation environment," *Simulation Modelling Practice and Theory*, 2018.

[50] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, 2008.

[51] A. J. Ferrer, D. G. PÃ©rez, and R. S. GonzÃ¡lez, "Multi-cloud platform-as-a-service model, functionalities and approaches," *Procedia Computer Science*, 2016.

[52] I. Brugere, B. Gallagher, and T. Y. Berger-Wolf, "Network structure inference, a survey: Motivations, methods, and applications," *ACM Computing Surveys (CSUR)*, 2018.

[53] G. Cordasco, R. De Chiara, F. Raia, V. Scarano, C. Spagnuolo, and L. Vicidomini, "Designing computational steering facilities for distributed agent based simulations," *Proc. of the 2013 ACM SIGSIM Principles of Advanced Discrete Simulation*, 2013.

[54] D. Ghosh, *DSLs in Action*, 1st ed. USA: Manning Publications Co., 2010.

[55] D. E. Verna, "Extensible Languages: Blurring the Distinction between DSL and GPL," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, 2012.

[56] JetBrains, *MPS: Domain-Specific Language Creator by JetBrains*, 2020 (accessed Dec, 2020). [Online]. Available: https://www.jetbrains.com/mps/

[57] E. Project, *Xtext, language engineering for everyone!*, 2020 (accessed Dec, 2020). [Online]. Available: http://www.eclipse.org/Xtext

[58] H. Hwang, G. C. Fox, and J. J. Dongarra, *Distr. and Cloud Comp. from Parallel Processing to the Internet of Things*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[59] Amazon.com Inc. (accessed: Dec,2020) Amazon web services - aws lambda. [Online]. Available: http://aws.amazon.com/lambda

[60] M. Corporation, *Microsoft Azure Functions*, 2020 (accessed Dec, 2020). [Online]. Available: http://azure.microsoft.com/services/functions

[61] Google Inc. (accessed Dec, 2020) Google cloud functions. [Online]. Available: http://cloud.google.com/functions

[62] IBM. (accessed Dec, 2020) Ibm bluemix. [Online]. Available: http://www.ibm.com/cloud-computing/bluemix

[63] Apache Software Foundation. (accessed Dec, 2020) Apache openwhisk. [Online]. Available: http://openwhisk.apache.org

[64] platform9, *Fission*, accessed Dec, 2020. [Online]. Available: https://docs.fission.io/0.8.0/

[65] Oracle, *Fn Project*, accessed Dec, 2020. [Online]. Available: https://fnproject.io/index.html

[66] Bitnami. (accessed Dec, 2020) Kubeless. [Online]. Available: https://kubeless.io/

[67] I. Baldini, P. C. Castro, K. Shih-Ping Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," *CoRR*, 2017.

[68] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 405–410.

[69] M. Stigler, *Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud.* Apress, 2018.

[70] Tablesaw, *JTablesaw is a Java dataframe similar to Pandas in Python, and the R data frame.*, Accessed Dec, 2020. [Online]. Available: https://github.com/jtablesaw/tablesaw

[71] dataframe js, *DataFrame-js provides an immutable data structure for javascript and datascience.*, Accessed Dec, 2020. [Online]. Available: https://github.com/Gmousse/dataframe-js

[72] LocalStack, *LocalStack*, accessed Dec, 2020. [Online]. Available: https://localstack.cloud/

[73] H. Lee and G. Fox, "Big data benchmarks of high-performance storage systems on commercial bare metal clouds," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019.

[74] *The 20 newsgroup dataset.*, accessed Dec 2020. [Online]. Available: http://people.csail.mit.edu/jrennie/20Newsgroups/

[75] (Accessed Dec, 2020) OPT Net. https://www.act-operationsresearch.com/opt-net-planning-and-design-inventory-order-fulfillment-software/.

[76] (Accessed Dec, 2020) ACTOR Company. https://www.act-operationsresearch.com/.

[77] S. Amaran, N. V. Sahinidis, B. Sharda, and S. J. Bury, "Simulation optimization: a review of algorithms and applications," *Annals of Operations Research*, 2015.

[78] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of Function-as-a-Service software development in industrial practice," *Journal of Systems and Software*, 2019.

[79] R. Pinto and M. T. Vespucci, "Progettazione della rete logistica," in *Modelli decisionali per la produzione, la logistica ei servizi energetici*, 2011.

[80] L. Perron and V. Furnon. (Accessed Dec, 2020) Or-tools. Google. [Online]. Available: https://developers.google.com/optimization/

[81] ISISLab, "FLY CVRP program," Accessed Dec, 2020. [Online]. Available: https://github.com/spagnuolocarmine/FLY-language/blob/master/examples/cvrp/src/main/fly/fly/cvrp.fly

[82] S. Thurner, P. Klimek, and R. Hanel, *Introduction to the theory of complex systems*. Oxford University Press, 2018.

[83] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, 2000.

[84] V. Tejaswi, P. Bindu, and P. Thilagam, "Diffusion models and approaches for influence maximization in social networks," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2016.

[85] B. Heath, R. Hill, and F. Ciarallo, "A survey of agent-based modeling practices (January 1998 to July 2008)," *Journal of Artificial Societies and Social Simulation*, 2009.

[86] C. Macal and M. North, "Tutorial on agent-based modeling and simulation part 2: How to model with agents," in *Proceedings of the 2006 Winter Simulation Conference*, 2006, pp. 73–83.

[87] G. Cordasco, A. Mancuso, F. Milone, and C. Spagnuolo, "Communication Strategies in Distributed Agent-Based Simulations: The Experience with D-Mason," in *Euro-Par 2013: Parallel Processing Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 533–543.

[88] S. Carmine, "Rust–AB: An Agent Based Simulation engine in Rust," https://github.com/spagnuolocarmine/abm, 2018.

[89] C. W. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioral Model." *Computer Graphics (ACM)*, 1987.

[90] J. Wejdenstål, *DashMap*, 2020 (accessed Dec, 2020). [Online]. Available: https://github.com/xacrimon/dashmap

[91] A. M. Law and W. D. Kelton, *Simulation modeling and analysis*. McGraw-Hill New York, 2007, vol. 3.

[92] E. Tekin and I. Sabuncuoglu, "Simulation optimization: A comprehensive review on theory and applications," *IIE Trans.*, vol. 36, pp. 1067–1081, 2004.

[93] Y. Carson and A. Maria, "Simulation optimization: Methods and applications," in *Proceedings of the 29th Conference on Winter Simulation*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 118–126.

[94] M. Fu, *Handbook of Simulation Optimization*. Springer Publishing Company, Incorporated, 2014.

[95] V. Donaldson, F. Berman, and R. Paturi, "Program speedup in a heterogeneous computing network," *Journal of Parallel and Distributed Computing*, vol. 21, pp. 316 – 322, 1994.

[96] J. Goux, K. Kulkarni, J. Linderoth, and M. Yoder, "An enabling framework for master-worker applications on the computational grid," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, 2000, pp. 43–50.

[97] E. Gabriel, F. G.E., G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 2004, pp. 97–104.

[98] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2003, p. 137–146.

[99] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, 2008.

[100] S. Gao, H. Xiao, E. Zhou, and W. Chen, "Optimal computing budget allocation with input uncertainty," in *Proceedings of the 2016 Winter Simulation Conference*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 839–846.

[101] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[102] A. Heppenstall, N. Malleson, and A. Crooks, ""space, the final frontier": How good are agent-based models at simulating individuals and space in cities?" *Systems*, vol. 4, no. 1, 2016.

[103] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii," in *International conference on parallel problem solving from nature*. Springer, 2000, pp. 849–858.

[104] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[105] S. Luke and L. Spector, "Evolving teamwork and coordination with genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, 1996, pp. 141–149.

[106] S. XiaoDong, J. Yue, and T. Peng, "Marketing research and revenue optimization for the cruise industry: a concise review," *International Journal of Hospitality Management*, vol. 30, pp. 746–755, 2011.

[107] L. Gui and A. P. Russo, "Cruise ports: a strategic nexus between regions and global lines evidence from the mediterranean," *Maritime Policy & Management*, vol. 38, no. 2, pp. 129–150, 2011.

[108] K. Wang, S. Wang, L. Zhen, and X. Qu, "Cruise shipping review: operations planning and research opportunities," *Maritime Business Review*, vol. 1, no. 2, pp. 133–148, 2016.

[109] S. Wang, K. Wang, L. Zhen, and X. Qu, "Cruise itinerary schedule design," *IISE Transactions*, vol. 49, no. 6, pp. 622–641, 2017.

[110] G. Di Pillo, M. Fabiano, S. Lucidi, and M. Roma, "A two-objective optimization of ship itineraries for a cruise company," in *Proceedings of the 19th European Conference on Mathematics for Industry*, June 2016.

[111] P. Ngatchou, A. Zarei, and A. El-Sharkawi, "Pareto multi objective optimization," in *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*, 2005, pp. 84–91.

[112] F. Glover and M. Laguna, *Tabu Search*. New York, NY: Springer New York, 2013, pp. 3261–3362.

La borsa di dottorato è stata cofinanziata con risorse del
Programma Operativo Nazionale Ricerca e Innovazione 2014-2020 (CCI 2014IT16M2OP005),
Fondo Sociale Europeo, Azione I.1 “Dottorati Innovativi con caratterizzazione Industriale”

UNIONE EUROPEA
Fondo Sociale Europeo