https://doi.org/10.1016/j.simpat.2018.09.002

Distributed MASON: A Scalable Distributed Multi-agent Simulation Environment☆

Gennaro Cordasco[a,*], Carmine Spagnuolo[b,*], Vittorio Scarano[b]

[a] *Dipartimento di Psicologia, Università degli Studi della Campania "Luigi Vanvitelli", e-mail: gennaro.cordasco@unicampania.it*
[b] *Dipartimento di Informatica, Università degli Studi di Salerno, e-mail: cspagnuolo@unisa.it, vitsca@dia.unisa.it*

Abstract

Computational Social Science (CSS) involves interdisciplinary fields and exploits computational methods, such as social network analysis as well as computer simulation with the goal of better understanding social phenomena.

Agent-Based Models (ABMs) represent an effective research tool for CSS and consist of a class of models, which, aim to emulate or predict complex phenomena through a set of simple rules (i.e., independent actions, interactions and adaptation), performed by multiple agents. The efficiency and scalability of ABMs systems are typically obtained distributing the overall computation on several machines, which interact with each other in order to simulate a specific model. Unfortunately, the design of a distributed simulation model is particularly challenging, especially for domain experts who sporadically are computer scientists and are not used to developing parallel code.

D-MASON framework is a distributed version of the MASON library for designing and executing ABMs in a distributed environment ensuring scalability and easiness. D-MASON enable the developer to exploit the computing power of distributed environment in a transparent manner; the developer has to do simple incremental modifications to existing MASON models, without re-designing them.

☆ An extended abstract of this paper was presented at the 1st IEEE Workshop on Parallel and Distributed Processing for Computational Social Systems (ParSocial 2016) [1].

*Corresponding author

This paper presents several novel features and architectural improvements introduced in the D-MASON framework: an improved space partitioning strategy, a distributed 3D field, a distributed network field, a decentralized communication layer, a novel memory consistency mechanism and the integration to cloud environments.

Full documentation, additional tutorials, and other material can be found at `https://github.com/isislab-unisa/dmason` where the framework can be downloaded.

Keywords: Agent-based Simulation, Parallel Computing, Distributed Computing, Scalable Computational Science, Cloud Computing.

---

1. Introduction

1.1. Compuational Social Science

Computational Science (CS) [2] is an emergent research field that exploits data-intensive computing and analysis to study real-world complex problems. CS aims to tackle problems using the predictive capability to support the traditional experimentation and theory, according to a computational approach to problem-solving. This new discipline in science combines computational thinking, modern computational methods, hardware and software to face (complex) problems, overcoming the limitations of traditional ways. Computational Social Science (CSS) is a branch of the CS that deals with social and behavioral dynamics. CSS involves interdisciplinary fields and exploits computational methods, such as social network analysis as well as computer social simulations to (i) collect social data; (ii) understand social dynamics; (iii) model the behavior of social systems; (iv) predict the behavior of such systems in order to answer what-if questions.

With emerging collection techniques for big datasets, CSS is facing a paradigm shift: fundamental changes are occurring related to research methods and the ways they can be applied too, in order to collect and analyze data with an unprecedented breadth, depth, and scale.

This paper provides computational tools and methods in order to setup and runs big scale Agent-Based Models (ABMs) simulations for CSS.

1.2. Agent Based Simulations

Computer simulation represents a novel methodology for research and development in many fields such as ecology, economics, physics, etc. In some fields, such as social sciences, computer social simulation enables scientist to reconcile the descriptive approach, used in the social sciences and the formal approach, used in the hard sciences. ABMs are modeling techniques designed to emulate or predict complex phenomena through independent actions and interactions performed by multiple agents.

An ABM consists of three components: agents, relations, and rules. The agents model a population, the relations define potential interactions among agents while the rules describe the behavior of an agent as a result of an interaction. ABMs can be used to represent a biological model like a flock of birds, as well as emergencies or evacuations scenarios or, more in general, computational sociology experiments. In social sciences, ABMs are seen as a fundamental tool [3] to model the generative approach [4], essential for understanding complex social phenomena. The relevance and effectiveness of ABMs have been recently recognized also in policymaking and economics [5]. Cioffi-Revilla in [6] raised a very challenging question: "How does a variation in the number of interacting units (grid size) affect the main results of an agent-based simulation?". This question motivates the study of complex phenomena at different scales.

1.3. Distributed ABM simulations

The idea behind ABMs is to exploit the KISS principle "keep it simple and stupid", that is, a small set of simple rules may be sufficient to reproduce complex behaviors [7]. Unfortunately, the emergence of such complex phenomena often requires the simulation of several models, with different parameters, on massive populations that require highly demanding computational software implementations.

On the other hand, distributed computing has become a consolidated technology considering the exponential growth of local networking and Internet connectivity coupled with the unused computational power available on average desktop PCs and the scalability limits of centralized approaches. The most common approach to Distributed Computing is based on the Master/Slave paradigm. In this paradigm, the user sends a complex job on a Master computer, which orchestrates the work to be done by Slaves (or Workers). Specifically, the Master divides the job into a set of independent tasks. Then, each task is sent to an available slave, which processes the task and sends back the output. Finally, the master obtains the results of the whole computation and combine the outputs.

One of the most challenging aspects in distributed computing is to balance the workload among the machines that provide computation (i.e., the slaves). Indeed, due to synchronization constraints, the entire computation advances with the speed of the slowest machine, which may represent a bottleneck for the overall system performance. Software for computer simulation, in the context of CSS, should be able to exploit both High-performance computing (HPC) systems, characterized by a large number of homogeneous computing machines, and heterogeneous systems.

Several distributed ABM solutions have been proposed in order to help the development and the testing of new models, by providing supporting tools that improve the efficiency and the effectiveness of managing (distributed) simulations [8, 9, 10].

Another interesting and emerging computing paradigm that is going to provide a significant impact in CSS is cloud computing [11]. Cloud computing offers the advantage of delivering computation and data storage without the user having to have an in depth knowledge of the technology.

Cloud providers offer different services like managed infrastructures (IaaS - Infrastructure as a Service) as well as managed platforms such as a key-value storage or a relational database (PaaS - Platform as a Service). The offered IaaS and PaaS services enable Cloud computing applications that range from

simple data backup to the possibility of deploying entire computing clusters or data centers in a remote environment. The flexibility, cost-efficiency, scalability, accessibility, and user-friendliness of cloud services make it also an attractive model to address computational challenges in the scientific community.

1.4. Distributed (D-)MASON a framework for scalable ABM simulation

dm [9] is a distributed version of MASON [12], a well-known and popular library for designing and executing ABMs. D-MASON is innovative because the parallelization is implemented at framework-level without modifying the interface on which simulations are implemented, so that scientists that use the framework (e.g., a domain expert with limited knowledge of distributed programming) are able to exploit a distributed environment using D-MASON as a sequential framework (being only minimally aware of such distribution). D-MASON has been designed, using MASON core, and is inspired by two widespread toolkits for ABMs, NetLogo [13] and Repast [8].

This paper introduces the latest functionalities of D-MASON, which significantly improve both the efficiency and the effectiveness of the framework.

2. Related Works

The computational requirement of simulators usually exceeds the capability of conventional sequential computer [14]. Several parallel and distributed simulation systems have been developed in order to speed up either the tuning or the execution of models. Two models have been considered [15]: Discrete Event Simulations (DES) have been showed to be efficient, scalable and embarrassingly parallelizable especially on models characterized by a bounded amount of tasks' interdependencies, but are not well suited to complex agent-based applications; ABMs, indeed, are much more expressive. They implement the *sense-think-act* paradigm, which offers an easy way to design agent programming model (only the definition of a communication protocol and a set of simple rules are required) but unfortunately, due to the high level of agents' interdependencies, these models [16, 12, 8] are not easy to parallelize.

Table 1: Distributed Agent-based simulation systems.

ABMs Software Tools	**System development's aim**	**License**	**Source Code**	**Type of ABM on Interactions Behaviour**	**Comm. Paradigm**
AnyLogic **??**	Interactive simulation in manufacturing, business strategy, transportation, social sciences, economics, networks	Closed Source	Java	Agents implemented as Java Class	-
AOR Simulation	Brandenburg University of Technology (Germany)	GPL	Java	Cognitive agents	Shared Memory
BSim	Cell Simulation Labs GitHub	MIT	Java	Reactive Behavioural agents	Shared Memory
CybelePro	Intelligent Automation Inc.	Open Source (Academic) and Proprietary	Java	Reactive agents	Shared Memory
Echo	Santa Fe Institute	Open Source	C	Adaptive Evolutionary agents	Shared Memory
FLAME	University of Sheffield (United Kingdom)	GNU Lesser General	C	Agents as objects characterized by state, functions, and set of variables	MPI
FLAME GPU	University of Sheffield (United Kingdom)	GNU Lesser General	C & CUDA	Reactive processing agents (X-Machines)	Shared Memory
GridABM	Institute for Advanced Study	GPL	Java	Reactive BDI agents	Java Socket
MASS	g6gTech Inc.	Proprietary and Free version	Java	Agents as Java class	-
Pandora	Xavier Rubio-Campillo	Closed source, Free	Microsoft .Net	Agents as C++ classes	-
Repast-HPC	Argonne National Lab.	Open Source	C++	Reactive Agents	MPI

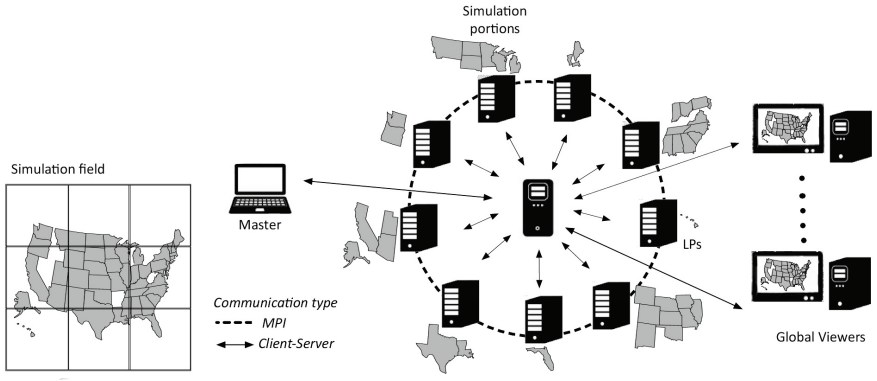


Figure 1: D-MASON Architecture.

Several distributed ABMs frameworks enable to manage explicitly the distribution of agents on several computing nodes, in order to get the most from the efficiency point of view. The D-MASON framework-level approach is different since it brings effectiveness and simplicity: scientists who use the framework can be mostly unaware of the distribution of agents. Previous works on distributed frameworks, such as [17, 18, 19], were focused on the implementation and the architecture of a distributed agent model (dealing with lazy synchronization etc.), while D-MASON has been designed with the purpose of hiding, as much as possible, the details of the architecture.

Distributed and parallel agent-based simulation systems are described by the communication paradigm used and parallel architecture exploited. The comparison table 1 shows the most popular software packages available, as described in [20]. They have been already studied, from a different point of view, in [21].

3. D-MASON

D-MASON has been conceived for the need to improve the efficiency and scalability of ABMs in a distributed setting where computing resources are scarce, heterogeneous, not centrally managed. Moreover, the multidisciplinary of the teams that use ABMs, often, places an important emphasis on easiness of development, thereby suggesting a compromise between efficiency and impact

by acting at the framework-level [9].

D-MASON is able to execute the ABMs within a distributed environment, thereby achieving both efficiency and effectiveness. The D-MASON approach is cost-effective since it provides a high degree of backward-compatibility with MASON simulations (only a few changes are needed in the source code of an existing MASON application).

D-MASON is based on a Master/Slaves paradigm that exploits a space partitioning approach in order to decompose the workload: the master application partitions the space to be simulated (i.e., a MASON field) into cells (see Figure 1). Each slave (worker) executes one or more Logical Processors (LPs), according to its computational capabilities, which provide the computational power of the system. The master establishes a one to one mapping between LPs and cells and accordingly each LP is in charge of:

- simulating the agents that belong to the assigned cell;
- handling agent's handoff (i.e. agent's migration between adjacent cells);
- managing the communication and synchronization between adjacent cells (this information exchange is required in order to let the simulation run consistently).

3.1. D-MASON *Design Issues*

The D-MASON design was done with the aim of addressing the following issues: Work partitioning, Load Balancing, Communication, Synchronization, and Reproducibility.

Work partitioning

A challenging problem for the design of parallel/distributed algorithm is the decomposition of the whole job to a set of tasks to be assigned to a set of LPs (see [22] for a comprehensive presentation). In the case of ABMs, a straightforward approach consists to assign a fixed number of agents to each available LP. This approach, named *agents partitioning*, provides a balanced workload but is

very demanding in terms of communication overhead (required for task synchronization). Indeed, at each simulation step, each agent can interact with other agents, then, in principle, an all–to–all communication among LPs is required.

Considering that ABMs emulates real models, where agent's limited perception enable to bound the range of interaction to a fixed size neighborhood named *Area of Interest* (AOI), a *space partitioning* approach has been proposed [23, 24] in order to reduce the communication effort. Using this approach the Job decomposition is done partitioning the field into a set of cells (one for each LP).

Since the AOI of an agent is small compared with the size of a cell, the communication is limited to local messages (messages between LPs, managing adjacent spaces). On the other hand, in the space partitioning approach, agents can migrate between cells. In D-MASON, by design, agents are allowed to migrate only between adjacent cells. This is consistent with a large family of models (e.g., biology-inspired models) that do not need any kind of *"teleportation"*.

Load Balancing

The problem of the *space partitioning* approach is that since agents can migrate between cells, the association between LPs and agents changes during the simulation. Moreover, the space partitioning approach does not guarantee the load balancing and this need to be addressed by the application. Since in D-MASON the simulation is synchronized after each simulation step, the system advances with the speed provided by the slowest LP in the system. For this reason, it is necessary to balance the load among workers in order to achieve efficiency.

Communication

In D-MASON, the communication between LPs is based on the Publish/-Subscribe (P/S) design pattern: a multicast channel is assigned to each cell/LP; Each LP then simply subscribes to the topics associated with its adjacent cells

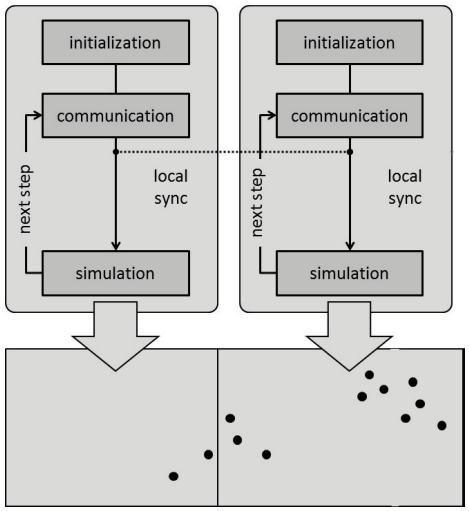


Figure 2: D-MASON LPs' synchronization.

in order to receive relevant message updates. Other channels are also used for the communication between the master application and the LPs. D-MASON has been designed to be used with *any* Message Oriented Middleware that implements the P/S pattern. Moreover, an implementation of the P/S pattern which exploits Java Message Service (JMS) and Apache ActiveMQ Server [25] as JMS provider has been provided in the first version of D-MASON.

Synchronization

In order to ensure consistency of a parallel implementation compared to the sequential one, during the simulation, every LP must continuously gather information on adjacent cells. In D-MASON the simulation proceeds in discrete steps, each simulation step is formed by three phases: *communication, synchronization* and *simulation* (see Figure 2). During the communication phase, the LP sends to its neighbors (i.e., the LPs responsible for its adjacent cells) the information about the agents that are migrating to them as well as the agents that may fall into the AOI of an adjacent cell (ghost agents). D-MASON uses a *conservative–synchronization* approach to achieve a consistent integration of the distributed simulations: this information obtained during the communication

phase is locally synchronized so that as soon as an LP obtains the information regarding all the adjacent cells, at a certain step $t-1$, the simulation phase for step t can start (see Figure 2).

Reproducibility

Reproducibility and repeatability are paramount objectives of the research areas interested in the ABMs. The ability to repeat an experiment is a distinctive feature of ABMs and enable to make different observations as well as a precise tuning of parameters without the risk of any interference. A good approach to achieve reproducibility is to design the simulation in such a way that agents are updated simultaneously and in discrete time (synchronization snapshot) [26]. Using this approach the simulation becomes embarrassingly parallelizable (there are no dependencies between agents' state), each simulation step can be executed in parallel overall the agents. Moreover, the order in which agents are scheduled does not interfere with the results. The deterministic reproducibility of results is hard to achieve in a distributed environment: some simulations, especially those that evolve using a random component, still require a mechanism that allows scheduling agents always in the same order and with the same random source. To achieve reproducibility, in D-MASON, each LP uses its own copy of the random source (e.g., `MersenneTwisterFaster`) and the scheduler ensures that the agents are always elaborated in the same arbitrary order.

3.2. D-MASON *Software Architecture*

D-MASON architecture is based on four design requirements: *efficiency* for exploiting hardware architecture, *effectiveness* for modeling different kind of ABM, *usability* from the users experience point of view and *correctness* of the results. In order to meet these requirements, D-MASON is composed of four functional blocks (or layers):

1. the *Distributed Simulation (DS)* layer, which adds some features to the MASON simulation layer enabling the distribution of the simulation work

on multiple, even heterogeneous machines.

2. the *Communication* layer, which provides communication functionalities to other layers.

3. the *Visualization* layer, which enables the developers to write ABM simulation visualization.

4. the *System Management* layer, which provides user-friendly facilities to manage simulations on a distributed system as well as on a cloud environment.

The DS layer consists of two main packages: *Engine* and *Field*, maintaining the same structure as MASON in order to provide to MASON developers a friendly environment. The *Engine* package, consists of three objects:

- `DistributedState` represents the state of the simulation in a distributed environment and includes: a cell identifier; a method to create a reproducible sequence of agent's identifiers; a method to retrieve the implementation of the field used by the model; a method to add agents in the cell (this method enables the migration of agents between cells).

- `DistributedMultiSchedule` represents the time of the distributed simulation and enables the self-synchronization between simulation steps.

- `RemoteAgent`, which represents the abstraction of a distributed object `Steppable`. `RemoteAgent` provides also a unique identifier for the agent across the system. The unique identifier is required because agents can migrate from one cell to another.

These three objects are the core of D-MASON. Listing 1 depicts a basic example of a `DistributedState`. The simulation initializes 100 agents for each LP, and sets their positions randomly on a distributed 2D continuous field (`DContinuousGrid2D`).

Listing 1: DSimulation

```
1  public class DSimulation extends DistributedState<Double2D>
2  {
3      public DContinuousGrid2D sim_field;
4      public DSimulation(GeneralParam params, String prefix)
5      { super(params,new DistributedMultiSchedule<Double2D>(),prefix,
           params.getConnectionType());}
6      public void start()
7      {  super.start();
8          try{ sim_field = DContinuousGrid2DFactory.
               createDContinuous2D(10.0/1.5, 200, 200, this, super.AOI,
               TYPE.pos_i, TYPE.pos_j, super.rows, super.columns, MODE,
               "dfield1", topicPrefix, true);
9              init_connection();
10         } catch (DMasonException e) { e.printStackTrace(); }
11         DAgent agent = null;
12         for (int i = 0; i < 100; i++) {
13             agent=new DAgent(this,new Double2D(0,0),
                   this.random.nextInt());
14             agent.setPos(sim_field. getAvailableRandomLocation());
15             sim_field.setObjectLocation(agent, agent.pos);
16             agent.setColor(Color.RED);
17             schedule.scheduleOnce(sim_field);}
18     }
19     public DistributedField2D getField() { return sim_field; }
20     public void addToField(RemotePositionedAgent rm, Double2D loc)
21     { sim_field.setObjectLocation(rm,loc); }
22     public SimState getState() { return this; }
23 }
```

Listing 2 depicts the code for a toy agent. An agent is build by two object: `RemoteDAgent` and its real implementation `DAgent`. `RemoteDAgent` is an abstract class that implements `RemoteUnpositionedAgent` or `RemotePositionedAgent`. These two objects are a subclass of `RemoteAgent` and are, respectively, an instance of an agent that has a position in a geometrical space (e.g., an agent in a continuous 2D space) and an instance of an agent without any positioning (usually used on network fields). This hierarchy is necessary to exploit all MASON features. Indeed, some MASON features, like agents visualization, are obtained by extending a visualization class. On the other hand, in D-MASON the agent class should extend a `RemoteAgent` class, while unfortunately, Java

does not support multiple inheritances. This reasoning justifies the hierarchy described above and ensures the compatibility with all MASON functionalities.

Listing 2: DAgent

```
1  //Abstract based agent class
2  public abstract class RemoteDAgent<E> implements Serializable,
       RemotePositionedAgent<E> {
3     public E pos;      // Location of agents
4     public String id; //ID remote agent
5     public RemoteDAgent() {}
6     public RemoteDAgent(DistributedState<E> state){
7        int i=state.nextId();
8        this.id=state.getType().toString()+"-"+i;
9     }
10   public E getPos() { return pos; }
11   public void setPos(E pos) { this.pos = pos; }
12   public String getId() {return id;}
13    public void setId(String id) {this.id = id;}
14    public boolean equals(Object obj) { //code omitted}
15 }
16 //Agent class
17 public class DAgent  extends RemoteDAgent<Double2D>{
18    private int val;
19    public DAgent(){}  // Required for D-MASON serialization
20    public DAgent(String id, Double2D location, Integer val) {
21       this.id = id; this.pos = location; this.val = val;
22    }
23    public Bag getNeighbors(DistributedState<Double2D> sm){
24       return ((DContinuousGrid2D)sm.getField()).
             getNeighborsExactlyWithinDistance(pos, 10, true);
25   }
26    public void step(SimState state) {
27       Bag b = getNeighbors((DistributedState)state);
28       int max=val;
29       for(Object f: b){
30          DAgent d=(DAgent) f;
31          int dval = d.getVal();
32          if(max < dval) max=dval;
33       } this.val = max;
34    }
35    public int getVal(DistributedMultiSchedule schedule){ return
        val;}
36 }
```

Listing 3 depicts an example of code for executing D-MASON simulation on a local machine. This code considers that an instance of the message broker `Apache ActiveMQ` is running on the local machine. The test initializes and executes 8 LPs (`DSimulation` object), using a uniform partitioning approach.

Listing 3: DTestLocalMachine

```
1  public class DTestLocalMachine {
2      private static int numSteps = 100; //number of step
3      private static int rows = 2; //number of rows
4      private static int columns = 4; //number of columns
5      private static int AOI=10; //AOI
6      private static int CONNECTION_TYPE=ConnectionType.pureActiveMQ;
7      private static String ip="127.0.0.1";
8      private static String port="61616";
9      private static String topicPrefix="toysim";
10     private static int MODE =
           DistributedField2D.UNIFORM_PARTITIONING_MODE;
11     public static void main(String[] args) {
12         (new ActiveMQStarter()).startActivemq();//Start Local
               Embedded ActiveMQ
13         System.setProperty("org.apache.activemq.
               SERIALIZABLE_PACKAGES","*");
14         class worker extends Thread {
15             private DistributedState<?> ds;
16             public worker(DistributedState<?> ds) {this.ds=ds;
                   ds.start();}
17             public void run() {
18                 int i=0;
19                 while(i!=numSteps){
20                     ds.schedule.step(ds); i++;
21                 }System.exit(0);
22             }
23         }ArrayList<worker> myWorker = new ArrayList<worker>();
24         for (int i = 0; i < rows; i++) {
25             for (int j = 0; j < columns; j++) {
26                 GeneralParam genParam = new GeneralParam(null, null,
                       AOI, rows, columns, null, MODE, CONNECTION_TYPE);
27                 genParam.setI(i); genParam.setJ(j);
28                 genParam.setIp(ip); genParam.setPort(port);
29                 ArrayList<EntryParam<String, Object>> simParams=new
                       ArrayList<EntryParam<String,Object>>();
30                 DSimulation sim = new DSimulation(genParam,
                       simParams,topicPrefix);
```

```
31                 worker a = new worker(sim);
32                 myWorker.add(a);
33             }
34         }for (worker w : myWorker) w.start();
35     }
36 }
```

4. D-MASON novel features

D-MASON has been conceived to harness the amount of unused computing power available in common installations like educational labs, that is, a loosely coupled environment with heterogeneous machines. Today, D-MASON is now able to exploit different computing environments ranging from a single multicore machine to a large set of machines available in a cloud environment. Furthermore, D-MASON has been significantly improved in terms of efficiency (*non-uniform partitioning*, *decentralized communication layer*), effectiveness (*memory consistency mechanism*, *network field* and *3D fields*) and usability (*web system management*). In the following, we present these novel features in detail.

4.1. D-MASON *Geometrical Field Partitioning*

D-MASON provides two kind of fields: geometrical fields, where each agent is positioned on a 2-(or 3-) dimensional field, and a network field where agents are placed on the node of a network while relationships are described by network edges. D-MASON provides two geometrical field partitioning approaches:

- *Uniform partitioning,* which divides the simulation field into cells having equal size. For instance, using a 2D field, the partitioning is described by a matrix $[r(ows) \times c(ols)]$ superimposed on the field. The Figure 3 depicts a case study simulation that uses a 2D field, representing the United State of America. The field is decomposed using a $[3 \times 3]$ uniform partitioning, which divides the field into 9 cells. The yellow, red and green zones are the overlapping regions (aka ghost regions) between the cells, which are defined by the AOI range and are exchanged between

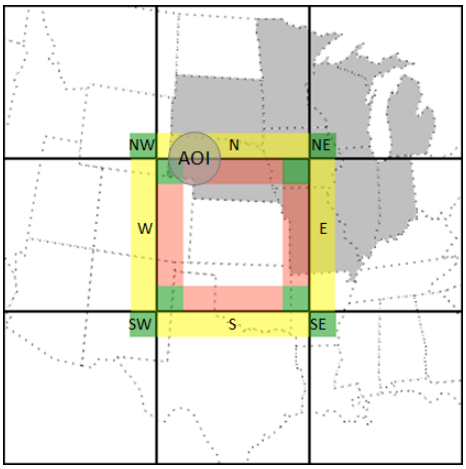


Figure 3: Uniform field partitioning with 9 LPs.

LPs managing adjacent cells for synchronization purposes. Unfortunately, uniform partitioning does not guarantee a balanced distribution of agents. For instance, assuming that the gray zones in Figure 3 represent zones with a high agents density, we have that the uniform partitioning provides a very unbalanced distribution of agents.

- *Non-uniform partitioning,* which exploits the information available on the simulation field (e.g., agents' positions and their computational complexity) to provide a roughly balanced partitioning. The *space partitioning*, in t his case, is described by a variant of the *Quad-tree* data structure [27] where each internal node divides a portion of the field into four balanced portions. The final partitioning is determined by the leaves of the tree, each one representing a cell (see Figure 4). The cell size is inversely proportional to the cell's density or their computational complexity in order to counterbalance the non-uniformity of agents on the field and/or their complexity.

Distributed 2D Fields

The package *Field* defines the logic of agents' distribution. This package provides the distributed versions of several MASON fields as well as several

auxiliary classes. The hierarchy of the package *Field* in D-MASON is based on a Java interface called `DistributedField` that represents an abstraction of a distributed field (cell). All the D-MASONfields implement this interface and expose some common functionalities, such as evaluating whether a given global position belongs to the current local cell or not. The interface also exposes a method `synchro()` that enable the local synchronization among LPs managing adjacent cells (see Figure 2). D-MASON local synchronization is made through the object `UpdateMap` that asynchronously receives messages by LPs managing adjacent cells.

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

Listing 1 (line 15) shows the code for instantiating a `DContinuous2D` field using the factory `DContinuousGrid2DFactory` that instantiates a new distributed field using the given construction parameters. This example uses the method `createDContinuonus2D` which performs a uniform partitioning. On the other hand, the method `createDContinuous2DNonUniform` can be used to perform a non-uniform partitioning.

Distributed 3D Fields

MASON provides also a 3-dimensional field named `Continuous3D`. Generalizing the field partitioning approach, described above, D-MASON provides the field `DContinuous3D` that is the distributed version of the `Continuous3D` field. In this case, the D-MASON simulation space is divided into 3D cells, which are divided in regions, the regions identify the overlapping space between two

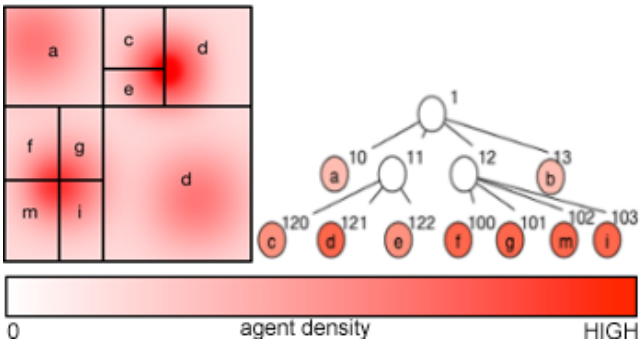


Figure 4: Non-uniform field partitioning with 9 LPs and the associated decomposition tree. The color map describes the agents density on the field.

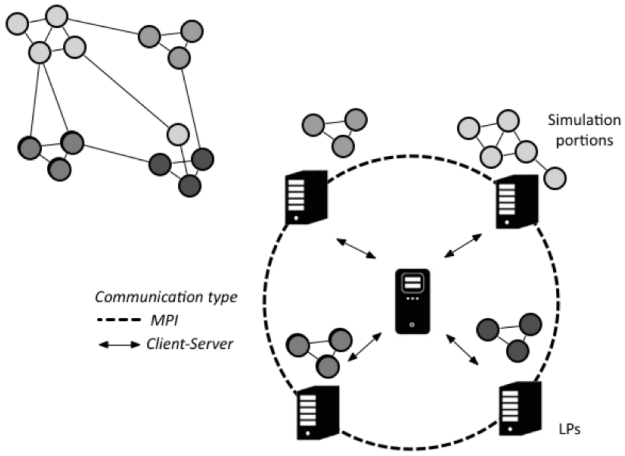


Figure 5: DNetwork Field D-MASON.

neighboring cells (as shown in figure 3 for the 2D case).

`DContinuous3D` divides the 3D space in $[r(ows) \times c(ols) \times d(epht)]$ cells. The simulation workflow is unchanged but the communication phase is much more demanding. The number of messages needed to synchronize the cells is proportional to the number of potential neighbors that in our case goes from 8 (2D fields) to 26 (3D fields).

4.2. The Distributed Network field

The space partitioning approach, implemented on D-MASON, and described in the previous Section, is devoted to decomposing ABMs based on geometric fields, which are motivated by the fact that local interactions are important and matter in everyday life. On the other hand, when interaction can happen on different forms different approach is needed. One way to deal with complex interactions is to model the field as a network [28] which can represent social, geographical or even a semantic space.

Why are Networks important in ABM simulation? Many research phenomena are structured as networks (i.e., sets of nodes joined in pairs by edges representing relations, communications or interactions). Nowadays, networks are everywhere and are the subject of a growing number of research efforts (some examples are the World Wide Web, metabolic, neural, communication, collaboration, and social networks). The study of networked phenomena has experienced a particular surge of interest due to the increasing availability of massive data about the static topology of real networks, as well as the dynamic behavior generated by the interactions among network entities. The studies of real networks have revealed several interesting structural properties, like the small-world phenomena, as well as the power-law degree distribution [29]. On the other hand, understanding the dynamic behavior generated by complex network systems is still an open problem. Indeed, networks are characterized by a dynamic feedback effect, which is hard to predict analytically.

The *distributed network field* enables to partition a network field in an efficient way. With more details, given a network field, the goal is to partition the network into a fixed set of components in such a way that:

1. the components have roughly the same size;
2. both the number of connections and the communication volume between nodes belonging to different components are minimized (see Figure 5).

This problem is well known in the literature as the graph-partitioning problem. It has been extensively studied (see [30] for a comprehensive presentation) and

it is known to be an NP-hard problem. Being a hard problem, exact solutions are found in reasonable time only for small networks. However, the applications of this problem require partitioning much larger networks. For this reason, several heuristics have been proposed in the literature and are discussed, in the context of ABM simulations. D-MASON provides a distributed network field, named *DNetwork* (see Figure 5), based on METIS [31], a graph multilevel k-way partitioning suite, developed in the Karypis lab of University of Minnesota, evaluated for our specific purpose in [32].

4.3. Memory Consistency Mechanism

ABMs are very versatile [15]; they implement the *sense-think-act* paradigm, which, on one hand, offer a real agent-based programming model enabling to model any complex phenomenon but, on the other hand, requires a continuous and strong interdependence among the agents that harm the degree of parallelism [16, 12, 8]. ABMs evolves in discrete steps, that is all the agents update their status at step t, considering the status of all neighbor agents at step $t-1$ (snapshot synchronization). Formally, assuming that an agent v has k neighbors $u_1, u_2, \ldots, u_k$. Let $S_v[t]$ be the state of agent v at the t-th iteration. We have,

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

where f_{upd} computes the state of an agent. Unfortunately, using a sequential computational environment, without the use of suitable strategies (like double buffering), the agents are updated asynchronously, that is

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

where $u_1, u_2, \ldots, u_m$ are the neighbors of v that have already updated their state in the current iteration, while $u_{m+1}, u_{m+2}, \ldots, u_k$ are the neighbors that have not been updated yet. Asynchronous execution exhibits a very strong side effect: the behavior of the model is biased by the order of agents' executions. Moreover, in order to meet the reproducibility feature (see Section 3.1), it is mandatory to update the agents using a deterministic schedule.

D-MASON provides a *memory consistency* mechanism which, in a transparent way, enables to obtain the desired snapshot synchronization. Using the *memory consistency* mechanism, during the t-th simulation step, the read of an agent state returns the state at the end of step $t-1$ when the read is performed from an outside agent while it returns the updated status when the read is performed inside the agent. The implementation of the *memory consistency* mechanism is quite simple. During a simulation step, the first time an agent state is updated the preceding state (that is the state at the end of the preceding step) is saved into an associative data structure, which maps the agent identifier to its state. Then all the remaining write operations operate normally. The read operation performed from the outside (that is using the `getter` method), first checks whether the data structure contains an entry for the corresponding agent. In case of success, the agent state is read from the data structure, otherwise, the agent state has not changed and can be recovered from the agent. Once a simulation step is completed the data structure is cleared so that a unique state is stored. Our approach has a limited memory overhead, compared to the classical double buffering, because each agent state is stored at most twice and a copy is generated only when necessary.

We acknowledge that the mechanism described above can be quite hard to implement, especially for model designers with limited experience in object-oriented and concurrent programming. D-MASON implements the *memory consistency* mechanism at the framework level; the model designers only need to indicate which variables represent the agent state, so that they are treated as described above.

The memory consistency mechanism exploits the Java Method Handles available using Java 8. The package `engine` provides the class `RemoteAgentStateMethodHandler`, a Java object that enables to access to the state of an agent in a consistent way. The same package also provides the `StateVariable` object, which binds each state variable name with the corresponding type. Basically, in order to use the *memory consistency* mechanism of D-MASON, the model's designer has to:

- annotate the variables that define the agent state. In order to do that, each state variable must be declared as part of the agent's state. The agent's state is defined in the agent class as a static list named `StateVariables`;
- define a new static object `RemoteAgentStateMethodHandler`, in order to access the agent state in the step method of the agent. The `RemoteAgentStateMethodHandler` provides two methods `getState(...)` and `setState(...)` to get and set the agents state, ensuring the desired snapshot synchronization.

4.4. Communication Layer

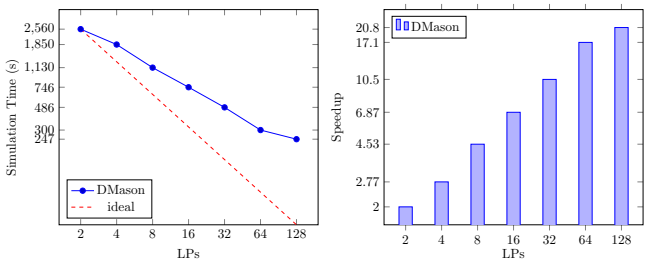
D-MASON provides two kinds of communication specialization:

- *Centralized*, which is used for general purposes architectures (heterogeneous computing or cloud computing), exploiting the Java Message Service standard. The Server side is represented by Apache ActiveMQ [25];
- *Decentralized*, designed mainly for homogeneous computing (such as Extreme-Scale computing), based on the Message Passing Interface (MPI) [33]. The functionalities of the decentralized communication layer have been tested on the OpenMPI [34] implementation.

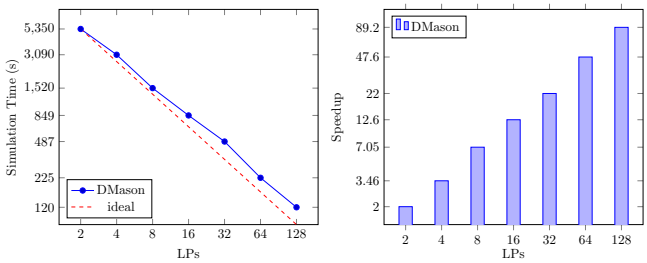
Using MPI, the overall communication is completely decentralized. Moreover, when the system requires some management functionalities, D-MASON communication can be configured using a hybrid approach: the synchronization messages, among LPs, are handled by the MPI infrastructure (in order to achieve scalability) while the management messages, being asynchronous, operate through the ActiveMQ Server, with no harm to the overall system efficiency.

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

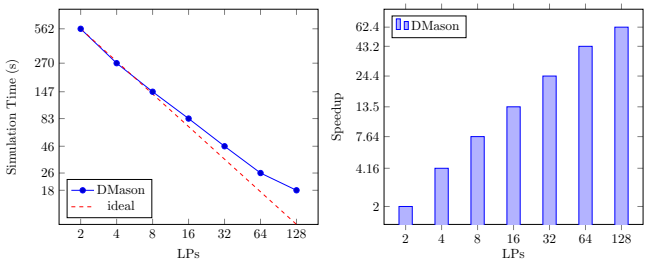
The following communication strategies are currently available:



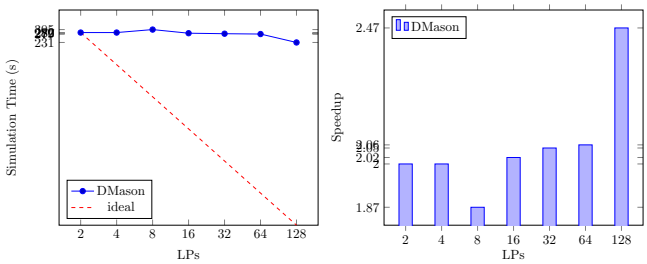
(a) Circle



(b) Flockers



(c) Game-of-Life



(d) Ants

Figure 6: Strong scalability of different D-MASON 2D simulations for 10^6 agents.

1. `pureActiveMQ:` uses the centralized communication strategy (*Apache ActiveMQ* as message broker) for both the management and the synchronization messages;

2. `pureMPI,` uses the decentralized communication strategy and can be used only when the management services (like the centralized visualization) are disabled. Three different centralized communication mechanisms are available [35, 36]:

   - `pureMPIBcast,` exploits an MPI broadcasting feature (`MPI Bcast`);
   - `pureMPIGather,` exploits an MPI gathering feature (`MPI Gather`). Using this mechanism an LP is able to get, in a single step, all the information needed from its neighborhood. This mechanism allows to decrease the number of communication rounds required for synchronization but increases the messages size.
   - `pureMPIParallel.` This approach is based on a randomized graph coloring algorithm, which aims to maximize the degree of parallelism during the communications between different LPs, in order to reduce the number of communication rounds required for synchronization. This mechanism generally provides the best performance and is highly recommended for simulations having a large number of LPs.

3. `hybridActiveMQMPI,` uses the centralized communication strategy for the asynchronous communication between LPs and the system management, while it uses the centralized communication strategy for simulation updates between LPs. Also, in this case, it is possible to choose the desired MPI communication mechanism.

4.5. System Management

The main purpose of the System management is to provide a better user experience to scientist that are using D-MASON for their experiments. The System management is used for the discovery of workers (the machines that

provides the computational power), the bootstrap and the management of simulations in a simple and efficient way.

In order to meet the requirements above, a fully decoupled system management service available via web services have been designed and deployed. D-MASON system management is based on Jetty [37] web server, the open web application containers available for Java. In order to develop an efficient, pleasant and engaging interface, the web system management design was based on Google material design [38], the guidelines provided by Google for the development of design interfaces.

The web system management provides four main views, selectable by a control panel:

1. *Monitoring* enables the user to discover the workers available in the local networks and, for each available worker, provides information about the available resources (number of CPUs, CPUs current load, available memory etc.). Using such information, the user is able to choose appropriately the workers to be engaged in future simulations. The system enables the user to set up and run a simulation; a library of preloaded simulation is available but, at the same time, it is possible to upload a novel simulation as a jar file. Once a simulation has been chosen, the user sets its parameters and starts it

2. *Simulations* shows the list of all the simulations running on the system and enables the user to manage and monitor the running simulations. While a simulation is running the user can start, pause or stop the execution using the *Simulation Controller*. Moreover, in order to monitor the evolution of a simulation, a logging mechanism has been implemented. All the log files are available at run-time on the Simulation Info panel.

3. *History* provides information about the performed simulations, their parameters, and the generated log files. All such files are also available for download for offline studies or comparison.

4. *Settings* enables the user to change system configurations, for instance, the IP address and PORT number for the JMS server.

4.6. D-MASON *on the cloud*

D-MASON on the cloud is an extension designed for scalable distributed Agent-Based simulations, realizing a SIMulation-as-a-Service (SIMaaS) environment. D-MASON on the cloud has been designed and tested on top of Amazon Web Services (AWS), a scalable and highly reliable cloud computing infrastructure.

AWS provides a wide range of services on the cloud. D-MASON on the cloud exploits Amazon Elastic Compute Cloud (Amazon EC2) that provides resizable computing capacity on the cloud. Amazon EC2 is an instance of the Infrastructure as a Service (IaaS) model, where the Amazon infrastructure is seen as a completely virtual environment, which enables to execute different instances of virtual machines. Amazon EC2 provides a wide portfolio of instance types [39], designed to be adopted for different use cases. Instance types vary by CPU performance, the speed and the size of memories and storage as well as the network bandwidth. AWS enables the user to deploy the operating system, application software, and configuration settings into an Amazon Machine Image (AMI). Advanced users may also create their own AMIs and publish them on the Marketplace Web Service (MWS).

D-MASON provides two different approaches for SIMaaS: a static approach [40], in which a cluster of machines is created and automatically configured on an IaaS, and an on-need approach, where the user can dynamically allocate and release D-MASON workers on an IaaS.

The *Static approach* is defined by three levels. The first level is the hardware infrastructure that is given by the IaaS Amazon EC2. The second level is a D-MASON AMI, that provides the whole D-MASON software stack, ready to run a worker. The last level is a D-MASON StarCluster plug-in, which exploits the functionality provided by the cluster-computing toolkit StarCluster [41]. This tool enables the user to configure and manage EC2 instances clusters.

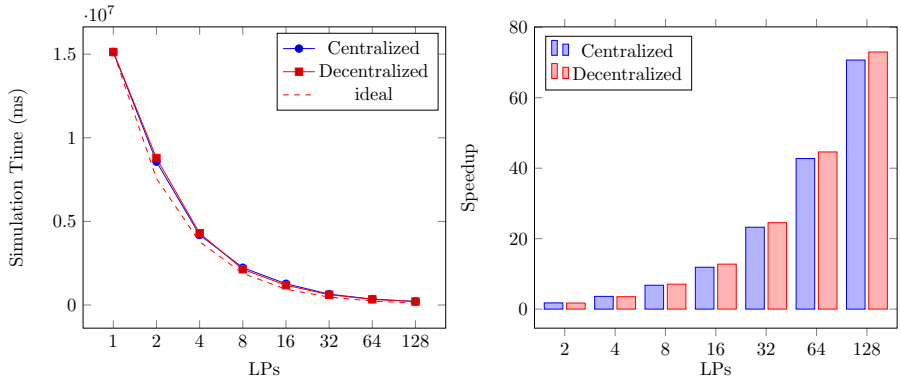


Figure 7: Strong Scalability of Woims 3D simulation.

One instance runs the D-MASON Master application, the web system management server and the JMS message broker (ActiveMQ) (for centralized or hybrid communication). The remaining instances run the D-MASON Worker applications, who carry out the simulation.

The *On-need approach* enables the user to dynamically manage (allocate and release) workers on AWS using the monitoring view of the web system management. This functionality exploits the AWS Java SDK and requires the AWS credentials [1]. Once the user credentials have been configured, in the settings view of the web system management, the user can select the number and the type of instances to be allocated as workers for each simulation.

5. Evaluations

This section presents the results of several experiments devoted to evaluating the scalability of D-MASON on different ABMs, exploiting the space partitioning, and overcoming the limits of sequential computing by simulating millions of agents. All the experiments have been carried out on the same infrastructure, a computing cluster of 15 nodes, each equipped as follows: 2 × CPUs Intel(R) Xeon(R) CPU E5-2430 v2 @ 2.50 GHz (#core 12, #threads 24);

[1] AWS credentials – `http://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/credentials.html`

32 GB RAM; Intel Corporation I350 Gigabit Network adapters; Ubuntu 14.04.3 LTS Operating System; Oracle 1.8 Java Virtual Machine.

We present our results in terms of *Weak* and *Strong* scalability. Weak scalability benchmarks explore the ability of the framework to scale using a fixed amount of computation for each LP, while strong scalability benchmarks explore the ability of the framework to scale using a fixed amount of computation but varying the number of LPs.

5.1. Scalability on different simulation models

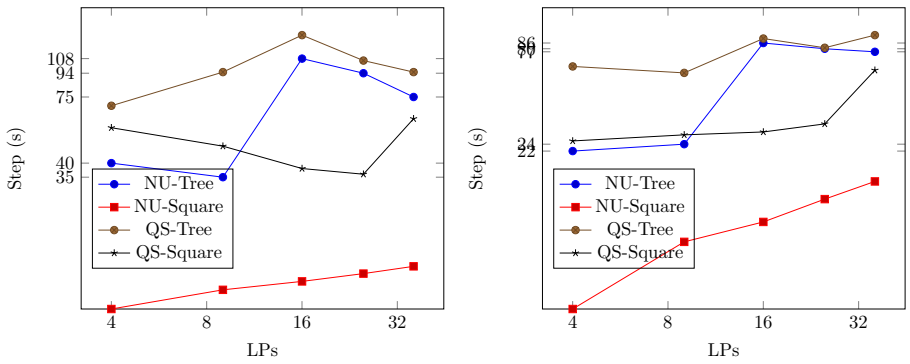
We evaluated the performances of D-MASON on four well known ABM models, developed on $2D$ environments:

- *Circle* represents standardized benchmark for assessing the performance of fixed-radius near neighbor lookups. It is part of the OpenAB initiative [42], and reference implementations are available in FlameGPU [10].

- *Flockers (Boids)* introduced by Reynolds [43] defines a steering behavior for autonomous agents, which simulates the flocking behavior of birds. The agent motion is derived from three components: separation, to avoid local birds; alignment, to match the average direction of local birds; cohesion, to move toward the average position (center of mass) of local birds.

- itGame-of-life is a well-known cellular model presented in [26]. Each agent/cell may assume one over two states: live and dead. Cells states are updated simultaneously and in discrete time; the cells change their state accordingly to the states of their neighboring cells.

- *Ants Foraging* model, described in [44], simulates ants foraging behaviors dynamics. When ants discover a food source, they establish a trail of pheromones between the nest and the food source. The model uses two pheromones, which set up gradients and evaporate after some simulation steps, to guide ants to the nest and to the food source, respectively.



(a) Weak Scalability

(b) Strong Scalability

Figure 8: Weak and Strong Scalability of geometrical field partitioning strategies.

Figure 6 depicts D-MASON results in simulating each model using 10^6 agents, for 1000 simulation steps. For each model, we present, on the left chart, the simulation times in seconds, varying the number of LPs from 2 to 128 (X-axis log scale), while, on the right chart, the obtained speedup is provided. As shown, in the figure, Game-of-life and Flockers model simulations provide very good results: the speedup is ≈ 62 and ≈ 89 respectively. The Circle model simulation provides a smaller speedup (≈ 20), due to the unbalancing of the agents over the field. Finally, the Ant Foraging simulation model, provides the worst speedup (≈ 2), because this model is characterized by a high degree of unbalancing. Moreover, the distribution of ants on the field continuously varies during the simulation, precluding the use of any static balancing strategy.

5.2. Scalability on 3D simulations

We also investigate the scalability of D-MASON in simulating models that exploit 3D environments. We used the Woims ABM model (available in the MASON examples). Woims is inspired by Reynolds bird-flocking model, but in this case, the agents (worms) moves in a 3D environments, avoiding obstacles and trying to avoid the head of others worms. We have developed a D-MASON simulation, by adapting the original MASON code in D-MASON.

We simulated 10^6 agents, for 100 simulation steps, on a 3D simulation field of

size $10^3 \times 10^3 \times 10^3$ using both the centralized and decentralized communication. Figure 7 depicts, on the left side, the simulation times expressed in milliseconds, and on the right side the obtained speedup, varying the number of LPs from 1 to 128 (X-axis log scale) using the two communication strategies (series). As shown in the figure the larger speedup (70.7 for centralized communication and 72.9 for decentralized communication) are obtained in the configuration that exploits 128 LPs (8 rows $\times$ 4 cols $\times$ 4 layers).

5.3. Evaluating the Geometrical Field Partitioning

This analysis aims to evaluate the scalability of the different geometrical field partitioning strategies adopted in D-MASON and described in Section 4.1. We tested the 2D *Flockers* model where the agent speed is limited to a fixed range, in order to keep the initial distribution of the agents on the field roughly constant.

The experiments compare two field partitioning strategies:

- *Uniform partitioning* (henceforth *Square*), which partition the field in k cells (number of LPs), using a $\sqrt{k} \times \sqrt{k}$ matrix (all the cells have the same dimensions $\left(\left(w/\sqrt{k}\right) \times \left(h/\sqrt{k}\right)\right)$, where w and h are the dimensions of the field).
- *Non-uniform partitioning* (henceforth *Tree*), the Non-uniform partitioning described in Section 4.1.

These strategies have been tested on two distributions of agents:

- *QU* (Quasi-Uniform distribution), the agents are distributed in 8 groups, 3 groups containing each one the 25% of the agents and 5 containing each one the 5% of the agents.
- *NU* Non-Uniform distribution), the agents are distributed in a dense giant group on very small field area.

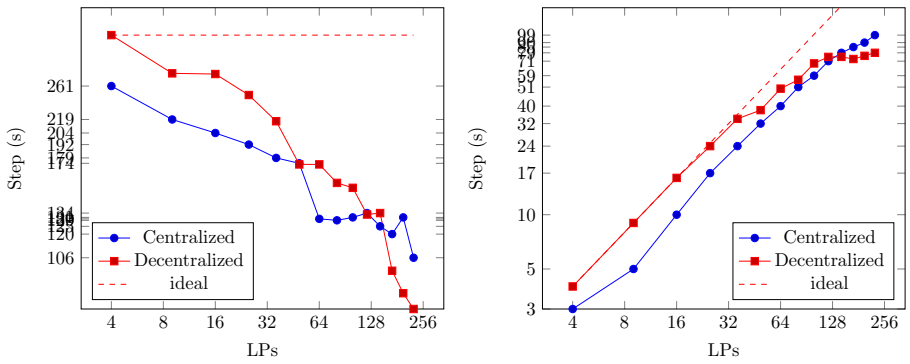
In the following, the performance results in terms of weak and strong scalability are described.

Weak scalability

The weak scalability test evaluates D-MASON scalability assigning a constant workload for each logical processors. Hence, in this test the number of agents A is proportional to the number of logical processors k (i.e., $A = 28000 \times k$) while the dimensions of the field are set in order to keep the agents density constant (i.e., $density = \left(\frac{w \times h}{A}\right) \approx 100$, where w and h denote the width and the height of the field). Twenty configurations varying the value of $k \in \{4, 9, 16, 25, 36\}$, the partitioning strategy (*Square* and *Tree*) and the agents' distribution (QU and NU) were tested. Figure 8 depicts, on the left side, the weak scalability results as the the number of simulation steps performed in a time span of 120 seconds (Y-axis), for each value of $k \in \{4, 9, 16, 25, 36\}$ (X-axis log scale) and for each *partitioning strategy-agents' distribution* (series). As shown in the picture, the *Tree* strategy always gives better results for both the agents' distributions. We notice that the performance trends are affected by the granularity of the decomposition, which impacts on the communications overhead. Indeed, in the *Square* strategy, the amount of communication overhead is proportional to k (8 communication channels for each cell) while the *Tree* strategy requires more channels. Hence, with small values of k, the gap is sensible, but increasing k, the difference tends to decrease as the communication overhead, for the *Tree* strategy, increases, especially using a centralized communication approach.

Strong scalability

The strong scalability test evaluates D-MASON scalability for problems of fixed size but varying the number of logical processors involved. Hence, in this test, the amount of computation consists of 10^6 agents moving on a 2D field of size $10^4 \times 10^4$. Figure 8 depicts, on the right side, the strong scalability results as the number of simulation steps performed in a time span of 120 seconds (Y-axis), for each value of $k \in \{4, 9, 16, 25, 36\}$ (X-axis log scale) and for each *partitioning strategy-agents' distribution* (series). Like the weak scalability results, the *Tree* strategy always gives better results for both the agents' distributions. The



(a) Weak Scalability

(b) Strong Scalability

Figure 9: Weak and Strong scaling of Flockers simulation for 20 Million of agents.

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

5.4. Beyond the Limits of sequential computation

The rationale behind this set of tests is to evaluate the performance of D-MASON on very large simulations (simulations that cannot be executed on a sequential machine). All these tests have been executed using the two communication schemes (centralized and decentralized) and running the *Flockers* simulation.

Weak scalability

In the weak scalability test, the amount of computation for each LP consists of around $90,000$ agents. Several tests were performed varying the number of LPs from 4 ($360,000$ Agents) up to 225 (20M Agents). Figure 9 (left) presents the results: the $X-$axis indicates the number of LPs (log scale), while the $Y-$axis indicates the number of steps performed within a time span of 15 minutes. The overall performance degrades gracefully and the centralized approach

seems to scale better than the decentralized one although the decentralized approach performs better when the number of LPs is small (< 64).

Strong scalability

In the strong scalability test, the overall amount of computation is fixed ($20M$ agents). Several tests were performed varying the number of LPs (from 4 to 225). Figure 9 (right) presents the results: the $X-$axis indicates the number of LPs (log scale), while the $Y-$axis indicates the number of steps performed within a time span of 15 minutes. The speedup provided is always better than half of the ideal speedup. Like for the previous test, the decentralized communication approach better performs when the number of LPs is small while the centralized communication approach seems to scale better.

6. Conclusion

This paper reports on a currently undergoing project, D-MASON, a scalable distributed multi-agent simulation environment that has been deployed with the purpose of speeding up the performance of MASON, a very well known and widespread framework for ABMs. We have presented, several novel features that have been developed in the framework:

- A novel geometric work partitioning approach (Non-uniform), which exploits the information available on the simulation field (e.g., agents' positions and their computational complexity) to provide a roughly balanced partitioning even for simulations characterized by an uneven distribution of agents.
- A `DContinuous3D` field, that is a distributed version of MASON `Continuous3D` field.
- A distributed network field, named `DNetwork`, which enable to perform distributed simulations, where agents' relations and interactions are described by a network.

- A fully decentralized communication strategy, based on the MPI standard.
- A *Memory Consistency Mechanism*, which, at system level, guarantees the perfect synchronization among simulation steps, allowing the designer to devote himself exclusively to the behavior of the agents.

Furthermore D-MASON provides integration for cloud computing environments. Two different approaches have been investigated: a static approach that enables to exploits StarCluster to initiate a D-MASON cluster on Amazon Web Services and an on-need approach, which exploits the novel web system management, enabling the user to allocate and release cloud instances according to the simulation computational needs.

Based on the same architecture, further work is already planned to improve load balancing techniques and support other cloud computing infrastructures. A Unit Test environment is also under development in order to improve the implementation and integration cycle. D-MASON is released under a Free and Open Software license and is available at [45].

References

[1] G. Cordasco, C. Spagnuolo, V. Scarano, Toward the new version of D-MASON: Efficiency, Effectiveness and Correctness in Parallel and Distributed Agent- based Simulations, in: 1st IEEE Work. on Parallel and Distr. Processing for Comput. Social Systems (ParSocial), 2016.

[2] P. I. T. A. Committee, Computational Science: Ensuring America's Competitiveness. (2005).

[3] A. López-Paredes, B. Edmonds, F. Klugl, Editorial of the Special Issue: Agent Based Simulation of Complex Social Systems, Simulation: Trans. of the Society for Modeling and Simulation International.

[4] J. Epstein, S. Levin, S. Strogatz, Generative Social Science: Studies in Agent-Based Computational Modeling, Princ. Univer. Press, 2007.

[5] T. E. A. of Change (2010). [link].
URL http://www.economist.com/node/16636121

[6] C. Cioffi-Revilla, Invariance and universality in social agent-based simulations, Proceedings of the National Academy of Sciences 99 (suppl 3) (2002) 7314–7316. doi:10.1073/pnas.082081499.

[7] H. Simon, The sciences of the artificial, MIT Press, 1996.

[8] N. Collier, M. North, Parallel agent-based simulation with Repast for High Performance Computing, SIMULATION: Transactions of the Society for Modeling and Simulation International.

[9] G. Cordasco, R. D. Chiara, A. Mancuso, D. Mazzeo, V. Scarano, C. Spagnuolo, Bringing together efficiency and effectiveness in distributed simulations: the experience with D-MASON., SIMULATION: Trans. of The Society for Modeling and Simulation International.

[10] M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, C. Greenough, Flame: Simulating large populations of agents on parallel hardware architectures, in: Inter. Conf. on Autonomous Agents and Multiagent Systems: Vol. 1, AAMAS '10, 2010, pp. 1633–1636.

[11] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599 – 616. doi:http://dx.doi.org/10.1016/j.future.2008.12.001.

[12] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, MASON: A new multi-agent simulation toolkit, in: Proceedings of the 2004 SwarmFest Workshop, Ann Arbor (Michigan), USA., 2004.

[13] S. Tisue, U. Wilensky, NetLogo: A simple environment for modeling complexity, in: International Conference on Complex Systems, 2004.

[14] B. Logan, G. Theodoropoulos, The Distributed Simulation of Multi-Agent Systems, in: Proceedings of the IEEE, 2000.

[15] M. Hybinette, E. Kraemer, Y. Xiong, G. Matthews, J. Ahmed, SASSY: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure, in: Winter Sim. Conf., 2006.

[16] N. Minar, R. Burkhart, C. Langton, et al., The swarm simulation system: A toolkit for building multi-agent simulations, Technical report, Swarm Development Group.

[17] N. Collier, M. North, A Platform for Large-scale Agent-based Modeling, in: W. Dubitzky, K. Kurowski, and B. Schott, eds., Large-Scale Computing Tech. for Complex System Sim., Wiley, 2011.

[18] D. Mengistu, P. Troger, L. Lundberg, P. Davidsson, Scalability in Distributed Multi-Agent Based Simulations: The JADE Case, in: Proc. Second Int. Conf. Future Generation Communication and Networking Symposia FGCNS '08, 2008.

[19] D. Pawlaszczyk, S. Strassburger, Scalability in distributed simulations of agent-based models, in: Proc. Winter Simulation Conf., 2009.

[20] A. Rousset, B. Herrmann, C. Lang, L. Philippe, A Survey on Parallel and Distributed Multi-Agent Systems, in: Euro-Par 2014: Parallel Processing Workshops, 2014.

[21] M. Berryman, Review of Software Platforms for Agent Based Models, Defence Science and Technology Organisation, DSTO-GD-0532.

[22] K. Hwang, Z. Xu, Scalable parallel computing: technology, architecture, programming, WCB/McGraw-Hill, 1998.

[23] B. Cosenza, G. Cordasco, R. D. Chiara, V. Scarano, Distributed Load Balancing for Parallel Agent-Based Simulations, in: Parallel, Distributed and Network-Based Processing (PDP), 2011.

[24] B. Zhou, S. Zhou, Parallel simulation of group behaviors, in: WSC '04: Proceedings of the 36th conference on Winter simulation, 2004.

[25] ActiveMQ (2017). [link].
URL http://activemq.apache.org

[26] A. Adamatzky, Game of Life Cellular Automata, 1st Edition, Springer Publishing Company, Incorporated, 2010.

[27] M. Overmars, J. Leeuwen, Dynamic Multi-dimensional Data Structures Based on Quad- and K–d Trees, Acta Inf.

[28] S. J. Alam, A. Geller, Networks in Agent-Based Social Simulation, Springer Netherlands, 2012, pp. 199–216.

[29] D. Easley, J. Kleinberg, Networks, Crowds, and Markets: Reasoning About a Highly Connected World, Cambridge University Press, 2010.

[30] D. Bader, H. Meyerhenke, P. Sanders, D. Wagner, Graph partitioning and graph clustering, in: 10th DIMACS Implementation Challenge Workshop Proceedings, 2013.

[31] G. Karypis, V. Kumar, Multilevel k-way Partitioning Scheme for Irregular Graphs, Journal of Parallel and Distributed Computing.

[32] A. Antelmi, G. Cordasco, C. Spagnuolo, L. Vicidomini, On Evaluating Graph Partitioning Algorithms for Distributed Agent Based Models on Networks, in: Proc. of the 3rd Work. on Par. and Distr. Agent-Based Simulations (PADABS). Euro-Par 2015, 2015.

[33] MPI Forum (2017). [link].
URL http://www.mpi-forum.org

[34] Open MPI: Open Source High Performance Computing (2016). [link].
URL http://www.open-mpi.org/

[35] G. Cordasco, F. Milone, C. Spagnuolo, L. Vicidomini, Exploiting D-Mason on Parallel Platforms: A Novel Communication Strategy, in: Proc. of the 2nd Work. on Par. and Distr. Agent-Based Simulations (PADABS). Euro-Par 2014, 2014.

[36] G. Cordasco, A. Mancuso, F. Milone, C. Spagnuolo, Communication Strategies in Distributed Agent-Based Simulations: The Experience with D-Mason, in: Proc. of the 1st Work. on Par. and Distr. Agent-Based Simulations (PADABS). Euro-Par 2013, 2013.

[37] Jetty (2013). [link].
URL http://www.eclipse.org/jetty/

[38] Google Material Design (2016). [link].
URL https://www.google.com/design/spec/material-design

[39] Amazon EC2 (2016). [link].
URL https://aws.amazon.com/ec2

[40] M. Carillo, G. Cordasco, F. Serrapica, C. Spagnuolo, P. Szufel, L. Vicidomini, D-Mason on the Cloud: an Experience with Amazon Web Services, in: Proc. of the 4th Work. on Par. and Distr. Agent-Based Simulations (PADABS). Euro-Par 2016, 2016.

[41] StarCluster (2017). [link].
URL http://star.mit.edu

[42] OpenAB, Open agent benchmark initiative for parallel and distributed benchmarking, http://www.openab.org/ (2017).

[43] C. W. Reynolds, Flocks, herds and schools: A distributed behavioral model, SIGGRAPH Comput. Graph. 21 (4) (1987) 25–34. doi:10.1145/37402.37406.

[44] U. Wilensky, W. Rand, An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo, The MIT Press, 2015.

[45] D-MASON Official GitHub Repository (2017). [link].
URL https://github.com/isislab-unisa/dmason