On Evaluating Rust as a Programming Language for the Future of Massive Agent-Based Simulations

Alessia Antelmi[1(✉)], Gennaro Cordasco[2(✉)], Matteo D'Auria[1(✉)], Daniele De Vinco[1(✉)], Alberto Negro[1(✉)], and Carmine Spagnuolo[1(✉)]

[1] ISISLab, Dipartimento di Informatica, Università Degli Studi di Salerno, Fisciano, Italy
{aantelmi,matdauria,alberto,cspagnuolo}@unisa.it, d.devinco@studenti.unisa.it

[2] Dipartimento di Psicologia, Università degli Studi della Campania "Luigi Vanvitelli", Caserta, Italy
gennaro.cordasco@unicampania.it

Abstract. The analysis of real systems and the development of predictive models to describe the evolution of real phenomena are challenging tasks that can improve the design of methodologies in many research fields. In this context, Agent-Based Model (ABM) can be seen as an innovative tool for modelling real-world complex simulations. This paper presents Rust-AB, an open-source library for developing ABM simulation on sequential and/or parallel computing platforms, exploiting Rust as programming language. The Rust-AB architecture as well as an investigation on the ability of Rust to develop ABM simulations are discussed. An ABM simulation written in Rust-AB, and a performance comparison against the well-adopted Java ABM toolkit MASON is also presented.

Keywords: Rust language · Agent-Based Model · Simulation · Framework

1 Introduction

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 [18]. The analysis of real systems has revealed several interesting emergent behaviours both in terms of structural features [10] and dynamic behaviours [17]. However, a full understanding of the dynamic behaviour generated by complex systems is extremely hard and requires innovative study methodologies. Recently, computational scientists have proposed the analysis of these phenomena through the exploitation of simulations based on Agent-Based Model (ABM). ABM simulations denote a class of models that, simulating the behaviour of multiple agents, aim to emulate and/or predict complex phenomena. An ABM consists of

G. Tan et al. (Eds.): AsiaSim 2019, CCIS 1094, pp. 15–28, 2019.
https://doi.org/10.1007/978-981-15-1078-6_2

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.

Motivation. 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 [8] - 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 [11]. In such cases, the problem consists in ensuring good performance and a high-level of effectiveness in simulation modelling. 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) [6]. The computation of a global parameter represents a bottleneck for distributed simulations, which jeopardise the performance due to the communication overhead. In such cases, the use of a parallel and/or sequential simulation framework provides better performances [7]. 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 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 recognised as the core advantage of using Rust over languages as C++ and Java as it allows the user to write correct code, particularly in the presence of concurrency and parallelism. We will describe Rust key concepts in Sect. 3.

Outline and Paper Contributions. We present an overview of the current state-of-art of the ABM frameworks/libraries for developing simulation (see Sect. 2). In Sect. 3, we analyse Rust features with a focus on its peculiarity for writing ABM simulation. The main contribution of this work is the description of a novel library, **Rust-AB**, for writing ABM simulations. The Alpha 1.0 version of the library is released on a public GitHub repository [3] and is presented in Sect. 4. In Sect. 5, we present a case study: we developed an ABM simulation using Rust-AB, and we present a performance comparison with the same model on MASON.

2 Related Work

This Section outlines some 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 by Abar et al. [1].

ABM software tools can be easily classified into two categories according to the underlying architecture: software for sequential computing architectures and software for distributed computing architectures. Table 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. First six rows of Table 1 summarise as many frameworks suitable for sequential computing simulations and that are fully described by Abar et al. [1]. Several frameworks are designed to develop large-scale simulations and provide good performance exploiting parallel/distributed computing architectures. The bottom part of Table 1 presents three frameworks for developing ABM simulations in distributed and parallel computing architecture.

This work focuses on improving the simulation performance, when the computation of global data is required, on sequential or parallel architectures, avoiding the complexity and the limitations introduced by distributed computing. We will further focus on the effectiveness and the expressiveness of the produced software in developing ABM.

Table 1. ABM frameworks/software comparison.

ABM tool	Source Language	Applications language	Computing platform	Application domain	License
SWARM [12]	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 [14]	Java/YoYo	StarLogo scripting	Desktop computer	Simulation in social and natural sciences, education	Closed source, Clearthought Software License version 1.0, Free
NetLogo [20]	Scala	NetLogo language	Desktop computer	2D/3D simulation in social and natural science, teaching/research	Open source, GPL, Free
REPAST [13]	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 [19]	Java	Java	Desktop computer, Workstation	General multi-purpose 2D/3D simulation	Open source, Academic Free License version 3.0
FLAME [9]	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 [4]	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 [5]	Java	Java	Desktop computer, Workstation, Clusters, Cloud architectures	General multi-purpose 2D/3D simulation	Open souce, Apache License version 2.0
FLAME-GPU [16]	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

3 Rust Background

Rust is a multi-paradigm system programming language, originally designed 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 behaviour 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 behaviours in an abstract way, in which a type's behaviour is defined by the methods we can call on that type.

4 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: (i) the simulation agents, specified by the Java interface Steppable; (ii) the simulation scheduler, defined by the Scheduler object; (iii) 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 an easier adoption of a new language as Rust. The Alpha 1.0 version of the Rust-AB simulation engine library is fully developed and released under MIT license on a public GitHub repository [3]. Section 4.1 describes Rust-AB architectural concepts and functionalities.

4.1 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 to 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. In the Alpha 1.0 version of Rust-AB, 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 step 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 neighbourhood interactions. Every Rust struct that implements this trait can be placed in a Rust-AB field. A position in a 2-D space is modelled 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 easily model interactions with any kind of simulation environment. 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. `Simulation State` is the state of a Rust-AB simulation. A Rust-AB simulation is composed by 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. To access this struct, the programmer has to declare the struct itself as a static reference and initialise it at running time. Moreover, to ensure the Rust memory model accesses to this struct must be done using a lock (or mutex), which secure safe memory access in the agent step function. This procedure is better described in Sect. 5.

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 Rust-AB does not support multiple definitions of an agent. Nevertheless, it is still possible to implement a multi-agents model by defining different behaviours 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).

5 A Case Study: The *Boids* Simulation

To analyse 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 have been compared against the MASON toolkit running the same ABM. We developed the Boids model [15] by Raynolds (1986), which is a steering behaviour ABM for autonomous agents simulating the flocking behaviour of birds. The agent behaviour is derived by a linear combination of three independent rules: (1) *Separation*: steer in order to avoid crowding local flock-mates; (2) *Alignment*: steer towards the average heading of local flock-mates; (3) *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 initialisation are provided.

5.1 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 Sect. 4.1, 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 neighbouring 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 behaviour model can be implemented by storing the position of the agent in the previous and current simulation steps. The agent position can be modelled using a `Real2D` Rust-AB struct. To easily develop the trait *Hash*, an unique identifier is stored in the agent. Listing 1.1 shows the Rust-AB agent struct definition.

Rust Code 1.1: Rust-AB Agent Struct.

```rust
#[derive(Clone, Copy)]
pub struct Bird{
    pub id: u128,
    pub pos: Real2D,
    pub last_d: Real2D,
}
```

The agent logic is defined in the `step` function. We designed the agent logic using three sub-functions defined in the agent implementation. Listing 1.2 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 9–12 show the implementation of the `Location2D` trait, which enables to place the agent in the `Field2D` environment. Lines 13–20 define the implementation of the traits `Hash` and `Eq`. Lines 21–39 implement the agent `step` function describing the agent logic, which simulates the steering behaviour of the model. The agent computes its neighbourhood (line 23) and, using the sub-functions, evaluates its new position. The computed position is then used to update the status of the environment (line 37), exploiting a lock mechanism.

Rust Code 1.2: Rust-AB Agent Implementation.

```rust
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 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,
    { state.write_u128(self.id); state.finish();}
}
impl Eq for Bird {}
impl PartialEq for Bird {
    fn eq(&self, other: &Bird) -> bool {self.id == other.id}
}
impl Agent for Bird {
    fn step(&mut self) {
        let vec = GLOBAL_STATE.lock().unwrap().field1.get_neighbors_within_distance(self.pos,10.0);
        let avoid = self.avoidance(&vec);
        let cohe  = self.cohesion(&vec);
        let rand  = self.randomness();
        let cons  = self.consistency(&vec);
        let mom   = self.last_d;
        let mut dx = COHESION*cohe.x + AVOIDANCE*avoid.x + CONSISTENCY*cons.x + RANDOMNESS*rand.x + MOMENTUM*mom.x;
        let mut dy = COHESION*cohe.y + AVOIDANCE*avoid.y + CONSISTENCY*cons.y + RANDOMNESS*rand.y + MOMENTUM*mom.y;
        let dis = (dx*dx + dy*dy).sqrt();
        if dis > 0.0 { dx = dx/dis*JUMP; dy = dy/dis*JUMP;}
        let _lastd = Real2D {x: dx, y:dy};
        let loc_x = toroidal_transform(self.pos.x + dx, WIDTH);
        let loc_y = toroidal_transform(self.pos.y + dy, HEIGHT);
        self.pos = Real2D{x: loc_x, y: loc_y};
        GLOBAL_STATE.lock().unwrap().field1.set_object_location(*self, Real2D{x: loc_x, y: loc_y});
    }
}
```

5.2 Model Definition

We define the Boids simulation state by declaring a new struct `State`. Listing 1.3 shows the code of the *State* struct (lines 1–6). 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. As described in Sect. 3, the memory model of Rust does not allow data sharing across several function invocations. Thus, to access the simulation state inside the agent `step` function, the `State` instance has to be a global variable, accessed though a lock (or a mutex) to safely read it. The `State` struct has to be initialised at running time using the macro `lazy_static!` (lines 8–10).

Rust Code 1.3: Rust-AB Simulation State.

```rust
pub struct State{
    pub field1: Field2D<Bird>,
}
impl State {
    pub fn new(w: f64, h: f64, d: f64, t: bool) -> State { State {field1:
↪   Field2D::new(w, h, d, t),}}
}
//Global variables definition
lazy_static! {
    static  ref GLOBAL_STATE: Mutex<State> = Mutex::new(State::new(WIDTH, HEIGHT,
↪   DISCRETIZATION, TOROIDAL));
}
```

The main simulation function is shown in Listing 1.4. At line 2, a new Rust-AB `Schedule` is defined, while from line 3 to 11 a given number of agents are randomly initialised, placed in the `Field2D` (line 9), and scheduled using the `schedule _repeating` method (line 10). At line 12 the schedule step is called for a given number of times.

Rust Code 1.4: Rust-AB Main Simulation Function

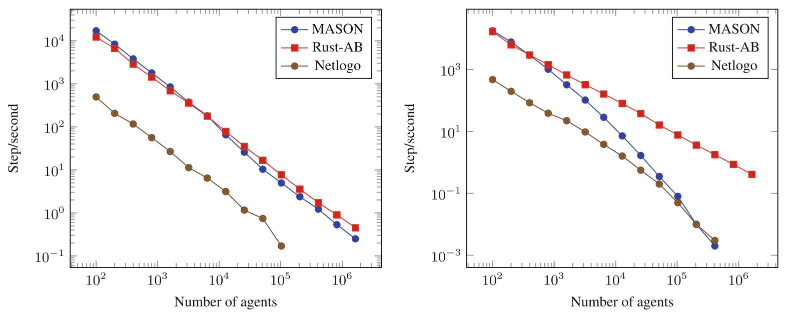
```rust
1   fn main() {
2       let mut schedule: Schedule<Bird> = Schedule::new();
3       let mut rng = rand::thread_rng();
4       for bird_id in 0..NUM_AGENT{
5           let r1: f64 = rng.gen();
6           let r2: f64 = rng.gen();
7           let last_d = Real2D {x: 0.0, y: 0.0};
8           let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

5.3 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 $\times$ 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 number 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.

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 Fig. 1(a). 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 min 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.

Constant Field Size. We evaluated the simulation engine's ability to simulate an increasing number of agents laying 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 neighbourhood and the new position of each agent. Figure 1 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 behaviour may be due to Rust's ability to efficiently manage a high computational workload, mainly thanks to its memory system. Further analysis are needed to asses 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.



(a) Constant Agents Density (b) Constant Field Size

Fig. 1. Rust-AB performance comparison.

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 2 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 become small enough to decrease the total computation load. When the density of the simulation field is small, the performance of the two libraries does not differ significantly. These results can be motivated by the same explanation given in the previous paragraph. For this experiment, we do not show the results of NetLogo as it was not able to run the simulation using the experiment configurations.

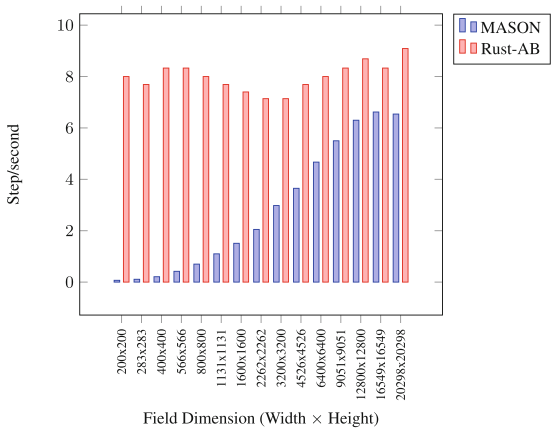


Fig. 2. Rust-AB vs MASON: constant number of agents.

6 Conclusion and Future Works

This work introduces the library *Rust-AB*, a discrete events simulation engine for ABM simulations written in Rust. Rust-AB is designed to be a *ready-to-use* tool for the ABM community and, for this reason, the architectural concepts of the well-adopted MASON library were re-engineered. We then described an example of Rust-AB simulation, implementing the Boids model, to investigate the performance of Rust-AB in comparison with MASON. Results, exploiting Rust-AB Alpha 1.0 release, are promising and exhibits a performance-enhancing compared to the MASON toolkit; in particular, for simulations with a high agent density. We consider this outcome as an important step ahead at disclosing the power of the Rust language to develop ABM simulations.

We plan to continue the development of the library to improve the simulation performance, through better exploitation of Rust peculiarities, such as safe concurrency capabilities. Moreover, we will experiment Rust-AB by developing more computing-intensive simulations, using models where the agent interactions are not only local but also exploits network-based interactions, as presented in [2].

References

1. Abar, S., Theodoropoulos, G., Lemarinier, P., O'Hare, G.: Agent based modelling and simulation tools: a review of the state-of-art software. Comput. Sci. Rev. **24**, 13–33 (2017)
2. Antelmi, A., Cordasco, G., Spagnuolo, C., Vicidomini, L.: On evaluating graph partitioning algorithms for distributed agent based models on networks. In: Hunold, S., et al. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 367–378. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_30
3. Carmine, S.: Rust-AB: An Agent Based Simulation engine in Rust (2018). https://github.com/spagnuolocarmine/abm
4. Collier, N., North, M.: Parallel agent-based simulation with repast for high performance computing. Simulation **89**, 1215–1235 (2013)
5. Cordasco, G., Spagnuolo, C., Scarano, V.: Toward the new version of D-MASON: efficiency, effectiveness and correctness in parallel and distributed agent-based simulations. In: IEEE International Parallel and Distributed Processing Symposium Workshops (2016)
6. Cordasco, G., De Chiara, R., Raia, F., Scarano, V., Spagnuolo, C., Vicidomini, L.: Designing computational steering facilities for distributed agent based simulations. In: Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (2013)
7. Cordasco, G., Mancuso, A., Milone, F., Spagnuolo, C.: Communication strategies in distributed agent-based simulations: the experience with D-Mason. In: an Mey, D., et al. (eds.) Euro-Par 2013. LNCS, vol. 8374, pp. 533–543. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54420-0_52
8. Heath, B., Hill, R., Ciarallo, F.: A survey of agent-based modeling practices (January 1998 to July 2008). J. Artif. Soc. Soc. Simul. **12**(4), 9 (2009)
9. Holcombe, M., Coakley, S., Smallwood, R.: A general framework for agent-based modelling of complex systems. In: Proceedings of the European Conference on Complex Systems (2006)
10. Kleinberg, J.: The small-world phenomenon: an algorithmic perspective (2000)
11. Macal, C., North, M.: Tutorial on agent-based modeling and simulation Part 2: how to model with agents (2006)
12. Mahé, F., Rognes, T., Quince, C., de Vargas, C., Dunthorn, M.: Swarm: robust and fast clustering method for amplicon-based studies. PeerJ **2**, e593 (2014)
13. North, M.J., et al.: Complex adaptive systems modeling with repast simphony. Complex Adapt. Syst. Model. **1**(1), 3 (2013)
14. Resnick, M.: StarLogo: an environment for decentralized modeling and decentralized thinking. In: Conference Companion on Human Factors in Computing Systems (1996)
15. Reynolds, C.W.: Flocks, herds, and schools: a distributed behavioral model. Comput. Graph. (ACM) **21**, 25–34 (1987)
16. Richmond, P., Chimeh, M.K.: FLAME GPU: complex system simulation framework. In: 2017 International Conference on High Performance Computing Simulation (2017)
17. Tejaswi, V., Bindu, P., Thilagam, P.: Diffusion models and approaches for influence maximization in social networks (2016)
18. Thurner, S., Klimek, P., Hanel, R.: Introduction to the Theory of Complex Systems. Oxford University Press, Oxford (2018)
19. Wang, H., et al.: Scalability in the MASON multi-agent simulation system (2019)
20. Wilensky, U.: NetLogo 3.1. 3 (2006)