PandoRust. Agent Based Modeling in Rust

Miguel Esteban *BSC—CNS,* Fernando Cuchietti *BSC–CNS,* Germán Navarro *BSC–CNS*

Abstract—The emergence of next-generation programming languages such Rust, focused on concurrency and parallelism, can improve the performance of existing Agent Based Modeling platforms. In this study an existing ABM framework, Pandora, is redesign using Rustlang's capabilities for safe parallelism to assess whether is possible to use Rust in ABM. It was found that Rust ownership policies, if correctly implemented, can massively improve the performance of this type of parallel projects and reduce the lines of code to maintain, improving the development experience while maintaining the high performance needed for the execution of complex social models.

Resumen—La aparición de lenguajes de programación de nueva generación como Rust, centrados en la concurrencia y el paralelismo, puede mejorar el rendimiento de las plataformas de simulación basadas en agentes actuales. En este estudio se rediseña un modelo existente, Pandora, aprobechando las características que ofrece Rustlang en materia de paralelismo para evaluar si es posible usar este lenguage para ABM. La implementación de estas políticas permite mejorar enormemente el rendimiento de este tipo de proyectos y reducir las líneas de código a mantener, mejorando la experiencia de desarrollo sin afectar las necesidades de rendimiento para la ejecución de modelos sociales complejos.

Resum—L'aparició de llenguatges de programació de nova generació com Rust, centrats en la concurrència i el paral·lelisme, poden millorar el rendiment de les plataformes de simulació basades en agents (ABM) d'avui dia. En aquest estudi es redissenya un model existent, Pandora, aprofitant les característiques que ofereix Rustlang en matèria de paral·lelisme per avaluar si és possible fer servir aquest llenguatge per a ABM. La implementació d'aquestes polítiques permet millorar enormement el rendiment d'aquest tipus de softwares i reduir les línies de codi a mantenir, millorant l'experiència de desenvolupament sense afectar les necessitats de rendiment per a l'execució de models socials complexos.

Index Terms—Agent Based Modeling, Urban Digital Twin, Mobility, Social Simulation, Rust

1 Introduction

AGENT based modeling is a powerful simulation technique widely applied in computational sociology for validating hypothesis. ABM platforms can be classified between simple fast-prototyping —MASON— and advanced high-performance —RepastHPC—. The formers have had a huge success thanks the little knowledge needed to deploy a model, but fell short on scalability compared to complex tools that use compiled languages, mainly C++, to achieve high levels of performance.

Pandora is an open-source simulation framework created by the BSC—CNS in 2014 with the aim of being an easy-to-learn tool for social scientist based on a Python library, but as powerful as its complex counterparts.

Since 2014, the Computer Science has evolve greatly; nowadays, computers have multiple cores and these architectures have shaped the next-generation programming languages —like Rustlang or Goland—, focusing on parallelization and providing new tools for facilitating the complex task of parallel software development. One of this next-generation languages is Rustlang, a multi-paradigm general-purpose language that enforces memory safety without the need of a garbage collector. Emphasises performance and concurrency using an *ownership* model that prevents race conditions and enforces memory safety. For this safety-first approach is being progressively introduced in the Linux kernel.

1.1 Objectives

The main objective of this article is to assess the feasibility of developing a ABM framework on Rustlang and improve the performance of the original Pandora implementation making use of the ownership model as well as the high level functional programming structures to make it more accessible to the general public.

This project will adapt the original framework developed on C++ to better suit its main use over the time, pedestrian modeling.

2 Rustlang

Rustlang —or just *Rust*— is a general-purpose programming language designed for **performance and safe concurrency** by enforcing memory safety without the requiring the use of garbage collectors.

2.1 Memory management

When a variable is created is immutable by default, this is due to the safety-first approach that Rust takes making the programmers aware of when a variable is mutated, thus reducing bugs.

2.1.1 Ownership system

It is a set of rules that governs how Rust manages memory; enables Rustlang to make safety guarantees without the need of a garbage collector.

- Each value in Rust has a variable called **owner**.
- There can only be **one owner at a time**.
- When the owner goes out of scope, the value is **dropped**.

2.1.2 References and borrowing

In Rust, a reference is a pointer to a memory address that is owned by some variable. Unlike a regular pointer, a reference is guaranteed to point to a valid value of a particular type and allows **functions** to operate without the needing of taking ownership of the value.

In the case that a function must modify a borrowed value, it is needed a mutable reference, making clear that a function will mutate the value it is borrowing. These references have one restriction: it can only be **one mutable reference** to a particular data **at a time**.

2.1.3 Concurrency

By leveraging ownership and type checking, many concurrency errors are compile-time instead of run-time, this approach has been named *fearless concurrency*. Rust, being a low-level language, offers a variety of tools to provide the best performance in any situation. In this article I will only be focused on **shared-state concurrency**, as is the one that will be used.

2.2 Rayon

Rayon is the *de-facto* crate for data-parallelism in Rust. Its goal is to make sequential loops or iterators run in parallel. It uses a technique called **work-stealing** that adapts to the CPU load [1] and only applies parallelism if there are IDLE threads available. Rayon's APIs guarantees data-race freedom removing the majority of parallel bugs, the parallel iterators are guaranteed to produce the same result as their sequential counterparts —if the iterator has *side effects*, e.g. write to disk, those may occur in a different order— but with a higher performance.

2.3 Performance & Programming Effort

Rust optimized versions of double-precision algorithms achieve similar performance to its C++ counterpart; how ever, single-precision one do perform significantly worst because Rust does not optimize these type of operations as well as C++ does.

Rust also offers the benefits of high-level languages – object-oriented and functional programming capabilities– generating more compact code easier to maintain without loosing significant performance in these abstractions [2].

3 PANDORA

Pandora is an open-source Agent Modeling Framework designed to fill the gap between prototyping (MASON [3]) and advanced simulations (Repast-HPC [4]) [5]. The platform has a flexible structure capable of providing the tools needed for creating any type of ABM and execute the environment in a transparent way [5].

3.1 General architecture

The general architecture of Pandora is shown in Figure 1. The classes `World` and `Agent` are the core of the framework.

- `World`. Manages the different layers of information that defines the environment.

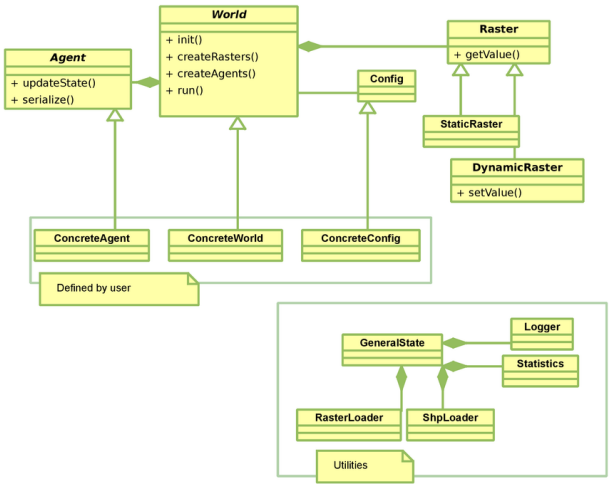

Figure 1. Class diagram of Pandora

- `Config`. Provides the information needed to initialize the environment.
- `Agent`. Encapsulates any entity with internal state, decision making process and behaviour. Can be stored in a file using HDF5 protocol.

3.2 Scheduler

The current distributed scheduler of Pandora is based on **spatial partition**, each node owns a section of the entire environment, containing the landscape and agents; this solution is also used by Repast-HPC [4].

The entire simulation is evenly distributed periodically by a load balancer utility between all nodes, border-agents information is distributed between neighbours in every step of the simulation in order to keep updated information in the whole scenario. The spatial section owned by each node is divided in 4 equal inner regions, as seen in Figure 2. The agents located in the section 0 are executed simultaneously; as there is no possible collision with other regions, once is completed modified border data is shared across neighbours and the next region begins its execution. Once all regions are simulated, the entire state is serialized and a new step can be evaluated.

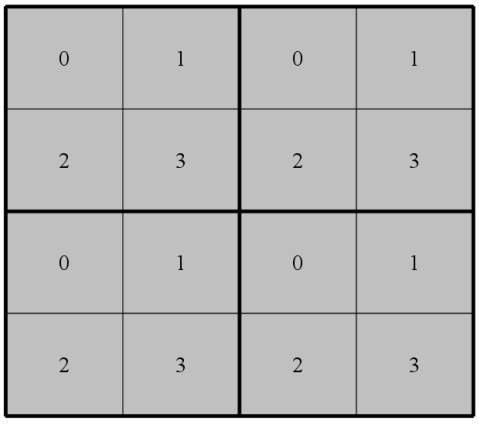

Figure 2. Each node splits its area in four regions executed sequentially

The main drawback of this approach is that agents on region 0 will be executed before than the ones in other sections, this could potentially produce artifacts in the simulation results.

4 PandoRust

PandoRust is a re-evaluation of the original idea presented by Xabier Rubio-Campillo in 2014. The idea of an **easy-to-use framework** continues to be the main focus.

4.1 Framework Design

In this revision of Pandora, the framework has been modified to better suit its main strength, **pedestrian modeling**.

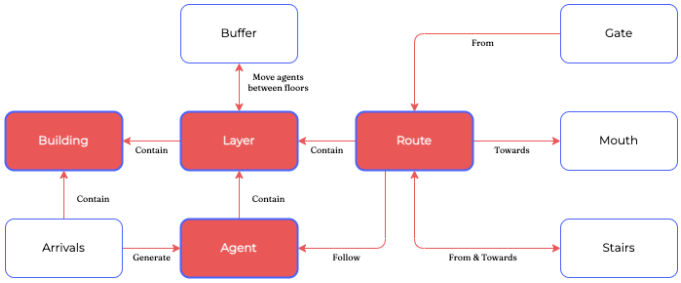

Figure 3. Briefed Pandorust diagram

4.1.1 Structure

Minimal environmental unit that represent important elements —e.g. doors, gates, stairs, elevators—. It is used as reference point for the movement of agents, as they will travel from one structure to another.

4.1.2 Layer

It is the **minimal simulation unit**, contains a simplified representation of the region for pathing purposes. Knows every structure in its domains and an optimal route to reach it from every other structure. Also keeps track of every agent in the region and its local path.

The spatial division of the environment is **modular**, making it possible to assign to each thread one area —e.g. building floor, sidewalk segment, hallway, room, etc—. in a more natural manner without hard edges that could generate artefacts in the simulation results. This method allows the model to have different granularity levels: increased computation resources for small key crowded areas —by reducing the region assigned to each node— and large uncongested open spaces can be modeled by a single thread.

4.1.3 Building

Collection of layers that function as a bridge between them moving agents between floors, introducing agents at the correct time. It also records the local path followed by a given agent when leaves it.

4.1.4 Route

A route is a **collection of optimal paths** between two structures. When required, can be inverted to accommodate bidirectional movement of agents, e.g. sidewalks. It stores all possible combinations between the pixels of both structures and returns a random path each time is called. Due to time constrains, if an agent is not capable of being given a precomputed route, it is discarded from the simulation.

4.1.5 Buffer

Each structure is linked to a **queue** with simulates the movement between layers and can be tuned to accurately describe the real time needed for moving from one layer to another, e.g. crosswalks or stairs.

4.1.6 Agent

Entity that interacts with the simulation environment while following a predefined route. This route can be modified by the behaviour of the agent allowing **complete freedom** of movement during its commute.

4.1.7 Auxiliary tools

- **Configuration parser**. A Python script was developed for converting original Pandora `XML` configuration files to `TOML` [6], the default Rust configuration file format.
- **Model serialization**. When a model is firstly created, it is saved as `BINCODE` format [7]. This file can be later loaded reducing the computational time required for arranging the simulation environment.
- **Output formatting**. For visualization and analysis purposes, a `CSV` file is generated after ever simulation with the path followed by every agent in the simulation.

4.2 Collisions system

A new collision system has been proposed with the main purpose of **abstracting agents from their path**, where agents only know their next movement, not the global path; allowing the parallel execution of a subset of the agents that do not interfere with other entities to move simultaneously, see Figure 4.

Require: Agents local path

function COLLISIONS(idx)
  posCounter ← ∅ ▷ Empty Hash table
  posCounter[idx] += 1
  return posCounter == 1 ▷ Safe locations
end function

Figure 4. Collision detecting algorithm

Once the agents are divided, those without data dependencies —collisions— can be executed in parallel; those who collide, have to be executed sequentially. The collision resolution system has to be implemented by the model to better suit the behaviour that is being analyzed.

5 IoTwins model

IoTwins aims at enabling SMEs in the manufacturing and facility management/service sectors to access edge-enabled and cloud-based big-data analysis services to create hybrid digital companions to improve their production process and optimize the management of their facilities.

Within this project a digital twin of the Camp Nou is being created in order to make possible the simulation of the movements of all the types of fan that attend the enclosure, modeling agglomerations at the facilities and settling down a new standard of security in this type of enclosures.

5.1 Pandora implementation

The original implementation of the model was made on Pandora, its approach is based on **on-demand path computing** for all agents. Once the agent has arrived to an stair, another a-star is executed to find the closest opposite stair in the corresponding layer.

The agents behaviour later explained on subsection 5.4 is a simplified version of the one implemented in the Pandora model, it adds different behaviour for a set of agents called *tourist* that have a higher chance of wandering.

5.2 Building tagging

5.2.1 Structure detection

The data provided by the Fútbol Club Barcelona for the digital recreation of the Camp Nou is composed by a set of 8-bits blueprints from the arena at scale $1:40$ —see figure 5— and a file with the location of all 107 gates.

Figure 5. Blueprint samples.

The proposed algorithm for detecting stairs and elevators is detailed on Figure 6. Obtains all structures traversing only once per matrix position.

5.2.2 Structure connection

In order to correctly model stairs and elevators, up-stairs shall be connected with other layers' down-stairs and elevators with the same position in other layers —if such layer is accessible—.

For linking structures together, the closest structure within 1.2 m. —3 px— range is selected. This method might produce crossed-structures, mixing together two stairs divided by a wall. If there is no structure inside the region, that stair is considered not to have a exit in that layer.

```
Require: blueprint image
Require: Values of interest

function CONTIGUOUS(idx, gridSize)
    return [idx ± gridSize ± 1]
end function

procedure STRUCTURES(blueprint)
    for all px in bluerpint do
        structure ← ∅                                  ▷ Empty array
        if px in interestValues & px_i not visited then
            queue ← contiguous(px_i, gridSize)
            visited ← px_i
            structure ← px_i
            for all contiguous_px ∈ queue do
                if cont_px = px then
                    queue ← contiguous(cont_px_i, ...)
                    facility ← cont_px_i
                    visited ← cont_px_i
                end if
            end for
        end if
        structure                          ▷ Grouped structure pixels
    end for
end procedure
```

Figure 6. Structure finding algorithm

5.3 Building routes

Given the nature of the simulation and the venue, only a subset of all possible routes has been computed, can be classified in two groups:

- Gates towards up-stairs and mouths
- Down-stairs towards up-stairs and mouths

As the entry points to the simulation and layer respectively, these structures are from where the agents will move towards their local destinations.

Every agent will look for the closest stair with access to their destination floor and move towards it. When on their seat layer, it will be assigned the route from the stair to the mouth.

5.4 Agents

5.4.1 Agent arrival

An arrival is temporal data point providing the **minute** when the agent shall enter the simulation, the **gate** from where the agent group will start, destination **mouth** where the agent will arrive[1] and **number of agents** that conform the arrival.

Arrivals are queued on each gate and released in a staggered manner following a queue in every gate, this allows to estimate agglomerations in surrounding areas as well as in the stairs inside the venue.

1. IoTwins only models the path towards stadium sectors

5.4.2 Agent pathing

Agents' global route is tailored individually for each arrival following this structure of local paths.

1) Route from their gate towards the closest stair with access to their destination layer.
2) Route from the stair they arrived towards their mouth.

This pathing behaviour has been chosen as is the most common pattern followed by **season tickets holders** fans as no distinction is made between tourists and fans in this smaller model; inverted routes are also possible for evacuation simulations where the stadium has to be emptied.

5.4.3 Agent behaviour

Every agent in the simulation has an inherit **interest** for the venue is in. This is how tourist and season tickets holders are differentiate. The former does not move directly towards its seat, walking around the stadium admiring the arena and the warm-up; the later goes directly towards their seat and arrives minutes before the match starts. This interest is defined for each agent following a **Normal distribution** and modeled to decay over the time, at a pace of 3% every minute —200 simulation steps—.

- Walk. Default action following the route from the initial structure towards the destination one.
- Wander. Deviation from the route that makes a 3 metres —10 simulation steps— stochastic movement followed by a path for rejoining the route.
- Stay in place. The agent does not move from the current location.

The action called for each agent is based on the interest and distributed in the following manner, being p the interest and a the action chosen for that step.

Table 1
Season ticket holder behaviour

Walk	**Wander**	**Stop**
$a > p$	$0.15p < a < p$	$a < 0.15p$

As previously mentioned, no distinction is made between tourist and season tickets holders, simulation rates for each action are tuned for the latter.

5.4.4 Agent collision

When an agent has announced to move into the same position as other, the previously explained collision mechanism is applied, and the agent will check if the other 3 possible locations are occupied; if that is the case, the agent will stay in the same place, otherwise will move randomly towards one of those positions and activate the wandering mechanism that simulates a person looking for an alternative route.

5.4.5 Wandering mechanism

When activated, a random 10-step —3 meters— path is computed from the current location, once this *wander path* is known, a new one is computed from its last location towards the route given by the layer. The joining position of the path is given by eq. 1.

$$\text{joinPos} = \text{steps} + (\text{pathLen} - \text{steps}) \ / \ 4 \tag{1}$$

After the destination position is known, a new route is computed, in case of no existing a possible path between those points, the wandered path is inverted.

6 Results

All simulations have been executed on a single node Ubuntu 16.04.7 (Xenial Xerus) LTS powered by an Intel Xeon Silver 4114 2.20 GHz CPU and a Docker container based on the Microsoft VS Code Rust development container [8] using Rust 1.63 (nightly).

6.1 Benchmark

All simulations have been executed with 40000 steps —200 minutes— starting 150 minutes before the kick off.

Table 2
Resolution: 627 px. Execution time (seconds)

# Agents	1 thread	2 threads	5 threads	10 threads
16000	166.71	**145.51**	156.63	169.63
20000	124.35	**109.57**	125.19	125.14
32000	304.12	**244.23**	276.58	298.08
Improvement	1x	**1.176x**	1.052x	0.998x

From table 2 can be observed that a simulation resolution of 627 pixels is too small for getting significant improvements over sequential computation; however, as the number of agents grow, so does the number of active layer —layers populated with agents— at the same time. During simulations, for most of the time only layers S1 and PB had agents at the same time, as those are the arriving layers and the ones with the higher number of mouths, this explains why 2 threads archives the highest performance.

6.2 Pandora comparison

Table 3
Execution time (seconds) comparison

# Agents	Pandora	PandoRust	Improvement
16000	23613.9	**145.51**	162.28x
20000	42330.8	**109.57**	386.33x
32000	155274	**244.23**	635.77x

As shown in table 3, the tagging of the simulation environment and the pre-computing of paths has proven to be a better approach in terms of computation times, where multiple tests where carried out with the same environment.

6.3 Programming effort

Pandorust in its current state takes 1955 lines of code, whereas the original Pandora 20639. This difference is in part produced by the lack of features —due to time constrains— compared to Pandora that can outputting the results to HDF5 and has MPI support for distributed computing; however, the lack of a scheduler, as commissions that job to Rayon, reduces substantially the lines of code to maintain together with the use of **iterators**, a high-level feature with zero-cost abstractions over loops that, when compiled, produces the same instructions as a hand-written C program in a more idiomatic manner and with less lines of code [9].

7 Conclusion

This article has proved Rust as an alternative to C++ for Agent Based Modeling in high performance environments, improving the overall performance of the Pandora framework —PandoRust— while trusting the compiler for optimizing the models, keeping the original idea of an easy-to-use framework trough a Python library that could be wrapped in a future work.

While the benchmark model was not big enough to prove the new design in terms of heavy load parallelization, the sequential execution of the model already shows a better performance than its C++ counterpart.

Due to time constrains, no alternative method to the predefined routes strategy could be developed, allowing complete environment path finding for difficult routes, so agents without a route are discarded from the simulation. This is mainly produced by the low resolution of the blueprints, that removed hallways between areas of the stadium making impossible to connect some structures. During the benchmarking process, the arrival file had to be hand-checked to ensure that the number of agents with reachable destination relates with the expected on table 2 for comparison purposes.

During this project Rustlang has being treated as a high-level programming language, as is mainly based on loops that have been implemented through iterators that made for clearer and simple code without reducing performance.

7.1 Future work

This implementation is a proof-of-concept; as so, part of the code has been directly implemented by the model rather than in the framework itself. The code should be **refactored to a generic version** of itself following Rust coding conventions for making possible the deployment of any model.

The python library could be adapted as well to allow for selecting the back-end and provide support for both legacy and new models.

A bigger model is needed for truly test its distributed capabilities and tune how layers should be modeled.

Acknowledgments

I would like to thank Fernando for his mentorship during the whole project, Germán for sharing everything about Pandora, Xavier Licerán for solving every doubt I had about the Camp Nou and Eduardo Cesar Galobardes for all the support during the last 4 years.

References

[1] N. Matsakis, *Rayon: Data parallelism in rust*, Dec. 2015. [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/.

[2] M. Costanzo, E. Rucci, M. Naiouf, and A. De Giusti, "Performance vs programming effort between rust and c on multicore architectures: Case study in n-body," Jul. 2021.

[3] G. C. Balan, "Mason: A java multi-agent simulation library," 2009.

[4] N. Collier and M. North, "Parallel agent-based simulation with repast for high performance computing," *SIMULATION*, vol. 89, no. 10, pp. 1215–1235, 2013. DOI: 10.1177/0037549712462620. eprint: https://doi.org/10.1177/0037549712462620. [Online]. Available: https://doi.org/10.1177/0037549712462620.

[5] X. Rubio-Campillo, "Pandora: A versatile agent-based modelling platform for social simulation," *Proceedings of SIMUL*, pp. 29–34, 2014.

[6] T. Preston-Werner, *Toml: Tom's obvious minimal language*, 2022. [Online]. Available: https://toml.io/en/.

[7] bincode-org, *Bincode*. [Online]. Available: https://github.com/bincode-org/bincode.

[8] *Vs code development containers rust*, https://github.com/microsoft/vscode-remote-try-rust.

[9] R. van Asseldonk, *Zero-cost abstractions*, 2016. [Online]. Available: https://ruudvanasseldonk.com/2016/11/30/zero-cost-abstractions.