High-Performance Computation on a Rust-based distributed ABM engine

Daniele De Vinco[1,*], Andrea Tranquillo[1], Alessia Antelmi[2], Carmine Spagnuolo[1] and Vittorio Scarano[1]

[1]Università degli Studi di Salerno, Salerno, Italy
[2]Università degli Studi di Torino, Torino, Italy

Abstract

An agent-based model (ABM) is a computational model for simulating autonomous agents' actions and interactions to understand a system's behavior and what governs its outcomes. When the data or number of agents grow or multiple runs are necessary, agent-based simulations are generally computationally costly. Therefore, adopting different computing paradigms, such as the distributed one, is essential to manage long-running simulations. The main problem with this approach is finding a way to distribute and balance the simulation field so that the agents can move from one machine to another with the least amount of synchronization overhead. Based on our experiences, we present a Rust-based ABM engine capable of distributing models on high-performance computing resources, gaining remarkable speedup against the sequential version.

Keywords

High-performance computing, Agent-based modeling, Distributed computing, Simulation, Complex systems, Computational social science

1. Introduction

Modeling real-world phenomena is incredibly challenging due to the intricate interactions among numerous interconnected elements. Understanding these systems is nearly impossible when they are viewed in isolation. Consequently, such systems are often referred to as complex systems, though a precise definition of complexity remains elusive [1]. These systems typically share features such as non-linearity, decentralized control, and feedback mechanisms. In recent years, Computational Science has leveraged data-intensive computing and analysis to study such issues. ABMs offer a bottom-up approach for analyzing complex systems, allowing modelers to design the behaviors of individual agents (e.g., members of a population) and the environments in which they operate. The interactions among these agents in the simulated environment produce emergent properties and phenomena that the modeler aims to examine and understand. These three components (behavior, environment, and interactions) are the building blocks of an ABM and have been proven to be very effective in modeling different scenarios across a vast corpora of fields [2].

ABMs are also a recurring theme in High-Performance Computing (HPC), since these models are designed to mimic social interactions, the global economy, or natural phenomena. In addition, they can help predict potential outcomes involving numerous entities. However, when the number of agents consistently grows, traditional ABM engines fail because the computational power required by a single execution becomes an unbearable limitation. The literature states that ABMs can intersect with HPC through two distinct ways: the outer and inner loops [3]. The former usually describes optimization techniques such as model parameters sweeping [4]. The latter, which is also the focus of this paper,

BigHPC2024: Special Track on Big Data and High-Performance Computing, co-located with the 3rd Italian Conference on Big Data and Data Science, ITADATA2024, September 17 – 19, 2024, Pisa, Italy.

*Corresponding author.
ddevinco@unisa.it (D. D. Vinco); alessia.antelmi@unito.it (A. Antelmi); cspagnuolo@unisa.it (C. Spagnuolo); vitsca@unisa.it (V. Scarano)
0000-0003-0781-3744 (D. D. Vinco); 0000-0002-6366-0546 (A. Antelmi); 0000-0002-8267-9808 (C. Spagnuolo); 0000-0001-8437-5253 (V. Scarano)

typically involves distributing a model across different computational nodes using de facto standards such as MPI (Message Protocol Interface) [5] or OpenMP (Open Multiprocessing) [6].

This paper presents the early stages of implementing a distributed version of krABMaga, an ABM engine written entirely in Rust. We have employed our experiences in ABMs and the distributed computing field to enhance the capabilities of the krABMaga engine, assessing the possibilities and opportunities for deploying a high-optimized tool to an HPC cluster with minimal effort and achieving good results [4].

2. Related Work

ABMs have been extensively studied and applied across various domains, providing valuable insights into complex systems through simulations [7, 8]. However, while there is a substantial body of work focused on ABMs in general, research into their distributed versions is relatively limited, with tools outdated or not supported anymore.

Generally, distributed ABMs involve partitioning the simulation across multiple computational nodes to handle larger-scale models or to achieve higher performance. This approach can significantly improve the scalability and efficiency of simulations by leveraging parallel processing and distributed computing resources. Despite its potential, the distributed version of ABMs presents additional challenges, such as managing communication between nodes, ensuring data consistency, and effectively balancing the computational load. Several frameworks have explored a distributed approach, such as:

Mason [9]. Mason is a Java-based ABM simulation toolkit. A distributed version of this library, known as D-Mason [10], was developed to enhance performance. It uses a Quad-Tree data structure to manage the simulation field.

Flame [11]. Flame is an ABM system designed to run on a wide range of heterogeneous computing platforms. It offers a formal framework for model creation using the XXML language, a variant of XML, along with automatic parallel code generation.

Flame-GPU [12]. Flame-GPU extends Flame to support GPU translation. It simplifies GPU programming by using an API that leverages the FLAME template to generate CUDA code for target GPU devices, eliminating the need for users to engage directly with GPU programming languages or optimization techniques.

Pandora [13]. Pandora is an ABM framework for large-scale distributed simulations. It provides two identical programming interfaces in different programming languages, one of them including the automatic generation of parallel and distributed code.

Repast-HPC [14]. Repast-HPC is a component of the Repast suite, specifically designed for large-scale simulations on C++-based systems. It is tailored for execution on large computing clusters and supercomputers.

This work revolves around a distributed version of krABMaga[1], an open-source discrete events simulation engine written in Rust for developing ABM simulations [4, 15]. The distributed engine, as described in the next sections, simplifies the simulation development process by abstracting the complexities involved in distributed computing. Therefore, this implementation allows modelers to focus on the core logic of their simulations without getting bogged down by technical issues and communication layers. KrABMaga aims to be an intuitive toolbox inspired by the popular MASON library, particularly its modular design that separates the simulation and visualization subsystems. The Rust programming language underpins this approach [16], thanks to its main principles:

- *Performance*: Rust offers both speed and memory efficiency. Its memory model eliminates the need for a garbage collector at runtime, making it well-suited for critical services, embedded devices, and seamless integration with other languages.

[1] https://github.com/krabmaga/krabmaga/

- *Reliability*: It exploits its ownership and borrowing system to guarantee memory and thread safety, removing many classes of bugs at compile time.
- *Productivity*: The language is shipped with great documentation, a friendly compiler with useful error messages, and a fast-growing community that has written a large corpus of high-optimized crates (libraries in the Rust ecosystem).

Rust stands out by performance similar to C, which can shorten the duration of a single simulation and by a unique programming approach that ensures no memory-related bugs occur throughout long-running experiments, hence enabling simulation reliability [17]. Every ABM engine needs each simulation to have at least two important components: a state and an agent. The state describes the environment of the simulation, which contains different elements, such as a field and its dimensions, the number of steps, the initial number of agents and others, while the agent contains the behavior of the population inside the simulation. Thanks to this decoupled structure, the simulation field can be modified without impacting other parts of the simulation, such as the agent behavior or the simulation parameters. Finally, the framework provides additional functionalities, such as monitoring, reproducibility, and visualization tools.

3. Methodology

A key aspect of ABM computation is the communication between agents. Typically, each simulation agent needs to identify its neighbors to exchange information and perform its tasks. In sequential execution, this process is routine and has a moderate impact on performance. However, in distributed execution, where neighbors may be located on different machines, discovering and interacting with them can become a significant performance bottleneck. Moreover, when an agent moves between partitions or is removed from the simulation, the process must proceed seamlessly as if all agents were in a single field. Addressing these challenges is crucial when developing an efficient distributed system that can manage multiple partitions of the same field across remote machines.

To facilitate the distribution of the simulation, we began by modifying an existing field in our framework. Our first attempt takes as a reference the *Field2D* implementation on the krABMaga repository[2], which is the standard field where an agent can move on a continuous 2D space.

Sequential structure. The *Field2D* uses a two-dimensional toroidal grid as a simulation field characterized by an origin point (x;y), a width, and a height. Each grid coordinate identifies a cell in which an agent can reside and interact with other agents.

Distributed structure. K-Dimensional Tree (K-D Tree) data structures are widely used in distributed computing, particularly for tasks like clustering and closest neighbors search when a scalable solution is required [18]. A K-D Tree is a tree structure in which each node has exactly two children and can be split until the desired number of nodes is reached. For each pair of children created, the parent node keeps references to them, allowing us to reach any leaf, starting from the root, in a short time [19]. We implemented a customized version of a K-D Tree, where each child node stores references to all other nodes created in the tree, referred to as blocks. Each block represents a segment of the simulation field and includes an ID corresponding to the process rank it will be assigned to, its origin point (x, y), as well as its width and height, as shown in Figure 1. By maintaining references to every child node, each node gains access to comprehensive information about all nodes, facilitating efficient neighbor search and synchronization operations. Although this modification increases the physical space required, the resulting efficiency in communication justifies this trade-off. Moreover, this approach remains practical since the growth rate of machines — and thus partitions — does not scale as rapidly as the number of agents.

After the main process has computed each block, it sends a reference to each block to all other processes. These processes then receive and store the reference in local memory, allowing them to communicate using the associated ID when necessary.

[2]https://github.com/krABMaga/krABMaga/blob/main/src/engine/fields/field_2d.rs

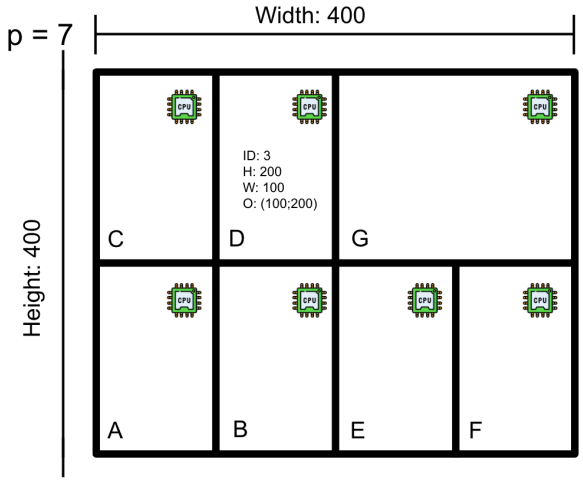

Figure 1: Partitioning of the field when there are 7 processors. Each block has a unique ID, height and width of its perimeter, an origin point, and it is assigned to a different processor (or machine).

When an agent changes position during the simulation and exits from a block's border, it must be sent to another block. Since every block knows the exact size and ID of all the others, when an agent moves outside its border, it can easily calculate the ID of the processor that will be responsible for the agent based on its position. To make this exchange possible, the object is saved into an array whenever an agent moves outside the border of its process field. At the end of every step, every process sends all the agents who have moved off their partition to their respective neighbors and receives all agents sent by their neighbors. This phase is preceded by a message exchange phase, where each process sends the number of agents it will exchange to each neighbor and receives the value from all its neighbors. When this process is complete, each process allocates the buffer with adequate slots for the incoming agents. This communication phase is handled using MPI collective operations, such as scatter and gather, combined with non-blocking send and receive operations to facilitate data exchange across processors efficiently.

Additionally, in many simulations, an agent needs to be aware of nearby agents within a specific area of interest (AOI), usually defined by a fixed-size radius (see Figure 2). This can be particularly challenging in distributed simulations because an agent's AOI may be divided across different processes. To accomplish this task, it is essential to identify first agents that could be neighbors of agents from other processes. This process is facilitated by Halo Regions, of which an example is illustrated in Figure 2. A Halo Region is a fixed-length zone located near the borders. When an agent moves inside the field and enters a Halo region, a copy of the agent is placed inside an array of agents, indexed by the Halo Region that keeps it. At the end of every step, these agents are sent to the potential neighbors and received from each one. This operation uses the same principles of the exchange between processors. Once all agents have been received, the process can calculate the neighbors of each agent, considering both the local agents and the received agents that are in the AOI.

4. Results

To evaluate the proposed implementation, we have chosen Flockers as the main benchmark model [20]. Tests are based on known parameters for the model [15]. The number of steps for each simulation is fixed to 200. The model is evaluated using the configurations for the number of agents and field dimensions listed in Table 1.

The code to reproduce the model is available on the GitHub repository[3]. We have built a cluster on Microsoft Azure[4] to make the benchmark as fair as possible, eliminating the noise from background

[3]https://github.com/krABMaga/examples/tree/main/flockers_mpi
[4]https://azure.microsoft.com

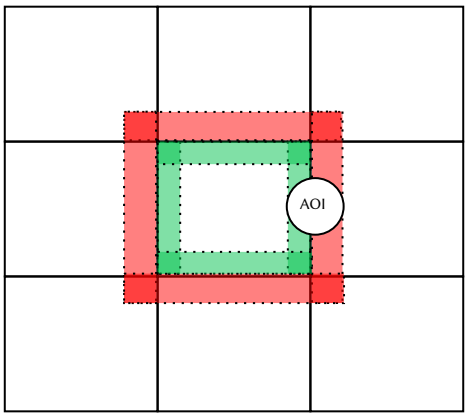

Figure 2: An example of halo region. In this example, the central square represents the main actor. The red-highlighted areas are the Halo regions of its neighboring agents, while the green areas indicate the portions of the field shared with those neighbors. Agents located within these green regions are locally stored by the processor managing them. At the end of each step, processors sharing borders exchange information about the agents in the red halo regions and, if needed, transfer agents between processors.

Table 1
Size of the examples.

# Agents	Field size	Density
1Million	3162x3162	$\approx 10\%$
2Million	4472x4472	$\approx 10\%$
5Million	7071x7071	$\approx 10\%$
10Million	10000x10000	$\approx 10\%$

activities on local machines. Each virtual machine on the Azure cluster was created within a Proximity Placement Group[5] and has the following specifications:

- *Size*: standard_DS1_v2
- *Number vCPU*: 1
- *CPU family*: Intel Xeon Platinum 8370C (or similar[6]).
- *Memory*: 3.5 GiB
- *S.O.*: Ubuntu Server 22.04 LTS
- *Disk*: 8 GB Standard SSD

All numerical results obtained with an average execution time of 10 runs for each setup are displayed in Table 2. The curve trend of this model is displayed when varying the number of processors in Figure 3. It is evident that when computation is the main task of the distributed system, every configuration performs efficiently, closely approaching the optimal curve. However, it is also apparent that the curves tend to slow down at specific thresholds and can sometimes deteriorate. The primary culprit is the communication phase, which becomes a bottleneck when many halo regions are filled with agents and need to exchange information at each step. The simulation with 1M agents running on 8 processing nodes reveals a speed-up exceeding 8, which is an unexpected result that warrants further investigation. This anomaly could be due to near-perfect system load balancing or optimal memory alignment with the machine's cache. However, these explanations seem improbable, considering the benchmark was executed multiple times with randomized seed values. These results highlight the need for load balancing to optimize the size of each partition, thereby reducing the number of agents that need to be exchanged each step.

[5]https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/
[6]https://learn.microsoft.com/it-it/azure/virtual-machines/dv2-dsv2-series

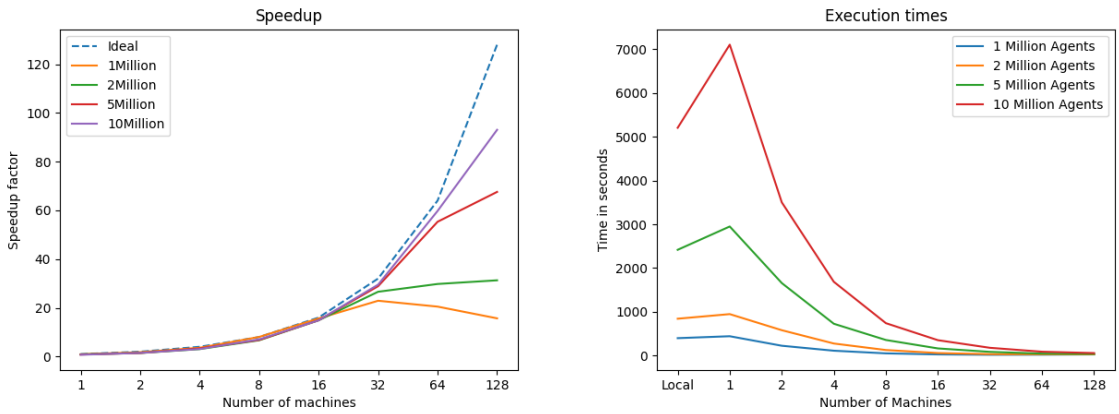

Figure 3: Speedup and execution times of the experiments.

Table 2
Numerical results of the experiments.

	Virtual Machines (vCPU)								
Speedup	**Seq.**	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
1M agents	1,00	0,89	1,78	3,60	8,04	15,64	22,90	20,49	15,68
2M agents	1,00	0,89	1,45	3,06	6,68	14,88	26,59	29,78	31,29
5M agents	1,00	0,82	1,46	3,33	6,80	14,90	28,87	55,32	67,57
10M agents	1,00	0,73	1,49	3,09	7,04	14,86	29,51	59,76	93,05
	Virtual Machines (vCPU)								
Exec. time (sec)	**Seq.**	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
1M agents	397,2	441,1	223,4	110,2	49,3	25,4	17,3	19,4	25,3
2M agents	841,4	946,9	578,8	274,4	125,8	56,5	31,6	28,2	26,8
5M agents	2417,1	2950,2	1656,4	726,2	355,3	162,2	83,7	43,6	35,7
10M agents	5206,9	7107,3	3501,8	1685,0	739,8	350,4	176,4	87,1	55,9

5. Conclusion

The results demonstrate that the current implementation achieves significant speed-up with an increasing number of processors, outperforming the sequential algorithm even with just 2 processes. Additionally, this implementation allows each simulation to utilize the modified K-D Tree with minimal adjustments required to the sequential model. These changes are related to the state and the launching parameters of the simulation and involve the necessity, until a new version of the library is released, to define the *Equivalence* trait on the agent to make the communication with MPI possible.

This work comes with two main limitations. First, the field structure does not implement any load-balancing system. The lack of this feature explains the degradation of the model's performance as the number of agents increases. An example of how the load balance should work is shown in Figure 4. Second, the tests have been evaluated only on the krABMaga engine, not against different engines. The next step should include a more extensive study comparing it with other distributed ABM engines.

Future work should concentrate on balancing the distribution of agents across partitions, ensuring each processor shares a fair amount of work. To provide a more comprehensive assessment, we will evaluate and compare the engine's performance against that of current state-of-the-art open-source engines with available distributed versions. Finally, after a more polished version of the framework is released, we should focus on abstracting the distributed field as much as possible from the modeler. This will make it easier to write a new model from scratch without dealing with the complex layer of the distributed computing paradigm.

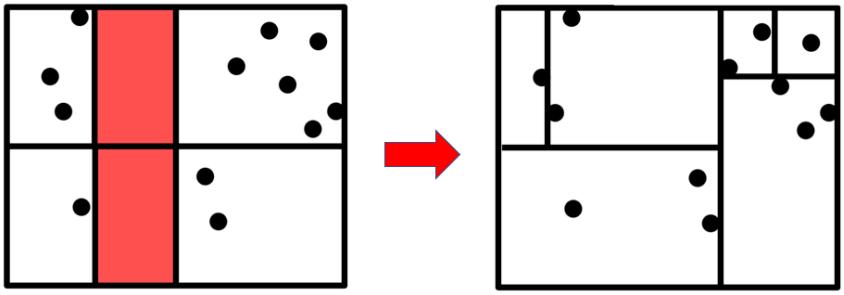

Figure 4: Load balance mechanism in the K-D Tree. Whenever a partition assigned to a processor is empty, or the distribution of the agents is not proportionate (left side), or the agents are heavily located in halo regions, the load balancer should rearrange the partitions to equalize the average work of each processor (right side) and minimize the communications overhead.

Acknowledgments

This work has been partially supported by the spoke “FutureHPC & BigData" of the ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing funded by European Union – NextGenerationEU.

References

[1] J. Ladyman, J. Lambert, K. Wiesner, What is a complex system?, European Journal for Philosophy of Science 3 (2013) 33–67.

[2] L. Xue, G.-P. Liu, W. Hu, All-in-One Framework for Design, Simulation, and Practical Implementation of Distributed Multiagent Control Systems, IEEE Transactions on Systems, Man, and Cybernetics: Systems (2024) 1–14.

[3] N. T. Collier, J. Ozik, E. R. Tatara, Experiences in Developing a Distributed Agent-based Modeling Toolkit with Python, in: 2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC), 2020, pp. 1–12.

[4] A. Antelmi, P. Caramante, G. Cordasco, G. D’Ambrosio, D. De Vinco, F. Foglia, L. Postiglione, C. Spagnuolo, Reliable and Efficient Agent-Based Modeling and Simulation, Journal of Artificial Societies and Social Simulation 27 (2024) 4.

[5] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 4.1, 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[6] R. Chandra, Parallel programming in OpenMP, Morgan kaufmann, 2001.

[7] C. C. Kerr, R. M. Stuart, D. Mistry, R. G. Abeysuriya, K. Rosenfeld, G. R. Hart, R. C. Núñez, J. A. Cohen, P. Selvaraj, B. Hagedorn, et al., Covasim: an agent-based model of COVID-19 dynamics and interventions, PLOS Computational Biology 17 (2021) e1009149.

[8] J. Lohmer, N. Bugert, R. Lasch, Analysis of resilience strategies and ripple effect in blockchain-coordinated supply chains: An agent-based simulation study, International journal of production economics 228 (2020) 107882.

[9] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, G. Balan, Mason: A multiagent simulation environment, Simulation 81 (2005) 517–527.

[10] G. Cordasco, V. Scarano, C. Spagnuolo, Distributed mason: A scalable distributed multi-agent simulation environment, Simulation Modelling Practice and Theory 89 (2018) 15–34.

[11] M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, C. Greenough, Flame: simulating large populations of agents on parallel hardware architectures, in: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1, 2010, pp. 1633–1636.

[12] P. Richmond, D. Walker, S. Coakley, D. Romano, High performance cellular level agent-based simulation with FLAME for the GPU, Briefings in bioinformatics 11 (2010) 334–347.

[13] X. Rubio-Campillo, Pandora: a versatile agent-based modelling platform for social simulation, Proceedings of SIMUL (2014) 29–34.
[14] N. Collier, M. North, Repast HPC: A Platform for Large-Scale Agent-Based Modeling, Large-Scale Computing (2011) 81–109.
[15] A. Antelmi, G. Cordasco, G. D'Ambrosio, D. De Vinco, C. Spagnuolo, Experimenting with Agent-Based Model Simulation Tools, Applied Sciences 13 (2023).
[16] W. Bugden, A. Alahmar, Rust: The programming language for safety and performance, arXiv preprint arXiv:2206.05503 (2022).
[17] R. Jung, J.-H. Jourdan, R. Krebbers, D. Dreyer, RustBelt: Securing the foundations of the Rust programming language, Proceedings of the ACM on Programming Languages 2 (2017) 1–34.
[18] A. Chakravorty, W. S. Cleveland, P. J. Wolfe, Scalable k-d trees for distributed data, arXiv preprint arXiv:2201.08288 (2022).
[19] R. A. Brown, Building a Balanced k-d Tree in $O(kn \log n)$ Time, Journal of Computer Graphics Techniques (JCGT) 4 (2015) 50–68.
[20] C. W. Reynolds, Flocks, herds and schools: A distributed behavioral model, in: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, 1987, pp. 25–34.