The impact of ECS logic on parallel performance in agent-based model simulations

Antonio Ambrosio[1,†], Daniele De Vinco[1,*,†], Francesco Foglia[1], Carmine Spagnuolo[1] and Vittorio Scarano[1]

[1]*Università degli Studi di Salerno - Via Giovanni Paolo II, 132 - 84084 Fisciano (SA)*

Abstract

Agent-Based Modeling (ABM) is a powerful technique for simulating complex systems through the interaction of individual agents, each following simple rules. However, the computational efficiency of ABM, especially in large-scale simulations, is often limited by the sequential nature of traditional methodologies. This paper presents an innovative approach to ABM using the Entity-Component-System (ECS) methodology, which adopts a data-oriented paradigm that prioritizes the model's logic over the complexity of the simulation. ECS, a software architecture commonly used in game development, decouples data (components) from behavior (systems), allowing for more granular control of each element. By applying ECS to ABM, the system's agents are treated as entities, with their characteristics stored as components and behaviors encapsulated in systems. This modular design enables the distribution of computational tasks across multiple processors or threads, enabling fine-grained parallelism and reducing bottlenecks commonly arising in traditional ABM frameworks. Our experiments on real-world use cases demonstrate that the ECS-based ABM offers significantly higher parallel efficiency, particularly in scenarios with a large number of agents and complex interactions, while preserving ease of use and a reliable underlying structure.

Keywords

Agent-based modelling, Parallel, Simulation, ECS

1. Introduction

Agent-based models (ABMs) have been proven effective in modeling real-world phenomena [1]. This bottom-up approach breaks down complex problems into smaller individuals called agents, and from their interactions, the emergent behavior of the system is constructed [2]. This method enables modelers to explore a wide range of phenomena, such as economic models, weather forecasting, social networks, evolutionary systems, and more [3, 4]. Using such a model allows the modeler to concentrate on the system's logic rather than its inherent complexity. Additionally, the complexity of ABM is transferred from the model to the simulation architecture, which must handle diverse information and features. As a result, it is typically impractical for a single individual to develop a comprehensive structure capable of managing various types of agents and handling significant computational demands. Therefore, it is common practice not to develop these models from scratch, but rather to utilize established and reliable toolkits or frameworks [5]. Among these, agent-based modeling (ABM) engines like MASON, NetLogo, and Repast are popular choices [6, 7, 8]. However, these engines share similar architectures and concepts, which also means they inherit comparable limitations. For example, ABM simulations can be very time-consuming because of their particularly high computational load. Additionally, the execution time can significantly impact the final result, as some simulations achieve greater accuracy over time. As a consequence, a focus should be placed on how these toolkits can be made as efficient as possible. Additionally, these simulations have proven effective in "what-if" scenarios, where modelers aim to

BigHPC2025: Special Track on Big Data and High-Performance Computing, co-located with the 4th Italian Conference on Big Data and Data Science, ITADATA2025, September 9 – 11, 2025, Turin, Italy.

*Corresponding author.

†These authors contributed equally.

a.ambrosio@studenti.unisa.it (A. Ambrosio); ddevinco@unisa.it (D. D. Vinco); f.foglia9@studenti.unisa.it (F. Foglia); cspagnuolo@unisa.it (C. Spagnuolo); vitsca@unisa.it (V. Scarano)

0000-0003-0781-3744 (D. D. Vinco); 0000-0002-8267-9808 (C. Spagnuolo); 0000-0001-8437-5253 (V. Scarano)

explore a vast configuration space to replicate real-world phenomena. Consequently, the efficiency of each simulation is crucial to explore this space within a reasonable time frame.

In this work, we present an experimentation of ECS (Entity-Component-System) within krABMaga [9], a discrete-event simulation engine written in Rust capable of handling large simulations. This modular framework enables modelers to easily implement models across different computational paradigms (sequential, parallel, and distributed) with minimal effort required to switch between them. Moreover, the krABMaga engine includes a visualization module built using Bevy, a game engine entirely developed in Rust. Our proposed variant introduces an architecture where the core module has been restructured around Bevy's ECS model. The ECS architecture is a software design pattern widely used in game development. Still, its flexibility and efficiency make it suitable for other computational domains like simulations, including ABM. ECS separates the data (components) from the logic (systems) and organizes them into entities, providing a highly modular and scalable framework for managing complex, dynamic systems. An example of this architecture is shown in Figure 1.

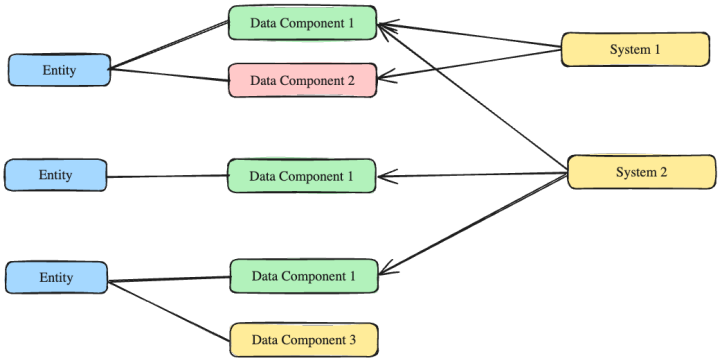


Figure 1: Entities are designed to be independent and point to the relevant data. The data associated with an entity is stored within components, while systems are responsible for processing and updating the components.

In agent-based modeling, agents can be represented as entities, with attributes such as wealth or location defined as components. This analogy creates a natural link between ABM and the ECS model, merging the benefits of both worlds, such as performance and usability, without increasing complexity. Breaking down agents into individual components in ECS allows for better reusability and fine-tuned control over each simulation aspect. Systems, such as a movement system, may update agent positions based on velocity components. ECS systems can also process entities in batches, enabling high-performance execution by applying the same operations to multiple entities within a system.

In ABM, large scale refers not only to simulations with many agents but also to the challenge of managing the complexity of such simulations [10]. Parry identified several methods to address the challenges of implementing large-scale ABM. These techniques range from adjusting the level of agent complexity, investing in new hardware, restructuring the software, or rewriting the program using a parallel approach. Each method has pros and cons, but the parallel approach appears to be the most efficient. However, developing new parallel algorithms and accurately defining the communication layer can be challenging and time-consuming.

To the best of our knowledge, the ECS model has never been employed to implement an ABM engine. We have experimented with ECS to develop the visualization module, but it was not involved in the management of the entire simulation. For this reason, we aim to fill the gap in the literature investigating the feasibility of applying such a model in the context of ABM. Therefore, the research questions we aim to address in this paper are:

- Is the ECS architecture feasible for use in agent-based modeling? Does it increase or decrease the difficulty of the development cycle?
- Does the ECS architecture show an increase in performance?
- How does the parallelized ECS model perform in agent-based simulations under high loads?

Furthermore, we empirically evaluate the ECS model's parallelization, examining its scalability and performance under high computational loads, testing the application on two well-known agent-based models.

2. Background

ABM. Over the past decade, the research community has developed numerous software frameworks and tools to support the creation of agent-based modeling simulations. These tools are often available as software libraries for various programming languages and may be optimized for specific computational platforms or tailored to particular application domains [12]. Most ABM simulation engines are designed to ensure ease of use, optimize performance, or strike a balance between the two, depending on the modeler's requirements and the computational demands of the simulation model [12]. It is important to note that high-performance computing solutions are not just suited for large-scale or fine-grained ABMs. They are also highly beneficial for small-scale ABMs that require substantial computational resources [13]. This is particularly relevant when small-scale models go through complex phases such as calibration, verification, validation, sensitivity analysis, and experimentation. Additionally, many simulations require numerous Monte Carlo repetitions, a type of simulation that relies on repeated random sampling and statistical analysis to compute results. This approach is particularly useful for experiments where the specific outcome is unpredictable in advance, as it allows for statistical insights into a range of possible outcomes. By simulating scenarios with varying inputs, Monte Carlo methods provide a robust framework for understanding probabilistic systems and are widely used in fields where uncertainty plays a key role [14]. Monte Carlo experiments can significantly benefit from horizontal scaling. In such cases, high-performance computing resources can accelerate these massive Monte Carlo runs [15]. In situations where vertical scaling is necessary, reducing the model's execution time becomes the only viable way to speed up the simulation process. As a result, efficiently managing computationally intensive models—whether large or small—and conducting long-running, reliable simulations is becoming increasingly critical for the field.

Parallel computation. One effective technique to boost computational performance is to redesign the architecture or model to support parallel optimization. Parallel computation is a powerful method in which multiple calculations or processes are executed simultaneously across multiple processing units, enabling complex problems to be solved more quickly than would be feasible with a single processor [16]. This approach is based on the idea that many problems can be decomposed into smaller, independent tasks that can run concurrently, allowing processors to collaborate in parallel to reach a solution. This structure aligns well with the design of ABMs, which often benefit from parallel processing due to the independent nature of agents' actions and interactions. Moreover, the significance of parallel computation has increased with the advent of multi-core processors and distributed computing systems, particularly in fields that require substantial computational power, such as scientific simulations, data analysis, and machine learning [17]. The capacity to handle more complex calculations efficiently has made parallel computation an invaluable asset in a wide array of applications, from research to industry.

ECS. Implementing complex applications, such as ABMs, using traditional object-oriented programming (OOP) architectures can often be challenging. Large class hierarchies, complex inheritance structures, and the addition of new entity types can quickly lead to a large codebase. As a result, code flexibility and reusability decrease, and performance can suffer. The encapsulation of data inherent to OOP can also present challenges when aiming for high performance, particularly when dealing with large numbers of entities. This encapsulation can make it more difficult to optimize memory usage and data access patterns, both of which are essential for efficiency in large-scale applications [18]. ECS architecture can provide a complementary framework for organizing ABMs in a parallel context. The clear distinction between logic and data is the core of the ECS architecture. This separation is achieved by defining an object as an entity, uniquely identified and associated with multiple dynamic components representing its data. This data is stored in memory to optimize cache usage and can be added, removed, or iterated during program execution. Iteration is facilitated through systems, which allow the selection

of the specific data type to process for each entity. More specifically, systems operate on a set of components known as archetypes. This approach enables the development of reusable, extendable, and high-performance code, focusing only on the relevant data for a particular operation while maximizing cache locality. Currently, several frameworks offer efficient implementations of this architecture. Unity and Unreal Engine, two popular game development engines, provide an ECS framework, Unity DOTS [1] and Apparatus [2]. Other notable examples are entt [3] and Flecs [4]. Although our choice has fallen on Bevy ECS [5] to speed up the development process, there are four other major ECS implementations for the Rust programming language (Specs [6], Legion [7], hecs [8], Shipyard [9]). Bevy ECS is the ECS implementation used in the Bevy Engine, a game development engine. Bevy ECS was developed to provide a simpler alternative to other implementations, which often required advanced Rust concepts like lifetimes, macros, or the builder pattern to construct ECS elements. One of the most intriguing features offered by Bevy is the ability to run applications directly in the browser with minimal performance loss. The web platform leverages this capability to enable the execution and visualization of various simulations created with krABMaga directly in the browser. This is made possible by integrating WebAssembly(WASM) and WebGL(Web Graphics Library).

3. Methodology

In most agent-based modeling (ABM) toolkits, simulations usually follow a structured workflow divided into three phases: pre-step, step, and post-step. Each iteration corresponds to a discrete time step, with the step phase being essential and the others optional. The pre-step phase prepares the simulation by organizing agents, allocating resources, and scheduling entities. The step phase is the core of the simulation, executing the model's main logic and handling agent behaviors and interactions. In the post-step phase, the toolkit finalizes tasks by updating shared data, releasing unused resources, and possibly performing data logging or analysis. This phased structure ensures efficient computation and supports scalability for complex models with many agents.

Specifically in krABMaga, the simulation progresses in steps using double buffering, meaning that agents scheduled at step i cannot see any changes happening in the current step unless the modeler forces a read on the specific structure of the buffering. Instead, they only see the state of other agents, fields, and the simulation as they were at step $i - 1$. This mechanism's clear separation of read and write structures allows the engine to perform better on several tasks.

In sequential execution, as the paradigm suggests, each stage of the process must wait for the previous one to complete, even when it is unnecessary. Moreover, the simulation's logic, encapsulated in the step phase, can be executed independently for each agent. This naturally leads to a scenario where parallel execution is both possible and highly beneficial. However, the original architecture of krABMaga, influenced by inherent features of the Rust programming language, limits the ease of writing parallel code due to its strict concepts of ownership and borrowing [19]. These constraints make it challenging to implement parallel execution on the original architecture without significant restructuring.

By contrast, modeling an agent-based model using an ECS architecture offers a more suitable approach. The main building blocks of an ECS are described as follows:

- *Entities*: An entity is essentially a unique identifier that serves as a container for various components. Unlike objects in object-oriented programming, entities do not hold data or logic themselves; they are purely identifiers. This makes entities lightweight and flexible.

[1] https://unity.com/dots
[2] https://www.unrealengine.com/marketplace/en-US/product/apparatus
[3] https://github.com/skypjack/entt
[4] https://github.com/SanderMertens/flecs
[5] https://github.com/bevyengine
[6] https://github.com/amethyst/specs
[7] https://github.com/amethyst/legion
[8] https://github.com/Ralith/hecs
[9] https://github.com/leudz/shipyard

- *Components*: Components are the pieces of data that define the characteristics or state of an entity. Each component typically holds a single aspect of data, such as position, velocity, or health.
- *Systems*: Systems contain the logic that operates on entities by modifying their components. They are responsible for processing data in a component-centric manner.

As mentioned earlier, ECS decouples data from behavior, facilitating the independent processing of agents and making the transition to parallel code much more straightforward. This decoupling enables better scalability and performance, particularly in large-scale simulations where parallel execution can significantly reduce computational time. The ECS design inherently aligns with parallelism by enabling simultaneous updates to entities without direct dependencies, making it an ideal solution for ABM simulations that require high performance.

4. Evaluation Strategy

The original version of KrABMaga leverages Bevy as its visualization module. Bevy is a straightforward, data-driven game engine that lets users take advantage of the ECS architecture within the Rust ecosystem. By default, Bevy supports highly parallel and cache-efficient programming. It offers various features and libraries for making queries, managing global and local resources, and implementing a lock-free parallel scheduler.

To assess our framework, we adapted two widely used examples from the ABM field: the Boids simulation [20], which mimics bird flocking behavior, and WolfSheepGrass [21], a predator-prey model that balances the interactions among three different populations, following Lotka-Volterra equations [22]. These examples are valuable for evaluating the engine's performance with single and multiple agent types. To better describe our efforts and improvements, we will provide insights into the development of the WolfSheepGrass implementation.

First experiment. The initial implementation forcefully imposed parallelism on agent data structures using mutable queries for sheep and wolves. When a wolf detects and eats a sheep, it gains energy; the sheep must be removed, and the field updated to prevent other wolves from eating the same dead sheep. Parallel implementation proved challenging due to the overhead of mutually exclusive field updates.

This experiment showed that the focus should be on tracking sheep deaths and wolf energy gains per location. To overcome this, we implemented atomic counters tracking wolves per grid cell. The system processed sheep and grid in parallel, despawning appropriate sheep from each cell. Using wolf counts, sheep fields, and wolf locations, we calculated energy gains by computing the minimum between sheep and wolves per cell. This approach failed to improve performance because atomic operations introduced locks during execution, creating bottlenecks that did not resolve field update issues.

Second experiment. The constraint that agents only interact within the same location was intended to isolate potential conflicts. Wolves kill sheep and gain energy in the same grid cell. The challenge was modifying wolves' energy values during parallel execution, as Rust's mutable objects cannot be shared across threads. To address this, we used locks, ensuring only one thread could access an agent's data at a time. Instead of locking entire query objects, locks were applied directly to agents' data, maintaining efficient parallel execution while satisfying Rust's memory safety requirements.

Third experiment. The previous experiment did not fix the grid's weak performance. Placing a lock on the entire grid would cause race conditions among agents. However, each agent only creates race conditions within its specific grid bag. Since we modify only the bag corresponding to the agent's location rather than the entire grid, the solution is to place locks on individual bags. This allows locking only the required bag, reducing thread contention and improving parallel efficiency.

Fourth experiment. Despite fixing some grid problems, agent management became a bottleneck. Spawning and despawning agents when they die or are born influenced engine performance. To optimize agent lifecycles, we replaced the spawning system with Bevy's parallel system, which creates a buffer

for the *spawn_batch* function. This allows instantiating all agents in a single call using an iterator containing initialization data, streamlining the process.

Current experiment. Previous attempts at the new implementation failed to improve performance across all critical sections of the architecture simultaneously. Each attempt addressed individual issues, but none provided a complete solution. The main bottlenecks were sequential field updates, agent spawning and despawning, and certain Bevy command executions. In the current implementation, which we refer to as the cemetery system, we aim to address all these issues concurrently, resulting in a more comprehensive and efficient approach. The cemetery system is a memory management technique designed to optimize memory reuse in simulations, especially when agents frequently die or need respawning. In traditional approaches, when an agent dies or is removed from the simulation, the associated memory is typically deallocated and must be reallocated later when a new agent spawns. Since this allocation-deallocation cycle repeats each iteration, it introduces performance overhead, especially in large-scale simulations where many agents are created and destroyed. In contrast, the cemetery system avoids this overhead by maintaining a pool of freed but unallocated memory locations when agents die. Instead of removing the memory entirely, the system places the memory of dead agents into a designated region called the cemetery. This region serves as a holding area, preserving space for future use. When new agents need to be created, the system reuses these empty cemetery spaces, assigning them to agents that require spawning. By reusing already-allocated memory, the cemetery system reduces frequent memory allocations, which are expensive in both time and system resources. The cemetery system operates within a single iteration window. Agent memory need not persist across the entire simulation, only during the current step. This approach enhances performance more granularly, avoiding overhead associated with Bevy's internal functions, such as querying and reordering, which occur when managing entities.

5. Empirical Results

To answer our research questions, we have tested Boids and WolfSheepGrass with the last architecture shown in the previous section (ECS with a cemetery system). The code for each attempt and benchmark is available as a GitHub repository [10]. As a baseline, we measured the execution times of these models using the original sequential krABMaga implementation. Starting from 32000 agents, we have doubled this number to 2048000. The field size was selected to maintain an agent density of approximately 10%. Moreover, each configuration is tested on different numbers of threads, starting from 1 to 20, with an average elapsed time of 5 runs. The machine is configured as follows:

- OS: Ubuntu 24.04.1 LTS x86_64
- CPU: Intel i9-14900KF @ 5.800GH (24 Cores, of which Performance-cores 8, Efficient-cores 16)
- Memory: 64 GB

Table 1 compares the performance of ECS-based models against their original implementations. The values in the tables reflect the execution times of the models in their sequential versions, with the ECS variant constrained to a single thread.

Interestingly, the multi-agent model (WSG) shows improved performance compared to its original counterpart, whereas the simpler Boids model exhibits slower execution times. This seemingly counterintuitive result can be attributed to the internal structure of the Boids model, which does not fully leverage the advantages offered by the ECS framework. As a result, the model has additional overhead throughout the computation, leading to reduced efficiency. On the other hand, WSG appears to benefit more from ECS optimizations, even in its single-threaded form.

This section specifically describes on the results of the WSG model in its final version. Table 2 presents the execution times of the WSG model, running with 2,048,000 agents, as the number of threads varies. Table 3 describes the execution times of the Boids model. To provide deeper insights,

[10] https://github.com/Tonaion02/KrABMagaPersonalDev

Table 1
Sequential krABMaga execution time vs the ECS implementation run with 1 thread (time in seconds).

# agents	Field	WSG (ECS)	WSG (seq)	Boids (ECS)	Boids (seq)
32000	565x565	3,50	5,61	4,32	2,18
64000	800x800	7,64	13,73	9,57	4,69
128000	1131x1131	16,29	30,40	20,90	9,92
256000	1600x1600	33,49	61,52	47,12	22,80
512000	2262x2262	69,07	112,54	130,17	63,78
1024000	3200x3200	141,52	238,70	378,11	162,79
2048000	4525x4525	287,79	487,66	966,40	394,37

Table 2
Relative speedup of WSG model on 2048000 agents (time in seconds).

# threads	Total time	Speedup	Step time	Speedup
1	287,79	1	0,52	1
2	161,09	1,78	0,28	1,84
4	98,72	2,91	0,15	3,42
8	64,55	4,45	0,08	5,97
16	56,43	5,09	0,07	6,85
20	54,54	5,27	0,07	7,02

Table 3
Relative speedup of Boids model on 2048000 agents (time in seconds).

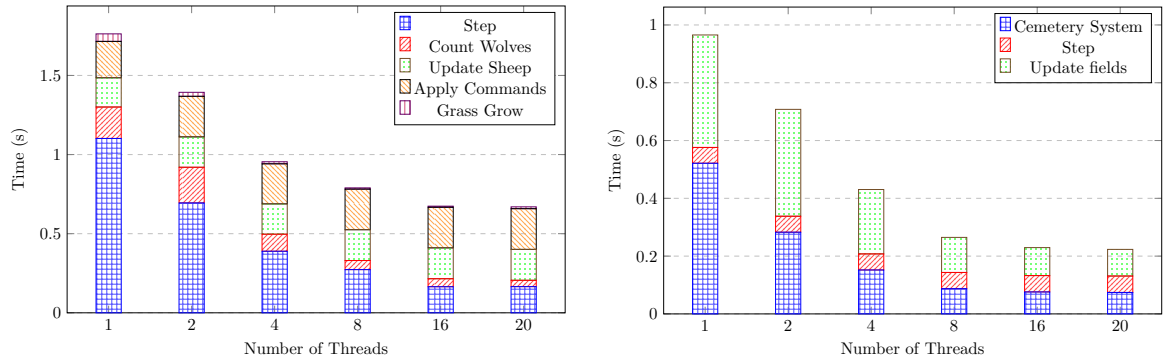
# threads	Total time	Speedup	Step time	Speedup
1	966,40	1	4,51	1
2	486,25	1,99	2,27	1,98
4	249,81	3,86	1,16	3,87
8	142,76	6,76	0,66	6,78
16	96,24	10,04	0,44	10,24
20	84,89	11,38	0,38	11,70

the table includes a "step" column, which highlights the performance improvements during the most computationally intensive tasks.

As discussed in the methodology section, the primary bottleneck in the simulation arises from updating the internal structure of the system. Up to 8 threads, there is a noticeable and consistent increase in speedup, reflecting efficient parallelization. However, beyond this point, the performance reaches a plateau. This plateau is due to the communication and synchronization overhead between threads, which becomes more pronounced as the number of threads increases. At this stage, the benefits of additional parallelism are offset by the cost of managing inter-thread dependencies and coordination.

Figure 2a illustrates a snapshot of the initial implementation of the ECS methodology, as described in the previous section. The computation is divided into distinct phases, each occupying separate slots within a single iteration of the model's execution. Notably, some of these phases overlap, meaning they must be executed sequentially, which extends the overall time required to complete each iteration. This segmentation reduces the potential for parallelism and introduces inefficiencies, as certain tasks must wait for the completion of others before proceeding.

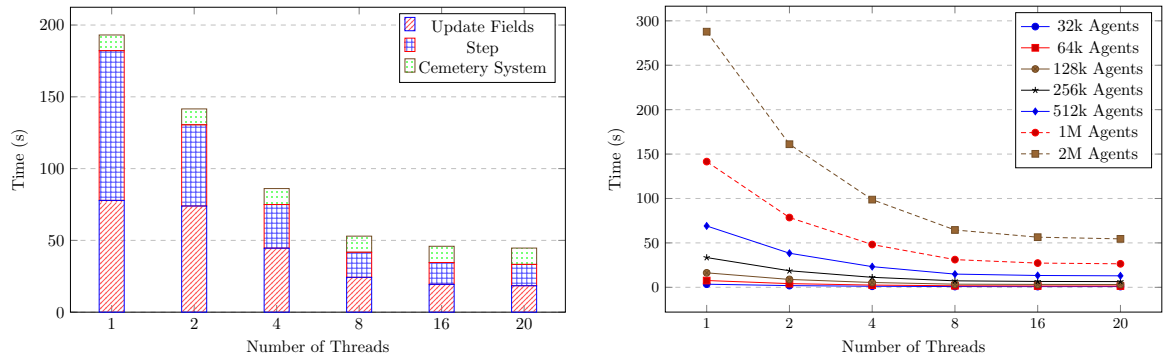
Figure 2b shows the execution time of the WSG model in its final iteration, incorporating the cemetery system. In this implementation, entities are allocated at the start of the simulation. When an agent dies, instead of being deallocated, it leaves behind an empty space in the "cemetery," which can be reused for future agents in subsequent steps. This approach eliminates the overhead of dynamic memory deallocation and reallocation during runtime. As a result, the previously time-consuming



(a) This snapshot captures a single iteration of the first implementation of ECS architecture.

(b) This snapshot shows a single iteration from the final implementation of the ECS architecture.

Figure 2: A look to the performance of the WSG model along different implementations on 2048000 agents varying number of threads.



(a) Stacked values of the components in the last experiment.

(b) Total execution times of each simulation size varying the number of threads.

Figure 3: A comparison of the WSG model's performance across different implementations on 2048000 agents, varying the number of threads.

"apply commands" phase, which handled entity management, is now optimized and no longer consumes significant processing time. This improvement allows the system to more effectively parallelize operations, enabling simultaneous execution of different components of the simulation. The cemetery system thus contributes to a more efficient overall workflow, reducing bottlenecks and improving parallel performance.

Figure 3b shows the total execution time of the final implementation of the WSG model. The cemetery system is stable, reaching a constant value across different thread counts, showing that it no longer significantly influences the overall performance. The computation time for each step scales well with the increasing number of threads, although performance plateaus after reaching 8 threads. This suggests that the simulation benefits from parallelization up to a certain point, beyond which the overhead of synchronization and communication between threads becomes a bottleneck.

While the update fields still occupy a portion of the total execution time, the introduction of parallelism allows various field management tasks to be handled concurrently. As a result, the overall impact of the field on the simulation's total runtime is reduced.

Additionally, Figure 3b highlights the performance of the WSG model across various simulation sizes and thread counts. Notably, the performance plateau discussed earlier is not influenced by the size of the field or the number of agents. Instead, it is a direct consequence of the inherent limitations of the ECS architecture. This suggests that while the ECS framework provides certain advantages, its current implementation reaches a point where additional parallelism no longer yields significant performance

improvements. Given this observation, future research should focus on exploring alternative methods for managing the different phases or structures of the simulation. By optimizing the organization and execution of the simulation's various tasks, it may be possible to overcome the current performance bottleneck and achieve further gains in efficiency.

5.1. Limitations

However, factors such as load balancing, synchronization costs, and the nature of agent interactions influence the level of parallel efficiency. In tightly coupled simulations, where agent interactions are frequent and complex, the overhead from synchronization offsets the gains from parallelization. This is highlighted by the simulation's plateau of performance, as shown in the previous chapter.

The framework was designed as a versatile tool to support a wide range of applications. However, this broad compatibility may result in certain components being less efficient due to the need for diverse use cases. To optimize performance and properly evaluate the benefits of this approach, additional workloads should be tested, and a code profiling should be conducted to fix issues such as caching or to evaluate the impact of tightly vs loosely coupled agents.

It is crucial to recognize that redesigning an engine for parallel architecture presents significant challenges, and the associated development costs must be carefully weighed when determining the feasibility of this solution.

6. Conclusion and Future Directions

This paper explored the parallel efficiency of an ABM engine built entirely on the ECS logic. The ECS architecture enables a clear separation of its elements by decoupling entities, components, and systems, thereby enhancing flexibility and modularity. Through parallelization, we aimed to improve the scalability and performance of ABM simulations, which are often computationally expensive due to the complex interactions between agents. Our results demonstrate that ECS provides a suitable framework for parallelizing ABM simulations, especially in scenarios with a large number of agents and interactions.

While this study has provided insights into the benefits and limitations of parallelization in ECS-based ABM engines, there are several areas for future exploration:

- Investigating the use of heterogeneous computing resources, such as GPU and FPGA acceleration, in conjunction with ECS logic could offer further performance gains. Understanding how to map ECS operations to these architectures will be a key challenge.
- ECS architectures may incur overhead in memory usage due to the separation of components and systems. Future work could investigate memory optimization techniques to reduce the footprint of ECS-based ABM engines.
- Expanding the benchmarking suite to include a wider variety of ABM scenarios, such as those with varying interaction complexity, agent heterogeneity, and environmental dynamics.
- Exploring hybrid architectures that combine ECS with other modeling paradigms, such as standard discrete-event simulation, could lead to more efficient engines that leverage the strengths of each approach.

Future research can address these challenges to further optimize ECS-based ABM engines, enabling them to handle increasingly complex simulations while maximizing efficiency.

Declaration on Generative AI

During the preparation of this work, the author(s) used Generative AI for grammar and spelling checks. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] A. F. Siegenfeld, Y. Bar-Yam, An Introduction to Complex Systems Science and Its Applications, Complexity 2020 (2020) 6105872.

[2] P. W. Anderson, More Is Different, Science 177 (1972).

[3] T.-S. Yun, D. Kim, I.-C. Moon, J. W. Bae, Agent-Based Model for Urban Administration: A Case Study of Bridge Construction and its Traffic Dispersion Effect, Journal of Artificial Societies and Social Simulation 25 (2022) 5.

[4] B. Alves Furtado, PolicySpace2: Modeling Markets and Endogenous Public Policies, Journal of Artificial Societies and Social Simulation 25 (2022) 8.

[5] S. Abar, G. K. Theodoropoulos, P. Lemarinier, G. M. O'Hare, Agent Based Modelling and Simulation tools: A review of the state-of-art software, Computer Science Review 24 (2017) 13–33.

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

[7] S. Tisue, U. Wilensky, Netlogo: Design and implementation of a multi-agent modeling environment, in: Proceedings of agent, volume 2004, 2004, pp. 7–9.

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

[9] 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).

[10] H. R. Parry, M. Bithell, Large scale agent-based modelling: A review and guidelines for model scaling, Agent-based models of geographical systems (2011) 271–308.

[11] H. R. Parry, Agent-based modeling, large-scale simulations, Complex Social and Behavioral Systems: Game Theory and Agent-Based Models (2020) 913–926.

[12] A. Antelmi, G. Cordasco, G. D'Ambrosio, D. De Vinco, C. Spagnuolo, Experimenting with Agent-Based Model Simulation Tools, Applied Sciences 13 (2023).

[13] L. An, V. Grimm, A. Sullivan, B. T. II, N. Malleson, A. Heppenstall, C. Vincenot, D. Robinson, X. Ye, J. Liu, E. Lindkvist, W. Tang, Challenges, tasks, and opportunities in modeling agent-based complex systems, Ecological Modelling 457 (2021) 109685.

[14] S. Raychaudhuri, Introduction to monte carlo simulation, in: 2008 Winter simulation conference, IEEE, 2008, pp. 91–100.

[15] W. Tang, D. Bennett, The Explicit Representation of Context in Agent-Based Models of Complex Adaptive Spatial Systems, Annals of the Association of American Geographers 100 (2010) 1128–1155.

[16] D. B. Skillicorn, D. Talia, Models and languages for parallel computation, Acm Computing Surveys (Csur) 30 (1998) 123–169.

[17] W. Tang, S. Wang, Hpabm: A hierarchical parallel simulation framework for spatially-explicit agent-based models, Transactions in GIS 13 (2009) 315–333.

[18] D. Wiebusch, M. E. Latoschik, Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems, in: 2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), IEEE, 2015, pp. 25–32.

[19] A. Antelmi, G. Cordasco, M. D'Auria, D. De Vinco, A. Negro, C. Spagnuolo, On evaluating rust as a programming language for the future of massive agent-based simulations, in: Methods and Applications for Modeling and Simulation of Complex Systems: 19th Asia Simulation Conference, AsiaSim 2019, Singapore, October 30–November 1, 2019, Proceedings 19, Springer Singapore, 2019, pp. 15–28.

[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.

[21] J. Chattopadhyay, O. Arino, A predator-prey model with disease in the prey, Nonlinear analysis 36 (1999) 747–766.

[22] A. A. Berryman, The orgins and evolution of predator-prey theory, Ecology 73 (1992) 1530–1535.