Efficient Agent-Based Simulations Using the Sender Asynchronous Programming Model

Pablo A.S. Hugen[1], **Guilherme Galante**[1]

[1] Colegiado de Ciência da Computação
Universidade Estadual do Oeste do Paraná (UNIOESTE)
85819-110 – Cascavel – PR – Brazil

pabloashugen@protonmail.com, guilherme.galante@unioeste.br

Abstract. *This paper develops and evaluates an Agent-Based Model (ABM) simulator for disease transmission using the* C++ Sender Programming Model. *The asynchronous approach with the Sender Model is, on average, 2.61 times faster than synchronous methods, enhancing performance while preserving cross-platform compatibility. This research offers a valuable alternative for simulating complex epidemiological scenarios, advancing computational epidemiology by optimizing both performance and portability. Future work will focus on improving memory management and validating the model across different hardware configurations and population densities.*

1. Introduction

The transmission of infectious diseases, such as the *SARS-CoV-2* and *H1N1* outbreaks, has been notably accelerated by global interconnectedness, underscoring the vital role of computational technologies in public health management. Consequently, mathematical models and simulations, as demonstrated by [Rachah and Silva 2024], have become crucial for developing effective health strategies. Additionally, the need for simulating complex epidemiological scenarios and advancements in High Performance Computing (HPC) has driven the adoption of Agent-Based Models (*ABMs*) in computational epidemiology, as discussed by [Elsheikh 2024]. These models provide insights into epidemic dynamics by simulating disease progression at an individual level, highlighting the spatial and temporal interactions between agents and their environment [Cunha et al. 2022]. However, the implementation of *Agent Based Simulations* presents challenges, particularly due to the significant computational resources required in terms of memory storage and processing time, which can become a limiting factor as the number of individuals or spatial resolution increases [Rosenstrom et al. 2024]. Another issue when implementing those types of simulations is portability, as the particularities of each parallel programming model, such as *CUDA*, can hinder the reuse of code and narrow the range of hardwares that can be used to run the simulations.

Usually, *Agent Based simulations* are *GPU* accelerated [Kitson et al. 2024, Thomopoulos and Tsichlas 2024], often utilizing the *CUDA* programming model. That approach, despite often providing the best performance, has the drawback of having limited portability, as *CUDA* is a proprietary technology that only works on *NVIDIA* hardware, causing a lock-in effect in the code and a shortage of libraries and resources available for the development of the simulations. Moreover, moved by the prominence of the *c++ programming language* in various areas of HPC, recent research has shown the

effectiveness of *ISO C++ parallel algorithms* in enhancing computational performance across various scenarios by leveraging *GPUs* and other accelerators to provide a standardized, cross-platform approach to parallelism [Brown et al. 2019]. Further research sits on developing a structured parallelism approach in *C++* called the *Sender Model*, which aims to provide portable and high-level abstractions for asyncronous and hetereogeneous parallelism in the language [Dominiak et al. 2024].

In light of this, this work aims to develop a parallel implementation of an *Agent-Based Model* (ABM) simulation in *C++* using the *Sender Model* for parallelism and to evaluate its computational efficiency. The authors seek to contribute to the literature by offering a portable and efficient alternative to traditional ABM simulation implementations. Also, this study contributes to the HPC literature by demonstrating the application and feasibility of the experimental Sender Model in parallel computing environments. The implementation demonstrated positive results, with the asynchronous and heterogeneous nature of the Sender Model enabling concurrent task execution on either the *CPU* or *GPU*, depending on the task's nature. Experimental evaluations showed that the asynchronous approach is significantly faster than the synchronous one and that assigning critical routines to the appropriate device can greatly enhance simulation performance.

That said, this paper is organized as follows: section 2 presents some recent examples of *ABM* implementations, highlighting the implementatios decisions, technologies used and key results; section 3 discusses the particularities of the model; section 4 examines the parallel programming model used, with subsection 4.1 discussing the implementation details; section 5 presents the results of the computational evaluation of the simulation; Finally, section 6 presents the conclusions of the work, discussing the main contributions and future works.

2. Related Work

The literature on *Agent-Based Models* is vast and diverse, encompassing numerous approaches and implementations. Therefore, this revision specifically focuses on implementations using the C++ programming language or its variants.

In [Cunha et al. 2022], an agent-based simulation system for dengue propagation is developed, employing a multi-agent compartmental SEIRS model. In the implementation of the proposed model, the author utilized the CUDA programming model and the *Thrust* parallel algorithms library. However, although Cunha's study does not focus on the use of high-performance computing techniques to improve computational efficiency in computational epidemiology, it has shown excellent results in approximating the dengue cases in the city of Cascavel-PR, Brazil over a one-year period.

In the work [Gallagher et al. 2024], the authors present a COVID-19 *ABM* simulation using the *C++* programming language and its standard threads library. Despite being designed as an educational tool, it has been able to simulate 5 million agents in New Zealand on a personal workstation in roughly 8 hours. The authors also discuss the limitations of the implementation, such as the compromise between performance and code readability.

Also, [Kühn et al. 2024] introduces a versatile software platform that implements various simulation models, including a COVID-19 model applied to Germany. The platform leverages OpenMP as the parallel programming model to enhance performance.

However, the authors report several issues when running the software on platforms other than Linux, with the most significant problems being related to build and dependency management.

So based on this, the present research differentiate itself from previous studies by offering a structured, asynchronous, and heterogeneous approach to parallelism applied to *ABM* simulations, providing a portable and efficient alternative to traditional implementations.

3. Agent-Based and Compartmental Modeling

In the simulator, each individual agent is characterized by attributes including age, location, and health status, within a spatial environment modeled as a mobility graph, as shown by Figure 1. Agents interact within the surroundings through a series of time steps or cycles, enabling movement within the environment, contact with other agents, and also transitions in health status. These status changes adhere to the *SEIRS* model (as illustrated in Figure 2), cycling agents through predetermined health states: Susceptible, Exposed, Infected, and Recovered; thus dynamically simulating the spread and impact of diseases over time. This conceptual disease spreading model is based on the work by [Cunha et al. 2022], incorporating the Monte Carlo method to account for the stochastic nature of disease spreading, running multiple simulations with random parameters within a predetermined range. Specifically, the model is adapted for *Dengue* fever, incorporating both human and mosquito agents in the simulation.

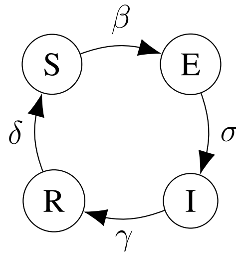


Figure 1. SEIRS Model

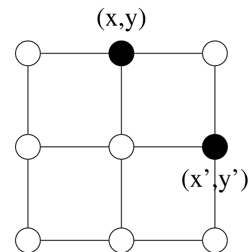


Figure 2. Mobility Graph

The simulation process process is conceptually simple. Before the execution, the environment and parameters are parsed and initialized. The environment is represented as a graph, with nodes representing the mobility points and edges representing the connections between them. The parameters are read from a configuration file, which specifies for each probability action (such as infection, recovery, and death) the respective minimum and maximum values. After this, each simulation is composed of the routines: *insertion*, *movement*, *contact*, and *transition*. The *insertion* routine initializes the agents in the environment, the *movement* routine updates the agents' positions, the *contact* routine simulates the interaction between agents, and the *transition* routine changes the agents' health status according to the *SEIRS* model. The simulation process is summarized in Algorithm 1.

Algorithm 1: Simulation Process

Input: Simulation parameters, Environment configuration
Output: Simulation results

```
1 Function Main():
2     parse_parameters();
3     parse_environment();
4     insertion();
5     for i ← 0 to num_cycles do
6         movement();
7         contact();
8         transition();
9     end
```

4. Parallelizing the simulator using the Sender Model

The maturity of *C++* in the *HPC* ecosystem has led to high-level parallel programming models directly based on the language, reducing the time and effort needed for maintaining and deploying applications while achieving portability and performance with a single codebase [Deakin and McIntosh-Smith 2020]. These models aim to enable heterogeneous parallel programming with near-native performance [Breyer et al. 2022].

The *ISO/IEC 14882:2017* standard (*C++17*) introduced synchronous (blocking) parallel algorithms in the *C++* standard library, supporting multi-core *CPUs* and *GPUs* with competitive performance [Lin et al. 2022]. This model is inherently synchronous, blocking program execution until parallel tasks are complete, limiting concurrent parallelism [Lin et al. 2022]. To address this, efforts are focused on developing the asynchronous (non-blocking) parallel programming model **std::execution** for the upcoming *C++26* standard, facilitating flexible and efficient asynchronous task execution using abstractions like *senders*, *receivers*, and *schedulers* [Dominiak et al. 2024]. Thus, the simulator is implemented with a combination of the *Sender Model* and the *C++17* parallel algorithms, using an hetereogeneous approach for executing different tasks on the *CPU* and *GPUs*. For example, the Code 1 shows how the insertion routine could be implemented[1] using the *Sender Model*.

4.1. Implementation Details

For the sake of brevity, this section will focus on the most relevant aspects of the implementation, for the complete code is fully open-source and available at `https://github.com/HpcResearchLaboratory/simulator`. It is implemented using the *NVIDIA HPC SDK* [NVIDIA 2024], which provides modern *ISO C++ Parallel Algorithms* and the experimental candidate for standardization `https://github.com/NVIDIA/stdexec` library.

As shown in section 3, the simulation process is divided into four main routines: *insertion*, *movement*, *contact*, and *transition*. The first one, *insertion*, executes in parallel for every agent type, basically assigning a random position for the agent and setting the struct into the unified memory. Next, *movement* also runs in parallel based on each agent,

[1] a

```
const auto insert_susceptible_human = [...](auto i) noexcept {...};
...
const auto insert_susceptible_mosquito = [...](auto i) noexcept {...};
...
const auto work = stdexec::when_all(
    stdexec::just() | exec::on(gpu.get_scheduler(),
                          stdexec::bulk(params->nh, insert_s_human)),
    stdexec::just() | exec::on(cpu.get_scheduler(),
                          stdexec::bulk(params->nm, insert_s_mosquito)))
stdexc::sync_wait(work);
```

Listing 1. Insertion routine with the Sender Model

choosing a random position based on the "possible paths from a position" set. Following, the *contact* routine simulates the interaction between agents, executing in parallel for every position and simulating the *human-mosquito* and *mosquito-mosquito* dynamics. Finally, the *transition* routine changes the agents' health status according to the *SEIRS* model state machine, also executing in parallel for every agent.

Complex routines, such as the movement and contact between mosquitoes—which involve many nested loops and data-dependent logical flows[2]—are now **executed on the *CPU***, as it is a more suitable device than the *GPU* for such operations. More details are discussed below in subsubsection 5.3.2. Also, the asynchronous nature of the *Sender Model* is used to execute various *kernel streams* concurrently, enabling the *GPUs* to run independent routines in parallel.

Next, the simulator was implemented with consideration for the unified memory architecture of *CUDA* and the *RAII* programming technique[3] [CPP reference 2023]. As a result, there are no small-volume memory transfers, as all memory used in the simulation is allocated and initialized in a shared memory space between all *CPUs* and *GPUs* at the time of simulation creation. This approach also eliminates the issue of asynchronous transfers, as the memory is shared and accessible by all devices, and page swaps are handled dynamically by the *CUDA* driver [Chien et al. 2019].

5. Experimental Evaluation

After the implementation of the simulator, were conducted a series of experiments to evaluate the computational feasibility of the simulator. The experiments used resources of the High-Performance Computing Park (*PCAD*), an HPC infrastructure located at the Institute of Informatics of the Federal University of Rio Grande do Sul (*INF/UFRGS*)[4]. The tests were carried out on the "beagle" computational node, with the configuration detailed in Table 1.

[2]Data-dependent logical flows occur when each branch of a decision structure (such as *if* or *case*) accesses data outside this structure.

[3]*RAII* (Resource Acquisition Is Initialization) is a programming paradigm in languages with managed memory that ties the lifetime of all resources used by an object (especially dynamically allocated memory) to the lifetime of the object itself.

[4]`http://gppd-hpc.inf.ufrgs.br`.

Table 1. Configuration of the "beagle" node.

Type	Component	Specification	Quantity
Processor	Intel Xeon E5-2650	2.0GHz, 32 threads, 16 cores	2
Memory	DDR3	32 GB	1
GPU	NVIDIA GeForce GTX 1080Ti	3584 CUDA cores	2

Two specialized tools were employed for analysis and benchmarking. *NVIDIA Nsight Systems*[5] [NVIDIA Corporation 2023] was used to analyze occupancy information, memory patterns, and kernel execution on the GPUs, providing detailed insights into performance and potential bottlenecks. For runtime benchmarking, *Hyperfine*[6] [Peter 2023] was adopted, a command-line tool that allows precise measurement of program execution time, facilitating comparisons between different software versions.

5.1. Simulation Configurations

The tests were conducted in four different environments corresponding to urban pedestrian street networks, ranging from small to very large, with the goal of evaluating the simulator's performance in different scenarios. The small environment, corresponding to the area near the State University of Western Paraná (*UNIOESTE*), has 362 mobility nodes and is illustrated in Figure 3. The medium environment, representing the southern part of Cascavel/PR, has 2,494 mobility nodes and is shown in Figure 4. The large environment, characterizing the entire city of Cascavel/PR, has 8,268 mobility nodes and is presented in Figure 5. Finally, the very large environment, representing the central region of Curitiba/PR, has 19,207 mobility nodes and is depicted in Figure 6.

For each environment size, a corresponding number of agents was used. The small simulation has 1,300 humans and 32 mosquitoes; the medium one has 13,000 humans and 3,000 mosquitoes; the large one has 130,000 humans and 120,000 mosquitoes; and the very large one has 100,000 humans and 60,000 mosquitoes. Although the very large simulation has fewer agents than the large one, this is due to GPU memory limitations in the execution environment. To effectively test the operators parallelized by location under these constraints, the authors chose to reduce the agent density in the largest environment. Additionally, each simulation was run for 10 iterations. Table 2 summarizes the configurations for each simulation.

Table 2. Simulation Configurations

Simulation	Environment Nodes	# of Humans	# of Mosquitoes
Small	362	1,300	32
Medium	2,494	13,000	3,000
Large	8,268	130,000	120,000
Very Large	19,207	100,000	60,000

[5] `https://developer.nvidia.com/nsight-systems`.
[6] `https://github.com/sharkdp/hyperfine`.

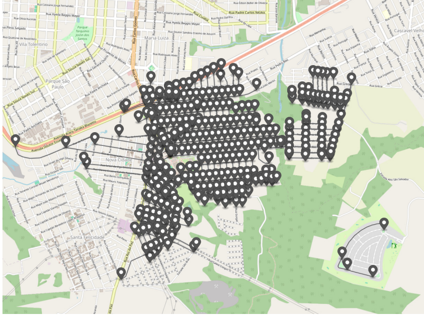


Figure 3. Small Environment

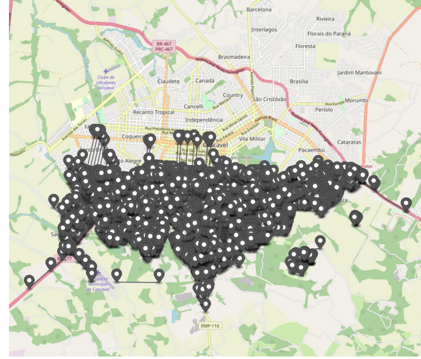


Figure 4. Medium Environment

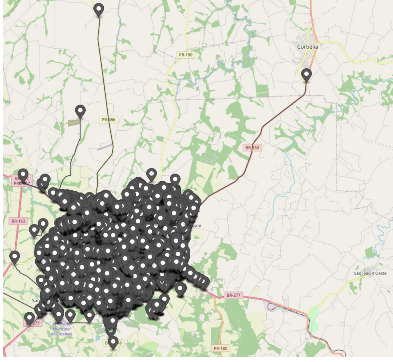


Figure 5. Large Environment

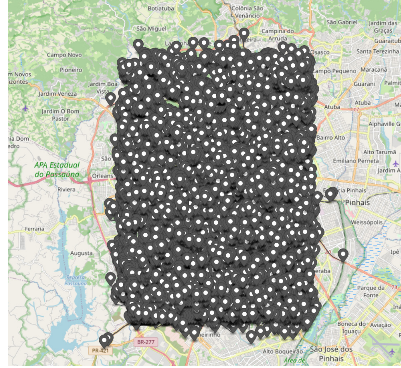


Figure 6. Larger Environment

5.2. Validation Criteria and Test Cases

The test cases for performance evaluation were planned to ensure statistical reliability. Each test scenario was executed separately ten times to obtain accurate metrics. To minimize performance variations caused by caching, each case was run three times beforehand to warm up the system before effective performance measurements were taken.

Profiling analyses were conducted in dedicated runs to avoid profiling overheads affecting the performance statistics. For collecting metrics on behavior, memory patterns, and kernel execution, the *NVIDIA Nsight Systems* tool [NVIDIA Corporation 2023] was used. Runtime metrics, on the other hand, were obtained using the *Hyperfine* software [Peter 2023].

In comparing runtime between two distinct tests, Welch's T-test was applied. Unlike the traditional T-test, it does not assume equal variances between groups, making it more appropriate when variances differ, which is relevant in the present testing context [West 2021]. The formula for calculating the t statistic is given by:

$$t = \frac{\Delta\overline{X}}{s_{\Delta\overline{X}}} = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{s^2_{\overline{X}_1} + s^2_{\overline{X}_2}}} \tag{1}$$

$$s_{\overline{X}_i} = \frac{s_i}{\sqrt{N_i}} \tag{2}$$

Where $\overline{X}_i$ and $s_{\overline{X}_i}$ represent the sample mean and standard error for the i-th group, respectively. The term s_i denotes the corrected sample standard deviation, and N_i indicates the sample size of each group. Thus, given the associated t value, the p value is calculated to determine statistical significance. If the p value is less than the significance threshold, the null hypothesis is rejected, indicating that the group means are significantly different. Otherwise, the null hypothesis is not rejected, suggesting that the data are not sufficiently persuasive to prefer the alternative hypothesis. For this work, the significance threshold was set at 0.05 (95% confidence).

5.2.1. Test Case 1

In the first test, the execution time was evaluated using the synchronous and asynchronous approaches in the very large simulation configuration. The objective here is to assess the impact of transitions between different execution approaches. For this, two hypotheses (null and alternative) were defined:

- H_0: There is no significant difference in execution times between the synchronous and asynchronous approaches.
- H_1: There is a significant difference in execution times between the synchronous and asynchronous approaches.

5.2.2. Test Case 2

In the second test case, using the very large simulation configuration, the execution time was evaluated when each of the four main routines (*insertion, movement, contact, and transition*) was switched to execution on the *CPU*. The objective is to assess the impact of moving critical routines to the CPU on the simulator's performance. For this, for each routine, two hypotheses (null and alternative) were defined:

- H_0: There is no significant difference in execution times when the routine is executed on the *CPU* compared to the standard execution.
- H_1: There is a significant difference in execution times when the routine is executed on the *CPU* compared to the standard execution.

5.3. Test Case Execution and Results

Hence, the test cases were executed in the "beagle" node, and the results and discussions are presented in the following sections.

5.3.1. Test Case 1

Test Case 1 revealed that the asynchronous approach significantly outperforms the synchronous approach in terms of speed, being on average 2.61 times faster than its alternative. Table 3 details the measured execution times, while Figure 7 visually illustrates the differences in execution times between the two approaches. Additionally, the asynchronous approach showed significant variation in execution times, with a standard deviation of 16.285 seconds, while the synchronous approach had a standard deviation of

0.809 seconds. This suggests that the asynchronous approach is more sensitive to workload variations.

Table 3. Test Case 1 Results

Approach	Mean [s]	Min [s]	Max [s]	Ratio
Synchronous	200.428 ± 0.809	199.263	201.444	2.61 ± 0.55
Asynchronous	76.836 ± 16.285	48.741	97.477	1.00

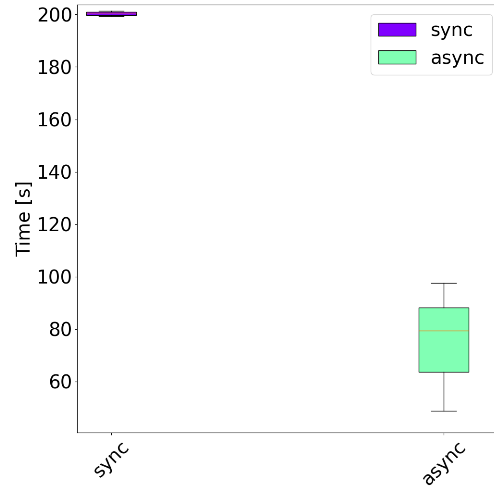


Figure 7. Test Case 1 Comparison

The t-test yielded a t value of 24.0 and a p value of 1.7×10^{-9}, indicating a statistically significant difference between the synchronous and asynchronous approaches. This result supports the rejection of the null hypothesis H_0 and does not reject the alternative hypothesis H_1, indicating that the **asynchronous approach is significantly faster than the synchronous approach**.

It is also important to note that the standard deviation of the asynchronous approach is significantly larger than that of the synchronous counterpart. This occurs because, in synchronous algorithms, operations are executed immediately, while in asynchronous algorithms, small tasks are individually scheduled to the devices. This individual scheduling becomes sensitive to latency, especially if the devices are loaded, resulting in greater variability in execution times.

5.3.2. Test Case 2

The analysis of execution times for critical routines in *Test Case 2* reveals distinct results regarding the impact of execution on the *CPU* compared to the *GPU*. As evidenced in Table 4, Table 5, and Table 7, the insertion, movement, and transition operations show that there are **no significant advantages between performance on the CPU and GPU**. Only the contact operation presents a significant difference, with execution on the CPU being 1.6 times faster than execution on the GPU.

Figure 8, Figure 10, Figure 9, and Figure 11 illustrate and visually compare the differences in execution times between the approaches.

Therefore, the **null hypotheses H_0 for the insertion, movement, and transition routines were not rejected**, indicating no significant advantages between execution on the *CPU* and the standard configuration, as evidenced by the high p values. However, **for the contact routine, the alternative hypothesis H_1 was accepted**, indicating a marked

Table 4. Execution: insertion

Approach	Mean [s]	Min [s]	Max [s]	Ratio
Insertion GPU	73.431 ± 16.310	55.239	93.531	1.19 ± 0.38
Insertion CPU	61.524 ± 13.829	48.159	87.195	1.00

Table 5. Execution: movement

Approach	Mean [s]	Min [s]	Max [s]	Ratio
Movement GPU	77.459 ± 17.718	53.858	101.082	1.00
Movement CPU	82.442 ± 12.377	64.529	99.681	1.06 ± 0.29

Table 6. Execution: contact

Approach	Mean [s]	Min [s]	Max [s]	Ratio
Contact GPU	69.267 ± 14.220	48.523	90.124	1.60 ± 0.33
Contact CPU	43.161 ± 0.805	41.838	44.388	1.00

Table 7. Execution: transition

Approach	Mean [s]	Min [s]	Max [s]	Ratio
Transition GPU	85.123 ± 14.336	60.055	105.431	1.00
Transition CPU	86.186 ± 15.762	65.164	109.046	1.01 ± 0.25

and favorable difference for the standard execution compared to execution on the *CPU*. This relationship of values and results is summarized in Table 8.

Table 8. Results of Test Case 2

Operation	t-value	p-value	Result
Insertion	1.76	0.0957	H_0 accepted: No significant difference
Movement	-0.729	0.476	H_0 accepted: No significant difference
Contact	5.8	0.000254	H_1 accepted: Significant difference
Transition	-0.158	0.876	H_0 accepted: No significant difference

6. Conclusions

This work proposed an *SEIRS* agent-based simulator for the spread of dengue fever, utilizing the *Senders* parallel programming model. The simulator was implemented using a heterogeneous approach, assigning different tasks to the *CPU* and *GPUs* to leverage the specific advantages of each hardware type. The experimental evaluation demonstrated the computational feasibility of this approach, showing viable execution times for simulating various scenarios. The results also highlighted the importance of the asynchronous approach for performance optimization, as well as the impact of offloading critical routines to appropriate devices. With this, the authors aim to contribute to the field of computational epidemiology by providing an alternative approach for simulating disease spread.

It is important to note that, despite the *Sender model* being in the middle of the process for standardization, it's current reference implementation on the *NVIDIA HPC SDK* only targets *NVIDIA* hardware. So, the portability benefits of using a parallel model built in the language will only come in later versions of the language.

For future work, the authors propose several strategies to enhance the simulator's performance and applicability. First, implementing efficient or lock-free synchronization structures on GPUs could optimize memory usage and enable larger-scale simulations. Second, exploring methods to minimize variability in execution times for the asynchronous approach—possibly by improving task scheduling and resource management—is recommended. Additionally, adapting the model for other infectious diseases would make the simulation more versatile and applicable in different epidemiological contexts. Furthermore, validating the simulator with different numbers of GPUs is crucial to assess performance and behavioral differences. Conducting further testing with varying

Figure 8. ***insertion***

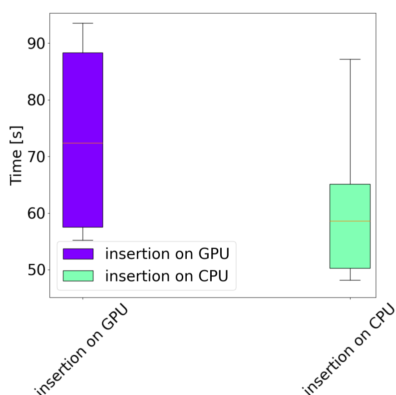


Figure 9. ***contact***

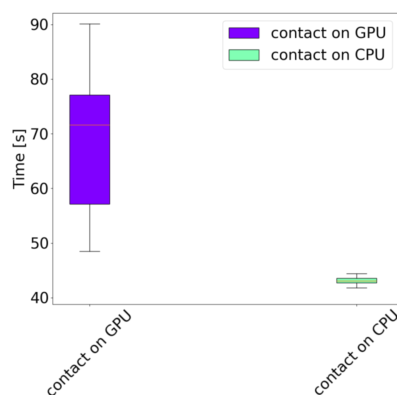


Figure 10. ***movement***

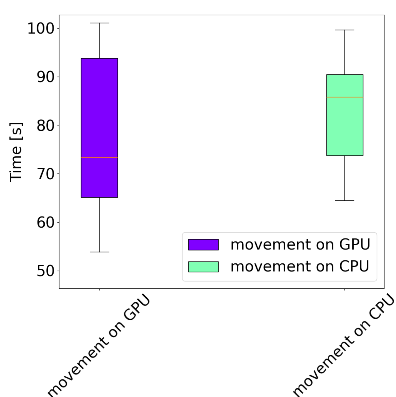
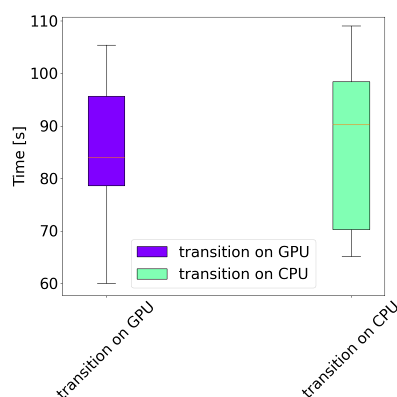


Figure 11. ***transition***



densities of humans and mosquitoes is also advised, as these factors could significantly influence GPU utilization and scalability. Finally, once the aforementioned standardization process is complete, the authors intend to make the implementation hardware-agnostic, allowing it to operate on *GPUs* beyond those of *NVIDIA*.

Acknowledgments

The authors would like to thank the High-Performance Computing Park (*PCAD*) at the Institute of Informatics of the Federal University of Rio Grande do Sul (*INF/URGS*) for the infrastructure offered and the Western Paraná State University for all the support.

References

Breyer, M., Van Craen, A., and Pflüger, D. (2022). A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware. In *International Workshop on OpenCL*, pages 1–12.

Brown, G., Reyes, R., and Wong, M. (2019). Towards heterogeneous and distributed computing in c++. In *Proceedings of the International Workshop on OpenCL*, pages 1–5.

Chien, S., Peng, I., and Markidis, S. (2019). Performance evaluation of advanced features in cuda unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 50–57. IEEE.

CPP reference (2023). Resource acquisition is initialization (raii). https://en.cppreference.com/w/cpp/language/raii. Accessed: 2023-12-09.

Cunha, E. A. A. et al. (2022). Aperfeiçoamento e ajuste paramétrico de modelo baseado em agentes para simulação da transmissão da dengue. Programa de pós-graduação em ciência da computaç ão, Universidade Estadual do Oeste do Paraná, Cascavel-PR.

Deakin, T. and McIntosh-Smith, S. (2020). Evaluating the performance of hpc-style sycl applications. In *Proceedings of the International Workshop on OpenCL*, pages 1–11.

Dominiak, M., Evtushenko, G., Baker, L., Teodorescu, L. R., Howes, L., Shoop, K., Garland, M., Niebler, E., and Lelbach, B. A. (2024). P2300r10: std::execution. ISO/IEC JTC1/SC22/WG21.

Elsheikh, A. (2024). Promising and worth-to-try future directions for advancing state-of-the-art surrogates methods of agent-based models in social and health computational sciences. *arXiv preprint arXiv:2403.04417*.

Gallagher, K., Bouros, I., Fan, N., Hayman, E., Heirene, L., Lamirande, P., Lemenuel-Diot, A., Lambert, B., Gavaghan, D., and Creswell, R. (2024). Epidemiological agent-based modelling software (epiabm). *Journal of Open Research Software*, 12(1).

Kitson, J., Costello, I., Chen, J., Jiménez, D., Hoops, S., Mortveit, H., Meneses, E., Yeom, J.-S., Marathe, M. V., and Bhatele, A. (2024). A large-scale epidemic simulation framework for realistic social contact networks. *arXiv preprint arXiv:2401.08124*.

Kühn, M. J., Lenz, P., Schmidt, A., Koslow, W., Bicker, J., Schmieding, R., Siggel, M., Binder, S., Lutz, A., Korf, S., et al. (2024). Software: Memilio v1. 1.0-a high performance modular epidemics simulation software. Technical report, Population Health Sciences.

Lin, W.-C., Deakin, T., and McIntosh-Smith, S. (2022). Evaluating iso c++ parallel algorithms on heterogeneous hpc systems. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 36–47. IEEE.

NVIDIA (2024). HPC SDK — developer.nvidia.com. https://developer.nvidia.com/hpc-sdk.

NVIDIA Corporation (2023). Nvidia nsight systems. Acesso em 6 de maio de 2024.

Peter, D. (2023). hyperfine.

Rachah, A. and Silva, T. L. (2024). An agent-based model for controlling pandemic infectious diseases transmission dynamics with the use of face masks. In *AIP Conference Proceedings*, volume 3034. AIP Publishing.

Rosenstrom, E. T., Ivy, J. S., Mayorga, M. E., and Swann, J. L. (2024). Covsim: A stochastic agent-based covid-19 simulation model for north carolina. *Epidemics*, page 100752.

Thomopoulos, V. and Tsichlas, K. (2024). An agent-based model for disease epidemics in greece. *Information*, 15(3):150.

West, R. M. (2021). Best practice in statistics: Use the welch t-test when testing the difference between two groups. *Annals of clinical biochemistry*, 58(4):267–269.