Performance and Programmability Comparison of Parallel Agent-Based Modeling (ABM) Libraries

Kevin Wang

A paper

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2022

Program Authorized to Offer Degree:

Computer Science & Software Engineering

University of Washington

Abstract

Programmability and Parallelization Comparison of Parallel Agent-Based Modeling (ABM) Libraries

Kevin Wang

Chair of the Supervisory Committee:
Professor Munehiro Fukuda
Computer Science & Software Engineering

Agent-based modeling (ABM) allows researchers in the social, behavioral, and economic (SBE) sciences to use software to model complex problems and environments that involve thousands to millions of interacting agents. These models require significant computing power and memory to handle the high numbers of agents. Various research groups have implemented parallelized ABM libraries which allow models to utilize multiple computing nodes to improve performance with higher problem sizes. However, it is not clear which of these libraries provides the best-performing models and which is the easiest to develop a model with. The goal of this project is to compare the performance and programmability of three current parallel ABM libraries, MASS C++, RepastHPC, and FLAME. The Distributed Systems Lab at the University of Washington Bothell developed Multi-Agent Spatial Simulation (MASS) C++ as a C++-based ABM library. Different research groups developed RepastHPC and FLAME before MASS C++, and SBE researchers have successfully used these libraries to create agent-based models. To measure performance, we designed a set of seven benchmark programs covering various problems in the SBE sciences, and implemented each of them three times using MASS C++, RepastHPC, and FLAME. We compared the average execution times of the three implementations for each benchmark to determine which library performed the best. We found that models written with MASS C++ generally performed the best with MASS C++ compared to RepastHPC. On the other hand, FLAME had the worst performance across all benchmarks since it could not handle the same parameters given to the MASS C++ and RepastHPC implementations. To measure programmability, we performed a static code analysis and manual code review of each benchmark implementation to assess the three libraries quantitatively and qualitatively. We found that in terms of quantitative metrics, MASS C++ was slightly more programmable than the others. However, regarding qualitative differences, MASS C++ and RepastHPC had different approaches which present different benefits, while FLAME had the most limitations.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thank you to Rithvik Koppolu, who helped perform the performance measurements as part of his internship with the University of Washington Bothell, sponsored by the IEEE Diversity and Inclusion Fund.

Chapter 1. INTRODUCTION

Researchers across all disciplines, particularly those studying Social, Behavioral, and Economic (SBE) sciences, use conceptual modeling to assess and simulate problems so that they can better understand and predict them [20][21]. For problems that involve several entities interacting with each other and their environment or ecosystem, researchers use agent-based modeling (ABM), also known as agent-based modeling and simulation (ABMS) [17]. In agent-based modeling, systems are represented as collections of autonomous, dynamic agents operating within a designated simulation space. These agents may have simple computational capabilities, but their behavior as a collective can form accurate models [4]. The models become more accurate to real-world systems and populations when given a larger simulation space and a higher number of agents. However, the problem size that a model can handle is limited by the model's scalability, its ability to handle increased problem sizes without compromising its execution time [21].

The scalability of agent-based models is a critical consideration in their design and implementation, because high numbers of entities are required for accurate simulations. On a single process, an ABM can simulate the activities of up to millions of simple entities, but such simulations can take hours to complete, and more entities or higher complexity will increase the execution time [6][21]. Therefore, agent-based models may use parallel computing to minimize the execution time even with a large workload. Parallel agent-based modeling distributes the simulation space across these processes and coordinates the agents such that they can still operate as if it was a single process.

Researchers can parallelize agent-based models on a single machine by using multithreading or multiprocessing, but this still limits the model to the machine's memory and processing power [21]. By distributing the model across a cluster of computing nodes, the model is no longer limited to the capabilities of a single machine. This addresses the scalability limitations of ABMs by providing more memory for the simulations to operate on. More memory allows for much higher entity counts, and the decreased load on each node can also allow simulations to be completed faster. Researchers may create their agent-based model entirely from the ground up and implement cluster-based parallelization themselves. However, this is costly and complex, so most ABM researchers will only implement sequential (non-parallelized) models [21]. Alternatively, researchers may build their model using an existing cluster-based ABM library as a basis. However, it is unclear which parallel ABM libraries are best for which models, or whether one library is better for developing models in general. Current research compares parallel agent-based models based, but they only implemented a single, generic mathematical model rather than models relevant to the SBE sciences [18].

Our research project analyzes and compares three different C/C++-based implementations of generalized, cluster-based ABM libraries: Multi-Agent Spatial Simulation (MASS) C++, RepastHPC, and FLAME. MASS C++ was written by the Distributed Systems Lab (DSLab) at the University of Washington Bothell [10]. Assessing the capability of MASS C++ compared to other cluster-based ABM libraries will guide further development for the MASS library and demonstrate its potential to ABM researchers. To compare the three libraries, the DSLab wrote seven benchmark programs for each library for a total of 21 benchmark programs. These benchmarks provide objective criteria for a comparative analysis on the bases of performance and programmability. Professor Munehiro Fukuda selected the seven benchmark programs in 2017 as a set of comprehensive and realistic applications for agent-based modeling which could adequately test MASS C++ and other ABM platforms such as RepastHPC and FLAME [21]. The benchmarks simulate applications in the social, behavioral, and economic (SBE) sciences as well as tasks in biology, ecology, and city-planning. The selected benchmarks are as follows:

- **Social Network** - a simulation of online communities and their interactions and activities. Each agent acts as an individual within the network, interacting with its neighbors [2].
- **Tuberculosis** - a simulation of the movement and growth of tuberculosis bacteria [8].
- **Conway's Game of Life** - a simple and predictable cellular automaton [11].
- **MATSim** - a simulation of transport organization based on a queuing network [13][22].
- **Brain Grid** - a simulation of a neural network modeling electrical connections along synapses between neurons [20].
- **Bail-in/out** - a simulation involving clusters of agents with each cluster representing firms, banks, or households and modeling how they interact [15].
- **Virtual Design Team** - a simulation of a design team for modeling productivity [16]. The team is represented as a hierarchical structure that moves through product phases.

To perform an accurate comparative analysis of the three libraries, we developed a testing methodology using the 21 benchmark programs. We defined consistent sets of parameters that would demonstrate the relative performance of each benchmark with a given number of computing nodes. In addition, we developed driver programs and shell scripts for executing the benchmarks, making the testing process easier and more replicable for future study. Finally, we performed a static code analysis and manual code review on each benchmark implementation to assess the overall programmability of each ABM library based on both qualitative and quantitative characteristics.

The structure of this paper is as follows: Chapter 2 covers the background behind the three parallel agent-based modeling libraries and the seven benchmark programs; Chapter 3 discusses previous efforts to compare agent-based modeling libraries; Chapter 4 details the methods by which this study compared the three libraries; Chapter 5 shows and evaluates the results of the comparison; and Chapter 6 summarizes and concludes the findings of this paper and discusses future opportunities for study.

Chapter 2. BACKGROUND

Section 2.1 describes and compares the design and implementation of the MASS C++, RepastHPC, and FLAME parallel ABM libraries. Section 2.2 describes the set of seven parallel ABM benchmark programs we designed and implemented for the three ABM libraries

2.1 PARALLEL AGENT-BASED MODELING LIBRARIES

MASS C++, RepastHPC, and FLAME are libraries which allow for a general implementation of parallelized agent-based modeling (ABM). However, agent-based modeling does not inherently require parallel computing, with most models executing as a sequential program rather than using parallel processes [21]. Some of these sequential models might utilize multithreading to improve performance, but the memory and processing power is still limited to that of a single machine. These memory and processing power limits represent the spatial and temporal scalability of a program respectively. Sequential models work for small problem sizes but lack the spatial and temporal scalability to address realistic problem sizes which better represent real scenarios. Parallel computing greatly increases the temporal and spatial scalability of agent-based models [17]. Parallelization with ABMs is based on a shared-memory programming paradigm where processes and threads share memory for the agents to execute on, increasing the model's spatial scalability. In addition, the multiple computing nodes effectively share their computing power to process the agents together, increasing the model's temporal scalability.

Researchers interested in agent-based modeling may use simpler, GUI-based but non-parallel methods for agent-based modeling [21]. These methods include interpretive platforms such as NetLogo, Repast Simphony, and MASON, which are easy to use but are less scalable with the high problem sizes that are often required for realistic agent-based modeling [3][23][24]. These platforms are also based on interpretive languages such as Java and Logo, which inherently have slower code interpretation than native execution with compiled languages such as C/C++. Rather than using these platforms, researchers may also decide to create bespoke agent-based models for a specific problem, as is the case with MATSim which is designed for traffic simulations [13]. These solutions are tailor-made for the problem but must be developed in-house and from the ground up, which is complex and costly. Researchers are therefore less likely to implement parallelization for these custom models because that would add further complexity.

Considering the limitations of interpretive platforms and application-specific agent-based models, there remains a need for generalized parallel ABM platforms which are highly scalable for realistic problem sizes while also being easily programmable for a variety of applications. To address this need, we developed the Multi-Agent Spatial Simulation (MASS) C++ library as a general parallel ABM platform. RepastHPC and FLAME are two other implementations which have been used in the industry for scalable agent-based modeling. We compared the performance and scalability of these three parallel ABM libraries to determine which of them best satisfies the general needs of agent-based model researchers.

Sections 2.1, 2.2, and 2.3 describe the designs of MASS C++, RepastHPC, and FLAME and cover the similarities and differences between them.

2.1.1 *MASS C++*

The Distributed Systems Laboratory (DSLab) at the University of Washington Bothell developed the Multi-Agent Spatial Simulation (MASS) library for the C++ compiled language. MASS is defined by its use of “Places” as well as its migrating Agents [5][9][21]. “Places” are agent-navigable spaces that are dynamically allocated over a cluster of computing nodes to form a network. Places can interact with the Agents that exist on or migrate to them and can exchange information with other Places. Agents in MASS migrate between Places to obtain or change information about those Places and interact with the Agents at their current Place.

We implemented parallelization for MASS C++ using intercommunicating processes that are forked over a cluster of computing nodes. These processes communicate to each other directly through TCP sockets rather than through the Message-Passing Interface (MPI) that RepastHPC and FLAME use. MASS C++ also features multi-threading, which allows for parallelization of Place and Agent functions and message exchanges. This multi-threading can be used on a single node or on multiple nodes, with each node’s process managing its local threads. The multithreading is designed to improve the performance compared to a single-threaded process but may not be compatible with all models because of issues caused by thread synchronization.

Developers seeking to create agent-based models with MASS C++ must define the behavior of their application’s Places and Agents by extending the base classes with new functions [5]. In their main program, they must place code that tells MASS to dynamically load Places and Agents. Then, within a loop, they must coordinate the function invocations, message exchanges, and the dynamic creation/termination/migration of agents that occurs in each iteration of the model.

2.1.2 *RepastHPC*

The Argonne National Laboratory has developed a parallel simulation platform for agent-based modeling known as RepastHPC. RepastHPC is defined by its execution environment, known as a “Context”, that populates the shared simulation spaces within it, known as “Projection” instances, with “Agents” [7][21][25]. The Context also manages the removal and movement of Agents over the shared space. RepastHPC Agents are implemented as extensible C++ objects, with users defining Agent behavior through its functions. Contexts and Projections act as spaces for Agents to exist on, but they do not perform any computations of their own. RepastHPC utilizes a dynamic discrete-event scheduler with conservative synchronization to organize its Agents. The order of Agent actions is determined through the scheduling of their Events which occur at specific ticks in the scheduler.

In addition to the Repast-specific constructs, RepastHPC also uses constructs from Logo, a programming language primarily used for education [7]. The designers of RepastHPC propose that it is easier to conceive of and design models using these constructs. In particular, “Turtles” represent mobile agents, “Patches” represent fixed agents, “Links” connect Turtles to form networks, and an “Observer” provides overall model management. Not all models require these Logo-like constructs, and so it is possible to develop models strictly using RepastHPC-specific constructs. We developed the RepastHPC benchmark implementations using elements from both.

Parallelization for RepastHPC is achieved using the Message-Passing Interface (MPI) which facilitates the communication between nodes on the cluster. Agents in one node on the cluster are unable to directly communicate with agents on other clusters. Instead, a copy of an agent must be created on the other node which retrieves up-to-date information about the original agent as needed [17]. This process causes communication overhead, and the agent copies add memory costs. RepastHPC models operate within “ghost spaces”, in which Agents can view the simulation boundary of adjacent MPI ranks. This allows agents in each rank to see the information of nearby spaces, such as their Agent occupancy. Ghost spaces are read-only, so there is a possibility of collisions when multiple Agents end up in the same logical position, and this is something users of the library must be aware of.

RepastHPC and MASS C++ are both implementations of parallel agent-based modeling, but they have fundamental differences in design which can affect their relative performance and programmability. RepastHPC’s contexts, projections, and agents are analogues to MASS’ processes, places, and agents respectively, but they have different capabilities. For example, RepastHPC is unable to handle collisions on a system level. Unlike RepastHPC’s Contexts and Projections, Places in MASS C++ can act upon the Agents that are on them and can perform computations of their own [21]. However, RepastHPC features more robust multi-dimensional spaces and graphing topologies for developers to use as compared to MASS, which can only use its Places arrays to emulate graph structures.

2.1.3 *FLAME*

The University of Sheffield, UK has developed FLAME as a C-based agent-based modeling platform [9]. A key aspect of FLAME is that it does not instantiate any simulation space on memory, with its processes retaining only environmental variables for its Agents to use. The Agents themselves, including their initial data and functions, are declared in XML files which act like C++ header files.

FLAME’s parallelization is based on the Message-Passing Interface (MPI) like RepastHPC. Because FLAME’s Agents do not share a simulation space, they communicate with each other through message broadcasts and exchanges on message boards at each MPI rank. This results in significant communication overhead as agents wait to receive messages. FLAME does not support the movement of agents like MASS or RepastHPC, instead statically mapping its Agents on MPI ranks and relying on message broadcasts to transmit and change Agent states. FLAME's lack of a managed simulation space means that collisions can only be avoided if every Agent broadcasts their location to all other Agents. Each agent must then process every incoming message to determine which messages are relevant before it can change its state. The overhead of this process is computationally costly and so agent migration is impractical on FLAME.

Despite FLAME's significant limitations, it may still be chosen for programming agent-based models that do not require agent migration. The code surrounding the allocation and deallocation of memory for agents and the code for inter-agent communication are generated automatically, allowing developers to focus on defining agent behavior, which both simplifies and limits programmability.

2.2 Parallel ABM Benchmark Programs

We selected seven agent-based models based on systems and problems in the social, behavioral, and economic (SBE) sciences as benchmarks for a comparative analysis between ABM libraries [20][21]. These models cover a range of distinct agent behaviors which perform and must be programmed differently, allowing for a more comprehensive performance and programmability analysis than a single, generic model. Agent-based models feature either static or dynamic agents, based on whether the agents have mobility across the simulation space. Depending on the model and the organization of the simulation space, they may have different forms of mobility. In addition, these agents may be grouped or operate independently.

1. Social Science
    a. Social Network – a network of static agents

2. Behavioral Science
    a. Virtual Design Team – communication within distinct groups of static agents
3. Economic Science
    a. Bail-in, Bail-Out – communication between groups of static agents
4. Biological/Ecological Sciences
    a. Brain Grid – dynamic agents jumping across a geometric space
    b. Conway's Game of Life – a simple cellular automata based on static spaces
    c. Tuberculosis – dynamic agents migrating across a geometric space
5. Urban Planning
    a. MATSim – dynamic agents migrating over a network of spaces

2.2.1 *Bail-in, Bail-out, Financial Simulation*

In 2015, Klimek et al., a group of economists and computer scientists, studied the problem of how to model and optimally resolve financial crises involving failing banks [15]. The most common method is a bailout using government funds and tax money to contribute to a loan to the failing banks, protecting the banks and creditors but at a cost to taxpayers. Bail-ins are an alternative crisis resolution in which the government cancels the failing banks' debts to their depositors, converting that debt into equity for the bank, protecting the banks and taxpayers but at a cost to the creditors of those banks. To determine which option works better, Klimek et al. developed an agent-based modeling framework called the Mark I CRISIS model to simulate a closed economy of banks, firms, and households [15]. This model forms the basis of the "Bail-in, Bail-out" benchmark programs we wrote to study the performance of ABM libraries for financial applications.

Our implementation of the model represents the firm owners, workers, firms, and banks as Agents. In this closed economy, all money must come from other agents in the simulation and the banks cannot print money or bring it in from outside of the simulation. The households consist of firm owners and workers who also possess personal bank accounts, the firms have profits and losses and must take out loans from banks, and the banks must offer loans while maintaining their cash reserves. The model ends after any bank in the simulation has reached bankruptcy, but a secondary option allows the model to continue for a specified number of iterations.

2.2.2 *Brain Grid, Self-Organizing Neural Network*

To cover the problem space of neuroscience and neural networks, we designed and implemented a self-organizing neural network simulation called Brain Grid [14]. The Brain Grid model takes place in a two-dimensional space composed of cells. Each cell may start empty, or it may start with a neuron. From these initial neurons, each representing a soma, axons and dendrites grow outward. Somas that have formed synaptic connections between each other will relay neural signals to other somas through the connections. These signals are simulated by agents jumping across the geometric space that represents the neural network. The model ends after a specified number of iterations.

2.2.3 *Conway's Game of Life*

We implemented the common Conway's Game of Life cellular automata because of its ubiquity as a program and its compatibility with agent-based modeling [11]. In Game of Life, each cell switches between an "alive" or "dead" state, based on the number of living cells in its Moore neighborhood. When implemented as an agent-based model, static agents represent each cell and synchronously communicate with their surrounding agents to determine whether they are alive or dead. The cellular automata continues until it reaches a specified number of iterations.

2.2.4 *Multi-Agent Transport Simulation (MATSim)*

The Multi-Agent Transport Simulation (MATSim) is an agent-based model framework for modeling the movement of vehicles and traffic flows using a queuing network of agents moving through a simulation space [13][26]. MATSim is a popular agent-based model framework because of its use in urban planning and city modelling, but it is limited to that application and does not support parallelization. We designed a simpler form of MATSim for use as a benchmark for comparing ABM libraries. Unlike the original MATSim which can model complex road geography, the benchmark version uses a simple adjacency list comprised of vertices representing intersections. Agents, representing cars, start on intersections on the outer edges of the simulation space, with the goal of moving to a predetermined target destination at the center. The path each car takes is precalculated based on an input file, so the model does not actively perform shortest-path calculations. However, both vertices/intersections and edges/roads have limited capacities, so during each iteration, some cars will need to wait before they can proceed on their path. This forms a queueing network which requires agents and the simulation space to communicate to avoid collisions. The simulation ends when all agents have reached their destination.

2.2.5 *Social Network*

Researchers commonly use agent-based modeling for simulating human systems because agents can abstractly model people and their interactions and connections on a large scale [2]. These models provide insights to the social scientists who study human behavior in certain contexts and situations they may not be able to simulate otherwise [4]. Social networks are one of the most common structures of interactive and interconnected humans, so they are commonly modeled with agents. We designed and implemented a simple social network model which simulates interactions between a network of thousands of people. Each Person is a vertex in the network that tracks an adjacency list of their first-degree friendships with other people. Each Person relays *N* messages to their first-degree friends, with each message containing a list of the Person's friends at the *Nth* degree. By the end of the simulation, each Person outputs a comprehensive list of their friends at each degree up to a degree specified by the user. The following is an example of what this output may look like:

- Agent 0's friends
    - Degree 1: 1, 5, 8, 10
    - Degree 2: 2, 3, 4, ... 21
    - Degree 3: 7, 11, 20, 30, 31, ... 52

2.2.6 *Tuberculosis*

Researchers in the biological/ecological sciences also use agent-based modeling to model the behavior of organisms interacting with each other in ecosystems [19]. The models are most applicable to simple cellular organisms which, like agents, have simple behaviors but at a large-scale can form complex ecosystems. Some biologists also use these models to design new treatments or even new novel cell functions. For example, in 2016, researchers in the field of synthetic biology proposed the use of these models to inform their design of synthetic biological systems that can perform computations, sense diseases, or produce drugs and chemicals [12]. However, most biologists use these models to simulate existing cellular organisms including viruses or diseases.

In 2004, researchers writing for the Journal of Theoretical Biology proposed using an agent-based model to simulate Mycobacterium tuberculosis (Mtb) bacteria in a human lung, the cause of the Tuberculosis (TB) disease [19]. The adaptive immune response to Mtb causes formations of multicellular structures, called granulomas, in the lungs of infected individuals. The formation of granulomas progresses the disease in infected individuals, so understanding this process guides understanding of the disease and its treatment. Modeling the process of granuloma formation involves simulating the spatial and temporal organization and interactions between the Mtb cells and the immune system cells. The model was originally implemented as a non-parallel, CPU-based model, but another group of researchers in 2009 implemented a GPU-based model to improve performance [8]. We implemented a simplified but parallelized version of the original CPU-based model as a benchmark for parallel agent-based modeling libraries. We simplified the immune system into macrophages, T-cells, and chemokines, and we modeled their generation, diffusion, and termination within the simulation space. The model ends after a specified number of iterations.

2.2.7 *Virtual Design Team (VDT)*

To cover the problem space of behavioral and organizational science, we developed implementations of the Virtual Design Team (VDT) model. VDT is a model first devised by Raymond Levitt in the late 1980s but iterated into the 2000s which sought to model organizations/teams as they performed routine design or product development work [16]. Levitt argues that through a similar process by which engineers design their products through modeling, analyzing, and evaluating virtual versions of their systems, organizations and teams can be designed and improved in the same way.

Our implementation of VDT forecasts the productivity of multiple teams of engineers using separate organizational structures. Each team is composed of project leads, UX designers, senior developers, junior developers, and test engineers, with tasks assigned to that team flowing through the hierarchy in that same order. Each team is composed of 25 engineers, each of which is represented by agents, and though the teams can be organized differently, they must all have at least one member in each role. Teams are assigned a list of tasks which may either be production tasks or collaborative tasks. Production tasks are passed down through the team hierarchy, with each engineer role taking a different amount of time to complete their part, whereas collaborative tasks involve multiple engineers simultaneously. Teams work independently of each other, but the engineers within each team work closely together, so this simulation exemplifies models focused on intra-group agent communication. The model ends after all teams have processed all tasks.

2.2.8 *Summary*

Table 2.1 provides an overview of the differences between each benchmark program in terms of how they manage their agents and space.

Table 2.1. Comparison of benchmark model designs

Benchmark Programs	**Domain**	**Computational Model**	**Entities**	**Space Structure**	**Agent Population**
Bail-in, Bail-out	Economic	Inter-group comm	Grouped agents	Groups	Multi-groups of agents (m:1)
Brain Grid	Biological	Agent jump over 2D	Agents on space	2D	Agents on given cells (0-m:1)
Game of Life	Behavioral	Cellular automata	Space only	2D	Cells in 2D (0:1)
MATSim	City Planning	Agent move on net	Agents on space	Group	Agents on given vertices (0-m:1)
Social Network	Social	Networked agents	Space only	Graph	Vertices in net (0:1)
Tuberculosis	Biological	Agent move on 2D	Agents on space	2D	Agents on given cells (0-m:1)
VDT	Behavioral	Intra-group comm	Grouped agents	Groups	Multi-groups of agents (m:1)

Benchmark Programs	**Classes of Agents**	**Execution time Agent Management**	**Inter-entity Communication**	**Lock/unlock**	**Collision control**
Bail-in, Bail-out	Multiple	Static	Multicast	Yes	No
Brain Grid	Single	Spawn, kill, move	8 neighbors, multicast	No	Yes
Game of Life	Single	Static	8 neighbors	No	No
MATSim	Single	Move	Adjacent roads	Yes	Yes
Social Network	Single	Static	Adjacent vertices	No	No

Tuberculosis	Multiple	Spawn, kill, move	8 neighbors	No	Yes
VDT	Single	Static	Multicast	Yes	No

We see that the selected benchmarks cover a variety of different application domains, as well as different ways of using entities. The set of chosen benchmarks is therefore comprehensive for the purpose of a general performance and programmability comparison. Due to the different characteristics of each application, some of them may lend themselves to a certain ABM library in terms of programmability. For example, the agent migration features supported in MASS C++ might make it the best choice for MATSim, which involves agent movement across space during execution time.

Chapter 3. RELATED WORK

As a result of the high number of agent-based modeling platforms, many researchers have performed surveys, reviews, and comparative analyses to assess them and determine which systems are best for which applications [1]. In the period between 2016-2019, three groups of researchers, one writing for the Journal of Supercomputing and two for the Computer Science Review, surveyed and reviewed the current state-of-the-art agent-based modeling software [1][17][18]. These studies examined RepastHPC and FLAME among several other agent-based modeling platforms. However, no studies have assessed the MASS C++ library since it has been in active development during the period these articles were written and published.

A review written by Abar et al. in 2017 broadly compares a comprehensive list of agent-based modeling tools and platforms [1]. This review does not attempt to compare the performance of each platform, and it is not limited to cluster-based, parallel ABM platforms. The survey briefly covers FLAME and RepastHPC, but only in limited detail and with only a manual code and documentation review process. According to the review, FLAME has moderate programmability and is suited for large-scale simulations. In comparison, RepastHPC is said to have moderate to complex programmability and to be suited for extreme-scale simulations. The overall review is useful for a brief overview of the many ABM platforms, but more data is required to qualify and quantify the differences between them. The performance and programmability results of our study provide a more in-depth comparison of FLAME, RepastHPC, as well as MASS C++.

A survey conducted by Rousset et al. in 2016 proposed a generic reference model by which specifically distributed/parallel agent-based modeling software can be compared [18]. The survey uses the reference model to compare the performance of FLAME, RepastHPC among other parallel ABM platforms. The performance results found that RepastHPC consistently performed faster than FLAME with the same model parameters and the same number of computing cores. Specifically, with 16 cores the RepastHPC implementation was 2.02 times faster than FLAME while at 128 cores it was 9.26 times faster. The survey notes that when the model was executed with smaller parameters, FLAME was able to perform faster than RepastHPC. By examining the memory consumption of the two implementations, the study found that FLAME uses significantly more memory than RepastHPC for the same parameters, indicating that RepastHPC has better scalability. However, the generic reference model is a simple mathematical model and does not represent realistic problems in the social, behavioral, or economic fields. Researchers seeking to develop a model may prefer an ABM platform that is suited for their specific field, so a generic reference model may not provide all the data they need to make an informed judgment.

In 2019, Moreno et al. designed a benchmark based on the reference model proposed by Rousset et al., and used it to compare FLAME, FLAME GPU, RepastHPC, and EcoLab [17]. When comparing FLAME and RepastHPC, RepastHPC was once again found to scale better than

FLAME. The RepastHPC benchmark implementation showed a linear increase in execution time as the workload increased, whereas FLAME showed a quadratic increase. This is mostly a result of FLAME's agent communication overhead compared to RepastHPC, with FLAME communicating at least 100 times more bytes. Interestingly, FLAME performed computations up to 10 times faster than RepastHPC, and with smaller workloads the model was faster than RepastHPC. However, the communication overhead caused FLAME to scale poorly as the workload increased compared to RepastHPC. These findings contextualize the results of our performance comparison, which supplements this generic benchmark with seven additional benchmarks and introduces MASS C++ into the comparison.

Chapter 4. MEASUREMENT METHODS

4.1 PERFORMANCE MEASUREMENT METHODS

We measured the performance of the 21 benchmark programs by executing each benchmark a total of twelve times to obtain average execution times. Although each benchmark has its own set of parameters which affect its execution time, we only wanted to assess how performance changes as the number of computing nodes increases from one to eight nodes. Therefore, we calibrated all other parameters to achieve an approximately 20-minute execution time on a single node and only increased the number of nodes to decrease that execution time. For each number of nodes, we executed the benchmark three times to obtain an average execution time. With the same data, we also compared the relative performance of MASS C++, RepastHPC, and FLAME's implementations of each benchmark.

MASS C++, unlike RepastHPC and FLAME, has an option which allows for multi-threading the processes running on each computing node. We ran each benchmark with 4 threads to assess how this feature affects the performance of MASS C++ compared to the single-threaded MASS C++, RepastHPC, and FLAME.

Benchmarks written for the FLAME library, due to how FLAME handles the instantiation of agents using pre-generated XML files, were unable to handle the higher numbers of agents which were used for the MASS C++ and RepastHPC benchmarks. We created another set of parameters with fewer agents that achieves an approximately 20-minute single-node execution time on FLAME so that a comparison could still be made between all three ABM libraries. This required that we also execute the MASS C++ and RepastHPC benchmarks using that second set of parameters. In this study, benchmarks executions using the set of higher/larger parameters tuned using MASS C++ are referred to as "using the higher parameters", while executions using the smaller parameters tuned for FLAME are referred to as "using the smaller parameters". Due to issues with the RepastHPC implementations of Social Network and Virtual Design Team which limit their scalability, they could not be compared in the performance assessment. The following is an overview of the two parameter sets used for each benchmark:

1. **Bail-In, Bail-Out**
    a. **Higher Parameters:** 20,000 workers, 2,000 firms, 5 banks, and 22000 turns/iterations
    b. **Lower Parameters:** 20,000 workers, 2,000 firms, 5 banks, and 1100 turns/iterations
2. **Brain Grid**
    a. **Higher Parameters:** 100 turns/iterations, 540 x 540 grid
    b. **Lower Parameters:** 100 turns/iterations, 180 x 180 grid
3. **Game of Life**
    a. **Higher Parameters:** 250 turns/iterations, 1000 x 1000 grid
    b. **Lower Parameters:** 250 turns/iterations, 165 x 165 grid
4. **Multi-Agent Transport Simulation (MATSim)**
    a. **Higher Parameters:** 110 x 110 grid, 1210 cars
    b. **Lower Parameters:** 40 x 40 grid, 160 cars
5. **Social Network**
    a. **Higher Parameters:** 580,000 agents, 50 friends each, up to 3rd degree of friendship
    b. **Lower Parameters:** 21,500 agents, 50 friends each, up to 3rd degree of friendship
6. **Tuberculosis**

    a. **Higher Parameters:** 10 iterations, 48 x 48 grid
    b. **Lower Parameters:** 8 iterations, 16 x 16 grid
7. **Virtual Design Team (VDT)**
    a. **Higher Parameters:** 1000 tasks, 400 teams
    b. **Lower Parameters:** 1000 tasks, 64 teams

4.2 BENCHMARK SCRIPTS

To improve the process of measuring performance across the 21 benchmark programs, we developed scripts which could compile and run each benchmark. We also standardized the scripts which were already written for some of them. This was necessary because all the benchmark programs were written at different times and often by different students, and therefore each required their own steps to compile and run. The "compile" shell scripts automate the process of exporting the necessary agent-based modeling libraries and compiling the benchmark code using g++ or Makefiles. Similarly, the "run" shell scripts simplify the process of executing each benchmark, although it does still require the user to know exactly which arguments they wish to pass to the program. These scripts simplify what would otherwise be several console commands into a single script execution. In addition, we wrote readme files for each benchmark which explains the process of executing these scripts. The benchmark runner program described in Section 5.3 uses these scripts and further simplifies the process of executing benchmarks.

4.3 BENCHMARK RUNNER PROGRAM

We developed a benchmark runner program, BenchmarkRunner, which leverages the compile and run scripts described in Section 5.2 to further simplify and automate the process of executing the benchmarks. Instead of navigating to each benchmark's directory and running the scripts manually, users can execute the BenchmarkRunner from a central directory and specify the target benchmark from there. The process navigates to the directory of the target benchmark and then runs the corresponding shell scripts. We designed BenchmarkRunner to allow users to specify prerecorded parameters into a text file that are then automatically entered into the corresponding scripts.

We further simplified the BenchmarkRunner program by writing more shell scripts to run the BenchmarkRunner. Specifically, “compile_benches.sh” and “run_benches.sh will run BenchmarkRunner which compiles or runs the benchmarks, only requiring the user to specify the target benchmark. Another shell script, “automate.sh”, automates the process of running multiple benchmarks sequentially with different numbers of computing nodes. For example, it will run a benchmark with a single node three times, then it will run the same benchmark with two nodes three times, and so on until it has run all numbers of nodes with all specified benchmarks. This script, which allows for the execution of benchmarks overnight without monitoring, is the primary method by which we collected the benchmark performance measurements.

We developed an additional functionality for BenchmarkRunner which allows multiple benchmarks to be executed simultaneously. This feature uses multiple execution threads, each of which remotely connects to other computing nodes with SSH2, navigates to the directories of a target benchmark, and then executes their scripts at the same time. This was implemented to allow for complete utilization of all 24 computing nodes in UW Bothell’s Linux machine network. On a single-threaded BenchmarkRunner instance, a single benchmark uses up to 8 out of the 24 available computing nodes, leaving 16 nodes unused. With the multi-threaded BenchmarkRunner, a user could simultaneously run up to 24 single-node or three 8-node benchmarks. We did not use this feature for our performance measurements because of inconsistencies between the computing nodes themselves, causing benchmark execution to succeed on certain nodes and fail on others.

4.4 PROGRAMMABILITY ASSESSMENT METHODS

We assessed the programmability of the ABM libraries by quantitatively analyzing each benchmark with static code analysis tools and performing a qualitative manual code review of each benchmark.

4.4.1 *Static Code Analysis*

We performed static code analysis to obtain quantitative metrics for each benchmark. Specifically, we used a program that counts the lines of code in each benchmark implementation and broke down the lines used for specific aspects of each model. For example, for each benchmark implementation we counted the lines of code dedicated to defining the behavior of agents and compared which library requires more. In the case of benchmarks written with the FLAME library, the tool must be used before the benchmarks are compiled. This is because FLAME's compilation process automatically generates many files not written by the benchmark programmer and are therefore not part of the programmability assessment. We gathered these quantitative metrics to compare each library:

- Lines of Code – the total lines of code for the entire program
- Control Flow Statements –a count of all if statements, for loops, while loops, and switch statements. This acts as an approximate measurement of cyclomatic complexity
- Lines of Boilerplate Code – the library code that sets up the parallel cluster but is not dedicated to the management of space or agents
- Lines of Agent Code – lines of code concerning agent behavior and computations
- Lines of Spatial Code – lines of code concerning the behavior of space
- Lines of Model Design Code – total lines of agent and spatial code

- Lines of Model Management Code – lines of code concerning the management of the agents and space on a model

4.4.2 *Code Review*

The manual code review of each benchmark produces qualitative observations about how models are implemented for each ABM library. We can use these observations to assess the capabilities and limitations of each ABM library broadly. These qualitative metrics describe how each library handles specific aspects of each benchmark:

- Computational Model – the approaches that the ABM library takes to different types of agent behavior and actions
- Entities (Space and/or Agents) – whether the ABM library supports the definition of the agents as well as the space of the model
- Space Structure – how the ABM library structures the simulation space that the model exists on
- Agent Population – how the ABM library populates the simulation space
- Multiple Classes of Agents – whether the ABM library supports multiple classes of agents or spaces
- Runtime Agent Management – how the ABM library manages the agents in the model during runtime
- Inter-Agent Communication – how the ABM library handles agents communicating with each other
- Synchronization (Lock/Unlock) – how the ABM library manages synchronization of data between agents

- Collision Control – how the ABM library addresses the risk of agent collision

Chapter 5. PERFORMANCE AND PROGRAMMABILITY EVALUATION

Chapter 5 illustrates and discusses the results of the performance and programmability analyses for each benchmark model and each ABM library. The performance results are visualized using line plots. The specific average execution times in seconds are listed in the tables in Appendix A. In addition, the standard deviation and variance of each set of executions is listed in the tables in Appendix B. The programmability assessments explain how each ABM library implements each model to provide a qualitative comparison of the benchmark models and ABM libraries. Then, these implementations are measured quantitatively, comparing lines of code

5.1 Bail-in, Bail-out, Financial Simulation

All implementations of the Bail-In, Bail-Out financial/bank simulation use a constant number of 20,000 workers, 2,000 firms, and 5 banks, only changing the number iterations/turns to affect the execution time. On a single node, the MASS C++ model must process 22,000 iterations to reach the target average execution time of 20 minutes, so the “higher parameters” uses 22,000 iterations. The FLAME implementation of the model is unable to process that many iterations in a reasonable time, so we selected 1,100 iterations for the “smaller parameters”.

5.1.1 *Performance Results*

We executed MASS C++ with 22,000 iterations using both single-threaded and four-threaded execution. Figure 5.1 illustrates the difference in execution times as both the number of nodes and the number of threads increase. The MASS C++ implementation of the Bail-In, Bail-Out model does not demonstrate a significant improvement in performance when the number of nodes is increased. This is a result of its usage of MASS C++'s "Place" construct to represent the model's "FinancialMarket". Each "FinancialMarket" Place acts as a form of shared memory, allowing local Agents (workers, banks, and firms) to communicate with each other. In addition, FinancialMarkets communicate with each other across the model to exchange transaction information. These communication processes incur a constant communication overhead per iteration which is dependent on the number of agents in the model, and so increasing the number of nodes does not significantly decrease execution time. The performance only worsens when using four threads because the added thread synchronization incurs an overhead of its own while not addressing the existing communication overhead.

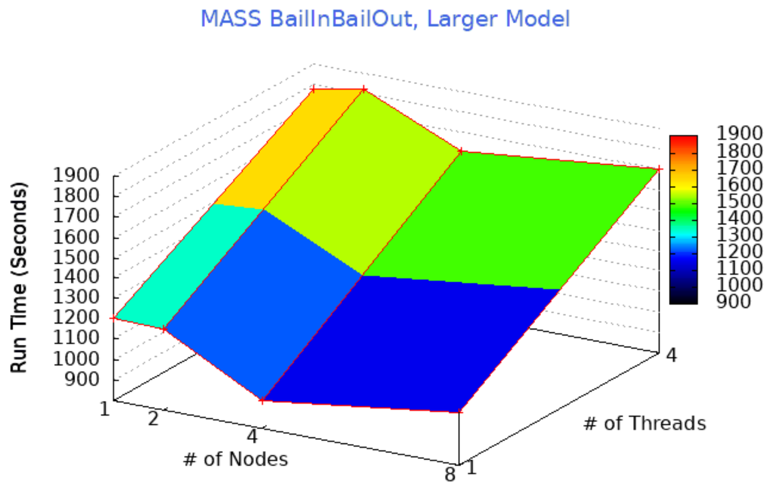

Figure 5.1. MASS C++ Bail-In, Bail-Out using the higher parameters, execution time in seconds as the number of nodes increases

Figure 5.2 demonstrates that, unlike the MASS C++ model, the RepastHPC implementation of Bail-In, Bail-Out has improved performance when increasing the number of nodes. On a single node, it takes an average of 6,812 seconds to complete 22,000 iterations, while on eight nodes it takes 1,493 seconds. However, even at eight nodes the RepastHPC model performs worse than the MASS C++ model. This is because of the model's usage of RepastHPC's "Observer" construct, which coordinates the agents on the model. Like the MASS C++ model, this coordination and communication incurs significant overhead. Unlike MASS C++, RepastHPC is better able to distribute this process across the cluster which improves performance as the number of nodes increases. The overhead still appears to be worse overall because of RepastHPC's communication system which requires communicating agents to be copied at other nodes.

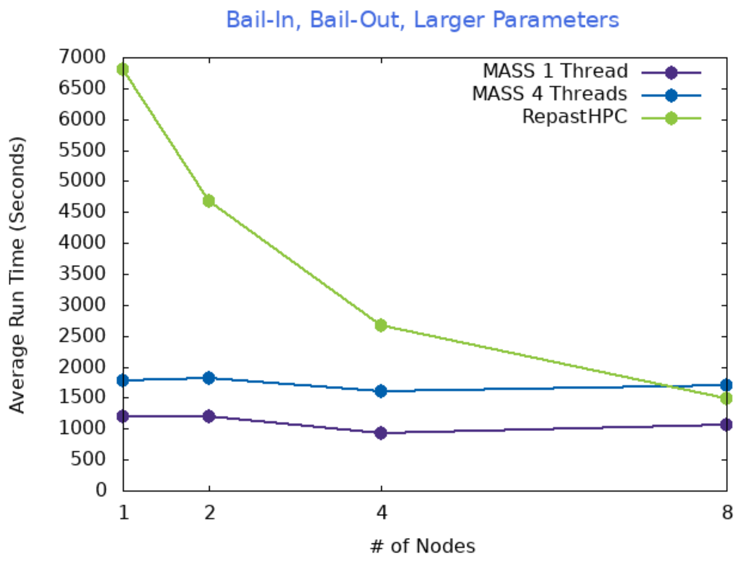

Figure 5.2. Bail-In, Bail-Out models using the higher parameters, execution time in seconds as the number of nodes increases

While FLAME appears to be able to handle the number of agents in the model, it is still significantly slower than MASS C++ and RepastHPC so it must use a lower number of iterations to execute in a timely manner. Figure 5.3 shows the execution times of the MASS C++, RepastHPC, and FLAME models with 1,100 iterations. It takes FLAME approximately 1,212 seconds to process 1,100 iterations, though this does decrease to 185 seconds when using a cluster of eight nodes. The FLAME model's worse performance is due to its extremely high communication overhead, which is itself a consequence of FLAME's broadcast-based, message board communication system [18]. The MASS C++ model performs much faster than the FLAME model, completing 1,100 iterations in less than 60 seconds. However, performance still does not improve with more nodes in the model. RepastHPC's performance is better than the FLAME model, but worse than the MASS C++ model.

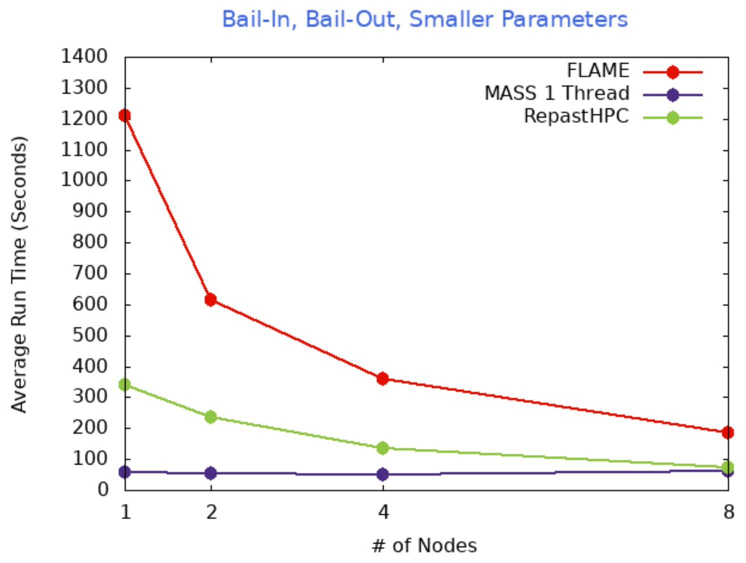

Figure 5.3. Bail-In, Bail-Out models using the smaller parameters, execution time in seconds as the number of nodes increases

5.1.2 *Programmability Assessment*

The MASS C++ model instantiates with several FinancialMarket Places, enough to contain one Firm Agent per Place. These individual FinancialMarket elements facilitate communication between the Agents in the simulation, acting as a sort of shared memory and messaging system for the model. The Bank, Firm, and Worker Agents are instantiated and associated with each other and with FinancialMarket Places. The Agents perform their various transactions, which are accordingly exchanged across the FinancialMarket with exchangeAll().

The RepastHPC model uses the Observer to manage the model, instantiating and coordinating all of its Bank, Firm, and Worker agents. The Agents and the Observer coordinate between each other using temporary Messenger agents which carry and exchange transaction information across the model.

The FLAME model instantiates all Bank, Firm, and Worker agents into the initial 0.xml file. Their transactions between each other are coordinated through the message board which they all communicate on, but there is no central controller like MASS C++'s FinancialMarket or RepastHPC's Observer. The FLAME model struggles with ending the model once bankruptcy has occurred, so for the sake of the comparison, all of the model implementations end after a certain number of iterations instead.

Table 5.1 compares the lines of code used to write each ABM library's implementation of the Bail-In, Bail-Out model. The MASS C++ model has the fewest lines of code overall, with almost half as many as RepastHPC. However, it uses a space construct as part of its model which makes its implementation slightly different from the RepastHPC and FLAME models.

Table 5.1. Quantitative comparison of benchmark implementations by library

Library	**Lines of Code**	**Control Flow Statements**	**Boilerplate LoC**	**Agent LoC**	**Space LoC**	**Model Design LoC**	**Model Management LoC**
MASS C++	688	69	123	331	186	517	48
RepastHPC	1293	74	134	699	0	699	460
FLAME	894	36	265	344	0	344	285

5.2 Brain Grid, Self-Organizing Neural Network

For the Brain Grid Self-Organizing Neural Network model, the number of iterations and the size of the simulation space determine its execution time. For MASS C++, a 540 by 540 grid space running for 100 iterations takes 20 minutes, whereas for FLAME a 168 by 168 grid running for 100 iterations takes 20 minutes.

5.2.1 *Performance Results*

Figure 5.4 shows that the MASS C++ Brain Grid model significantly improves in performance when the number of nodes increases, as well as when the number of threads increases. On single-threaded models, the execution time drops from 1,191 seconds at one node to 170 seconds at eight nodes. On a single node, the execution time drops from 1,191 on one thread to 487 seconds on four threads. On both eight nodes and 4 threads, the execution time drops as low as 84 seconds. The MASS C++ model effectively utilizes parallelization to divide the simulation space across the cluster's computing nodes, with each node using multiple threads to perform computations.

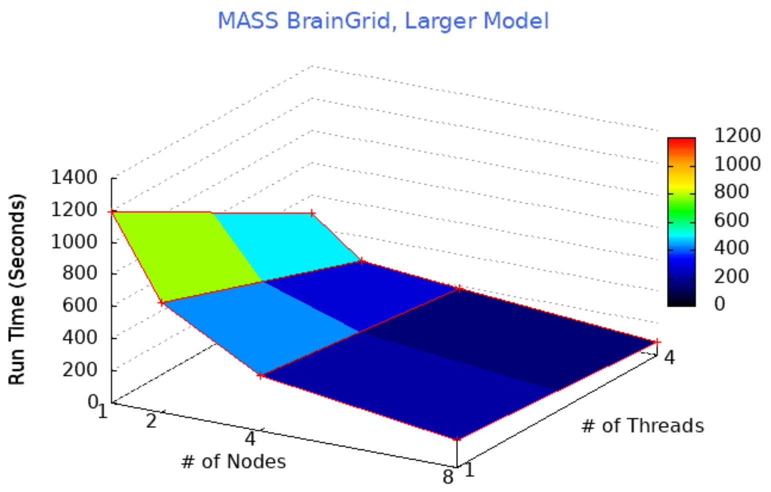

Figure 5.4. MASS C++ Brain Grid using the higher parameters, execution time in seconds as the number of nodes and the number of threads increase

The RepastHPC implementation of Brain Grid uses Logo constructs to simulate the neural network grid. The neurons and signals are represented by Turtles, which are mobile agents. The grid spaces they travel across, known as BrainPlaces, are represented by Patches which are fixed agents. An Observer at each rank manages the model by coordinating the creation of new agents and movement of existing agents. Figure 5.5 shows that the RepastHPC model has worse performance than the MASS C++ model. Once again, the communication overhead of RepastHPC itself likely causes this worsened performance, especially with this many spawning, moving, communicating agents.

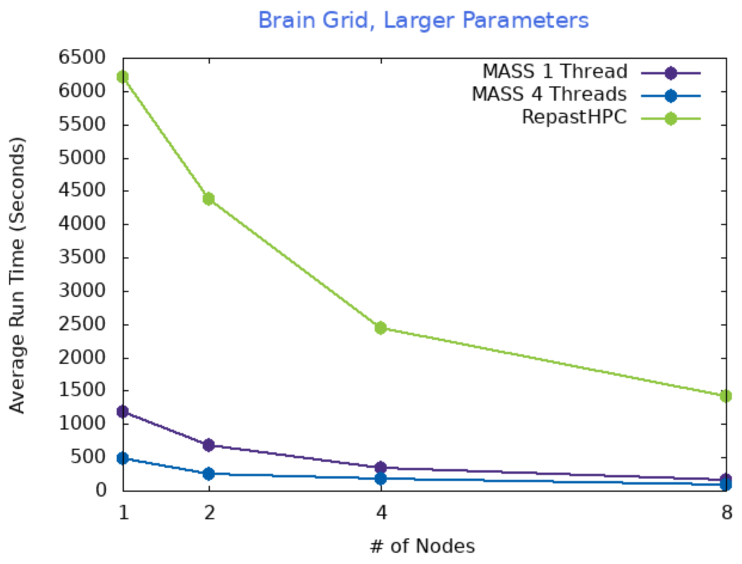

Figure 5.5. Brain Grid models using the higher parameters, execution time in seconds as the number of nodes increases

Figure 5.6 shows that the FLAME model can handle a 168 by 168 simulation space running for 100 iterations. Figure 5.6 demonstrates that the MASS C++ Brain Grid model has significantly better performance than the FLAME version. In addition, while Figure 5.6 shows that RepastHPC also performs better, MASS C++ is still better than both.

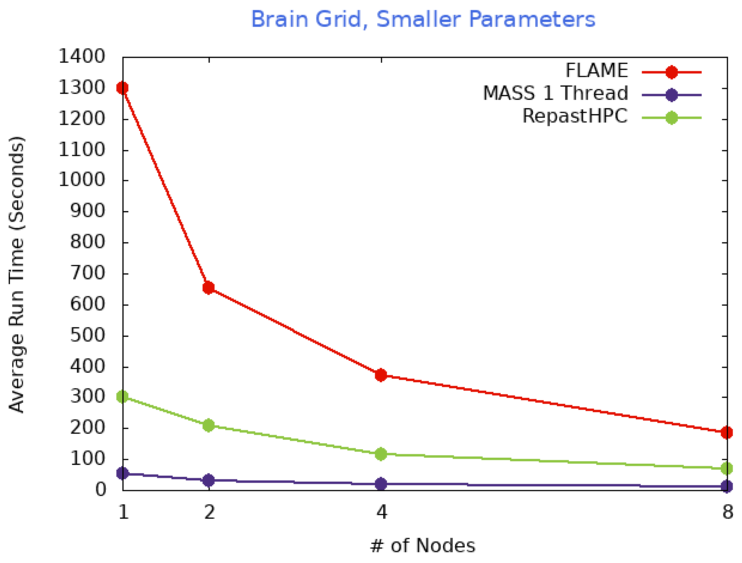

Figure 5.6. Brain Grid models using the smaller parameters, execution time in seconds as the number of nodes increases

5.2.2 *Programmability Assessment*

The MASS C++ model starts as a grid of “BrainPlaces”, some of which begin as neurons. “GrowingEnd” agents move outwards from these starting neurons using neighbor information from exchangeBoundary(). These movements grow the network by changing unoccupied BrainPlaces into somas, dendrites, axons, and synaptic terminals. Once synaptic connections have been made between neurons, a soma BrainPlace starts sending signals to other somas via the connections, calling exchangeAll().

The RepastHPC model populates Neuron Agents over a two-dimensional SharedDiscreteSpace and diffuses them to neighboring coordinates to mimic the grow of neural network. RepastHPC natively supports the SharedDiscreteSpace structure, which allows for more robust control over the model. Rather than changing the state of existing spaces on the network, the model directly adds and removes Agents from the context using addAgent() and removeAgent(). Signal Agents jump from soma to soma with moveTo().

The FLAME model populates the network with pairs of Place Agents and Neuron Agents in 2D. The model uses Agents to mimic Places because FLAME only supports Agent constructs. Each Neuron sends a request-to-grow message to the corresponding remote Place that arbitrates multiple different requests to allow only one neuron to occupy that place. Once a neural connection is set up, a Neuron agent starts sending a signal to another.

Table 5.2 compares the lines of code used to write each ABM library’s implementation of the Brain Grid model. The MASS C++ model has the fewest lines of code overall, while the FLAME model has the most. Unlike the RepastHPC and FLAME models which use agents, the MASS C++ model represents Persons in the social network using Places.

Table 5.2. Quantitative comparison of benchmark implementations by library

Library	**Lines of Code**	**Control Flow Statements**	**Boilerplate LoC**	**Agent LoC**	**Space LoC**	**Model Design LoC**	**Model Management LoC**
MASS C++	918	129	167	282	431	713	38
RepastHPC	1184	118	152	514	154	668	364
FLAME	1403	103	322	461	23	484	597

5.3 Conway's Game of Life

For the Game of Life cellular automata model, the number of iterations and the size of the simulation space determine its execution time. The model is very simple, so all implementations can handle a much larger simulation size. A 250 by 250 grid space running for 250 iterations takes 20 minutes for MASS C++, whereas a 165 by 165 grid running for 250 iterations takes 20 minutes for FLAME.

5.3.1 *Performance Results*

Figure 5.7 shows that, for the MASS C++ model, the performance increases as the number of nodes and the number of threads increase. With 8 nodes each running with 4 threads, the average time reaches as low as 69 seconds. RepastHPC and FLAME lack this multi-threading capability and cannot achieve similar performance improvements compared to a single node.

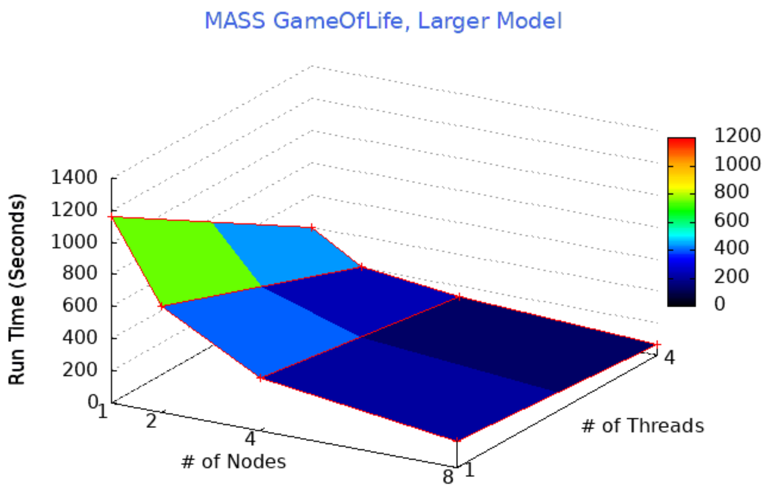

Figure 5.7. MASS C++ Game of Life using the higher parameters, execution time in seconds as the number of nodes and the number of threads increase

Figure 5.8 shows that RepastHPC has worse performance as compared to MASS C++. Given the simplicity of the model in terms of what the programmer defines, the difference in performance must be attributed to the overhead of RepastHPC's own processes.

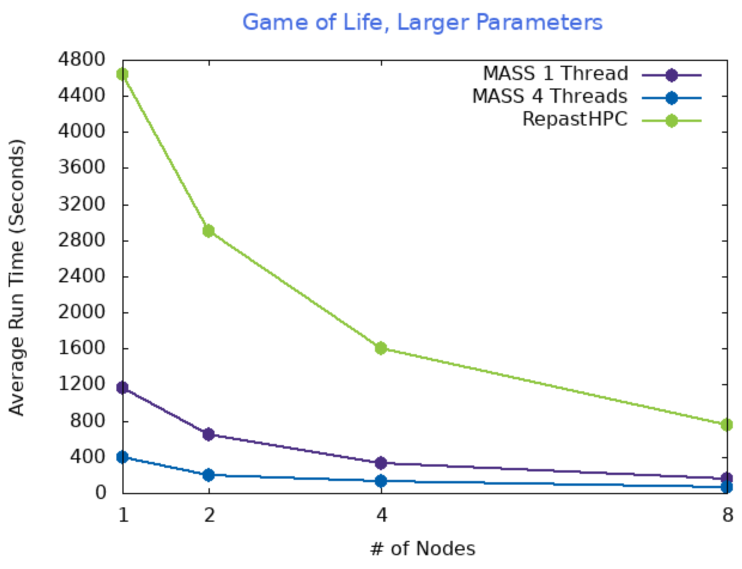

Figure 5.8. Game of Life models using the higher parameters, execution time in seconds as the number of nodes increases

Figure 5.9 shows the performance of the FLAME model with a 165 by 165 grid. In terms of scalability, this grid size is the closest out of all benchmarks to what MASS C++ and RepastHPC are capable of. However, the FLAME model is still significantly slower than the MASS C++ and RepastHPC versions. With the 165 by 165 grid used in the smaller parameters and with a single node, the MASS C++ model only takes 31 seconds to achieve what FLAME achieves in 20 minutes. RepastHPC is also able to perform significantly faster than FLAME using the same parameters, but its execution times are slower than MASS C++.

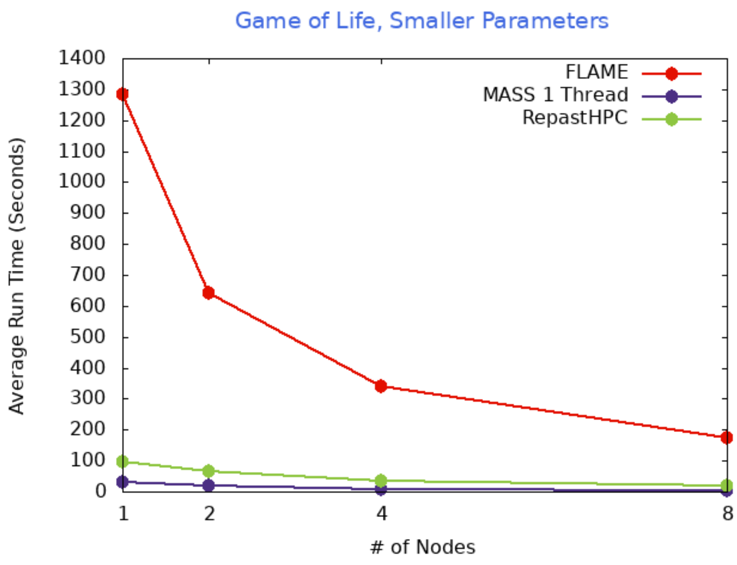

Figure 5.9. Game of Life models using the smaller parameters, execution time in seconds as the number of nodes increases

5.3.2 *Programmability Assessment*

The MASS C++ model creates a two-dimensional array of Places. Each Place exchanges its state with the Places in its Moore neighborhood through exchangeBoundary(). The state of each Place is determined when the main program executes callAll() on all Places.

The RepastHPC model creates static Patch agents in a two-dimensional space and controls them with two ask() methods. The first call of ask() to a Patch agent checks the states of its neighbors and the second call changes its state according to the state of the neighbors.

The FLAME model creates agents representing the cells, each with i- and j-coordinates. In each iteration, the Agents repeat a series of actions: write_state to send their state to neighbors, read_state to receive neighbor information, and react calls to change their state.

Table 5.3 compares the lines of code used to write each ABM library's implementation of the Game of Life model. The FLAME model has the fewest lines of code overall, while the RepastHPC model has the most. The differences are minor, however, since the models are so simple.

Table 5.3. Quantitative comparison of benchmark implementations by library

Library	**Lines of Code**	**Control Flow Statements**	**Boilerplate LoC**	**Agent LoC**	**Space LoC**	**Model Design LoC**	**Model Management LoC**
MASS C++	203	17	21	0	111	111	71
RepastHPC	235	18	72	0	49	49	114
FLAME	144	7	3	0	47	47	64

5.4 Multi-Agent Transport Simulation (MATSim)

Each MatSim model utilizes the same input files to determine the size of the model. To achieve a 20-minute execution time on MASS C++ on a single node, the intersection input file contains a 110 by 110 grid, and the car input file contains 1210 cars and their routes. The FLAME intersection input file contains a 40 by 40 grid, and the car input file contains 160 cars and their routes.

5.4.1 *Performance Results*

Figure 5.10 shows the performance of the MASS C++ MATSim model using 1 thread. The four-threaded performance is unavailable because the model stops working with multi-threading when multiple nodes are used, demonstrating that multi-threading is not always an option. This is caused by the unique shared file memory developed specifically for the MATSim model which does not work when multiple threads and nodes are contending for the shared memory. Multi-threading can still be used on a single node for the MatSim model. This results in an execution time of 640 seconds, which is still an improvement over a single-threaded single node. This multi-threaded, single node configuration may be useful if a user is only able to execute the model on a single node but seeks better performance.

Figure 5.10 also shows that RepastHPC can achieve superior performance to MASS C++ for certain models. The key difference with this model is that the Observer contains less code dedicated to coordinating communication between agents. Instead, the Observer passes static instances of the intersections and roads to each agent, and the agents determine for themselves whether they can move from one location to the next.

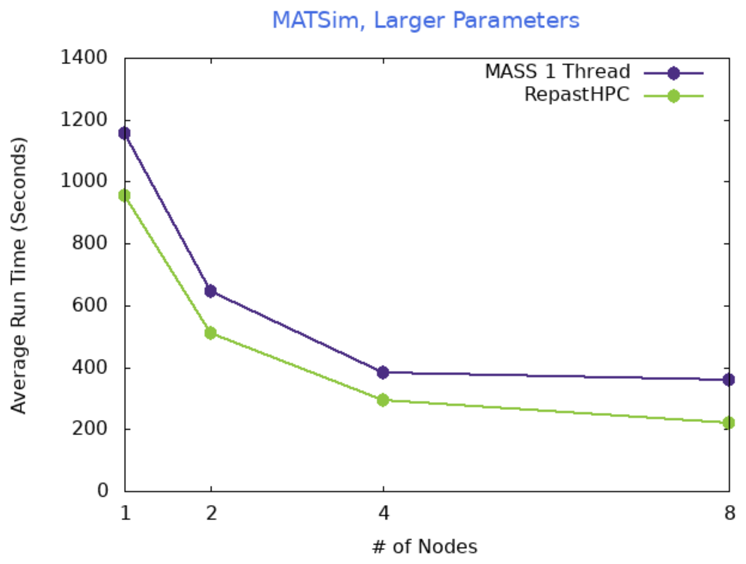

Figure 5.10. MATSim models using the higher parameters, execution time in seconds as the number of nodes increases

Figure 5.11 shows the performance results of the FLAME MATSim model using the 40 by 40 grid with 160 cars. Each car sends a message requesting to move for the upcoming iteration, which slows the model's performance. The results for the MASS C++ and RepastHPC models also demonstrate a phenomenon in which increasing the number of nodes decreases the performance when the execution times are already low. This is due to the increased overhead of managing multiple nodes and organizing the communication between agents. Regardless, the performance for both models is still much faster than the FLAME model.

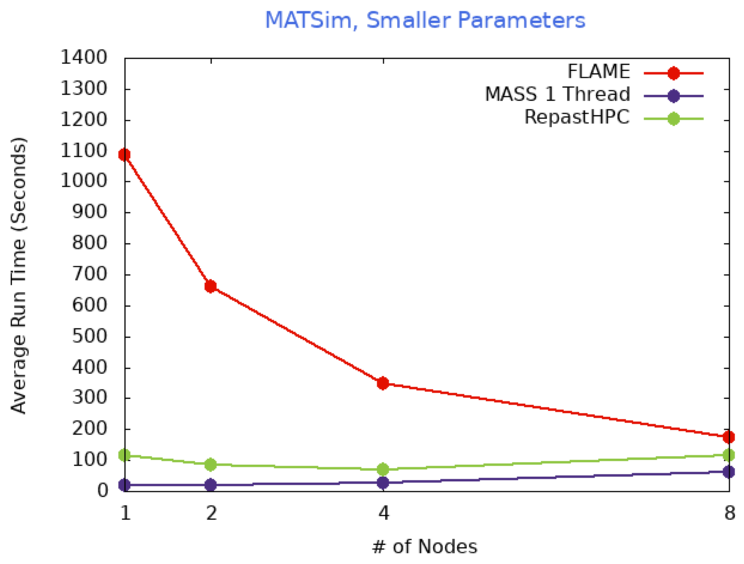

Figure 5.11. MATSim models using the smaller parameters, execution time in seconds as the number of nodes increases

5.4.2 *Programmability Assessment*

The MASS C++ model first reads the input file containing the list of intersections and roads connecting them, loading them into Places which form a shared memory for the model. These Intersections are then initialized as a network of Intersection Places, connected by Road objects. The input file containing the cars and their routes is also loaded into shared memory, and then the cars are initialized as Car Agents. Once the model has been fully initialized, the main program iteration loop tells the Car Agents to proceed along their routes towards their destination at the center. To avoid collisions or locks caused by Cars blocking each other, all Cars moving North are told to move first, then all Cars moving East move, and so on for each iteration.

The RepastHPC model uses the Observer to read the intersection/node and car/route input files. The Observer initializes the Point (intersection) objects, the Road objects connecting them, and the Agents representing the cars. The Observer then commands each Agent to move along its route, directly passing the Point and Road information in the command so that the Agent knows whether it can move.

The FLAME model reads the intersection and car input files and formats them into Place Agents and Car Agents in the 0.xml input file. Using the message board, the Car Agents send requests to move along their route to the Place Agents, which approve or deny the requests according to their capacity. If the request has been approved, the Car Agent moves along the route.

Table 5.4 compares the lines of code used to write each ABM library's implementation of the MATSim model. The MASS C++ model has the fewest lines of code overall, while the FLAME model has the most.

Table 5.4. Quantitative comparison of benchmark implementations by library

Library	**Lines of Code**	**Control Flow Statements**	**Boilerplate LoC**	**Agent LoC**	**Space LoC**	**Model Design LoC**	**Model Management LoC**
MASS C++	587	84	89	193	211	404	93
RepastHPC	897	73	104	295	173	468	325
FLAME	1025	103	322	461	23	484	597

5.5 Social Network

MASS C++ can handle 580,000 people in the social network model, although they are represented by MASS C++'s "Place" spatial constructs rather than agents like RepastHPC and FLAME. An issue occurs when the MASS C++ Social Network model is executed with four threads. FLAME can reach 20 minutes with at most 21,500 people with all other parameters being the same. Due to differences in implementation, RepastHPC can only achieve a 20-minute execution time with 1400 agents. The differences in implementation may mean that the comparison between the three libraries for this specific benchmark model are not entirely valid.

5.5.1 *Performance Results*

Figure 5.12 shows the performance of the MASS C++ Social Network model using a single thread. The single-threaded and four-threaded figures have been separated because of irregularities with the four-threaded performance. The MASS C++ Social Network model does not properly execute with four threads, producing incorrect output. It is currently unclear what causes this issue to occur, though this is the only model and configuration it seems to occur on. This demonstrates a downside of the MASS C++ multithreading feature, which is that it is not guaranteed to work even when the single-threaded operation works normally. The RepastHPC implementation of the Social Network model cannot execute with either the larger or smaller parameters. Instead, to reach 20 minutes, it can only execute up to 1400 agents. This implementation will need to be investigated and fixed to make it eligible for the performance comparison.

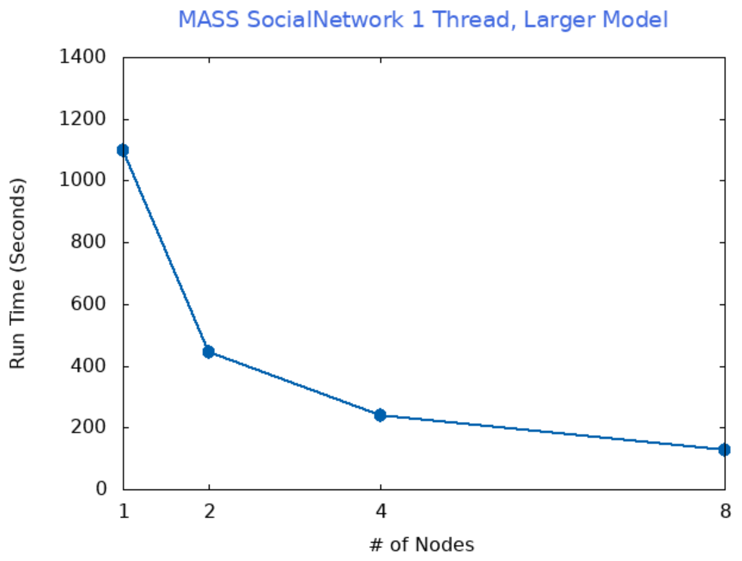

Figure 5.12. Social Network models using the higher parameters with one thread, execution time in seconds as the number of nodes increases

Figure 5.13 shows that FLAME can execute the Social Network model with no issues, though it is limited in its scalability compared to MASS C++. The MASS C++ model can execute with the smaller parameters in 33 seconds on a single node or as fast as 5 seconds with eight nodes.

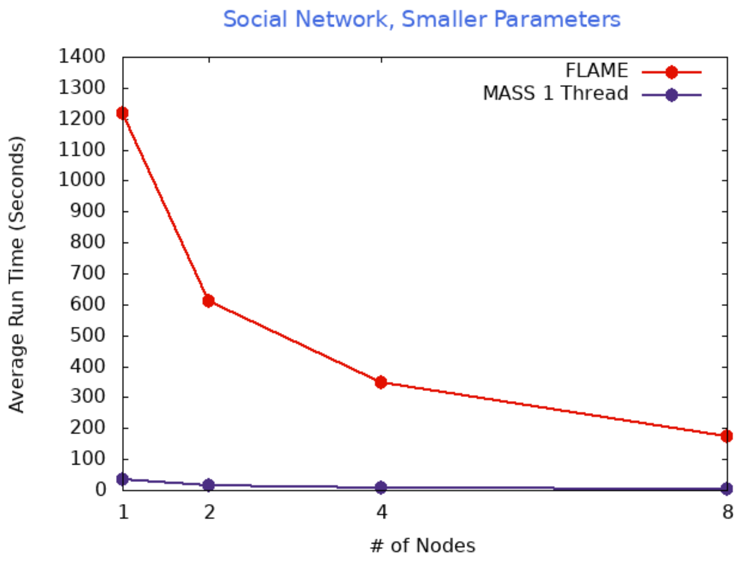

Figure 5.13. Social Network model using the smaller parameters, execution time in seconds as the number of nodes increases

5.5.2 *Programmability Assessment*

The MASS C++ model mimics a social network with a 1-dimensional array of Places, each representing a Person. Each of the Places maintains an adjacency list of other Places representing the Person's friends. The main program orders each Place to disseminate their lists of friends as messages over the network with exchangeAll().

The RepastHPC model creates a SharedContext space over the MPI ranks and populates the context with Agents which each maintain a list of their first-degree friends. The model examines an Agent, prints that Agent's first-degree friends, then examines each of the friends to print their first-degree friends, and so on until the target degree-of-friendship has been reached. There are no message exchanges because the agents exist in the same SharedContext.

The FLAME model represents social network vertices with Person agents, each repetitively sending its list of Nth degree of friends to the message boards of its first degree of friends.

Table 5.5 compares the lines of code used to write each ABM library's implementation of the Social Network model. The MASS C++ model has the fewest lines of code overall, while the RepastHPC model has the most. Unlike the RepastHPC and FLAME models which use agents, the MASS C++ model represents Persons in the social network using Places.

Table 5.5. Quantitative comparison of benchmark implementations by library

Library	**Lines of Code**	**Control Flow Statements**	**Boilerplate LoC**	**Agent LoC**	**Space LoC**	**Model Design LoC**	**Model Management LoC**
MASS C++	275	35	35	0	113	113	127
RepastHPC	426	35	87	78	0	78	261
FLAME	334	12	67	56	0	56	211

5.6 Tuberculosis

The relative complexity of the Tuberculosis model means that it reaches 20 minutes even with a much smaller grid size and number of iterations than other benchmarks. The MASS C++ and RepastHPC Tuberculosis models can both able to execute within 20 minutes with a 48 x 48 grid and 10 iterations, while the FLAME model is limited to a 16 by 16 grid with 8 iterations.

5.6.1 *Performance Results*

Figure 5.14 demonstrates that for the MASS C++ Tuberculosis model, performance increases both as nodes increase and as threads increase. The performance of the Tuberculosis model is not noteworthy, since it predictably improves in performance with more nodes and more threads.

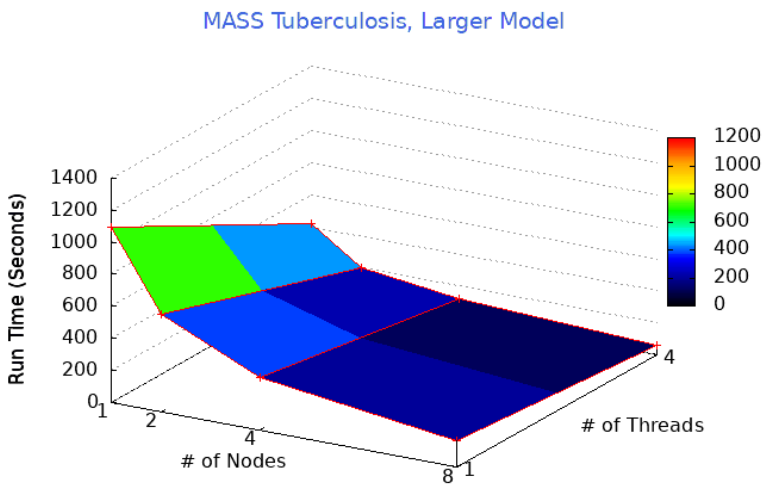

Figure 5.14. Tuberculosis models using the higher parameters, execution time in seconds as the number of nodes and the number of threads increase

Figure 5.15 shows that the RepastHPC model executes with approximately double the execution time as compared to the MASS C++ model with the same parameters.

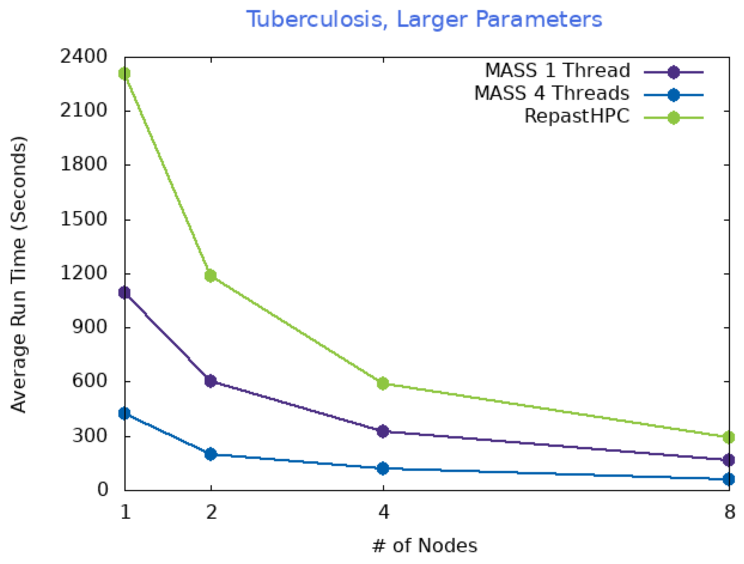

Figure 5.15. Tuberculosis models using the higher parameters, execution time in seconds as the number of nodes increases

Figure 5.16 shows that with the lower parameters, the MASS C++ and RepastHPC models perform faster than the FLAME model. This indicates that even with a significantly smaller grid size and number of iterations FLAME still cannot perform as well as MASS C++ or RepastHPC.

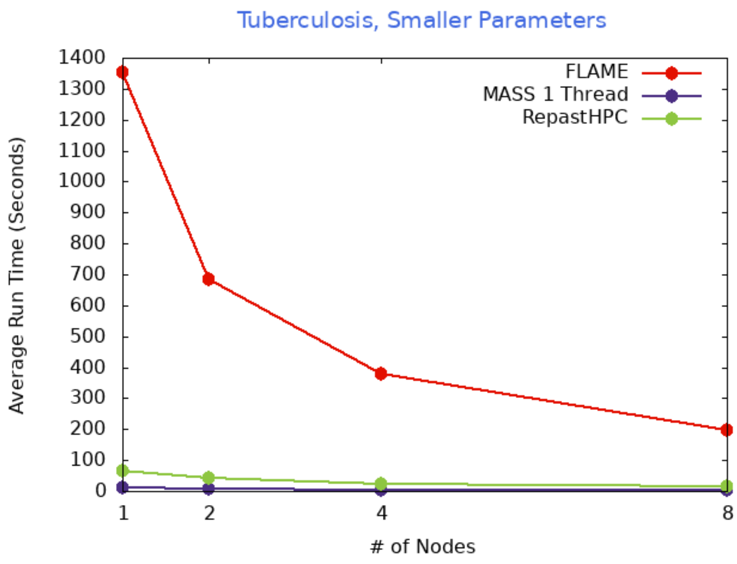

Figure 5.16. Tuberculosis models using the smaller parameters, execution time in seconds as the number of nodes increases

5.6.2 *Programmability Assessment*

The MASS C++ model creates a two-dimensional array of TB_Places, each of which can contain a single bacterium, up to one T-Cell and one Macrophage, and a chemokine (a trail left by infected Macrophages). The model begins with certain Places starting with bacteria, and with Macrophages starting in random Places. T-Cells enter the simulation through certain Places that are designated as “blood vessels”. The Agents in the MASS C++ model have autonomy of space navigation and action, allowing Macrophage agents to eat bacteria and T-Cell agents to activate/burst Macrophages as they encounter them.

The RepastHPC model instantiates the TuberculosisObserver context, which creates a two-dimensional space of LungPlace Patches. Each Patch is either healthy, a blood vessel entry point, or occupied by bacteria. The Observer initially populates and continuously spawns Macrophage agents at given LungPlaces. It later spawns T-cell agents from the blood vessel entry points. Since agent-to-agent direct communication is hard to implement in RepastHPC, the Patches manage the agents' interactions with each other.

The FLAME model instantiates place Agents that simulates a two-dimensional space. Each Agent acts as a healthy place, a blood vessel entry point, or a place occupied by bacteria. Macrophage Agents are initially populated from the 0.xml file but are continuously spawned from blood vessel entry points. T-Cell agents are also later spawned from the blood vessel entry points. Both Macrophage and T-Cell agents exchange messages with the current and neighboring place Agents. The place Agents manage the other agents regarding collision avoidance, agent migration, and agent termination.

Table 5.6 compares the lines of code used to write each ABM library's implementation of the Tuberculosis model. The RepastHPC model has the fewest lines of code overall, while the FLAME model has the most. Interestingly, MASS C++ has very little code for managing the model, with the agents and Places largely managing themselves. In contrast, FLAME has the most code for coordinating the model.

Table 5.6. Quantitative comparison of benchmark implementations by library

Library	**Lines of Code**	**Control Flow Statements**	**Boilerplate LoC**	**Agent LoC**	**Space LoC**	**Model Design LoC**	**Model Management LoC**
MASS C++	883	138	153	331	377	708	22
RepastHPC	647	68	87	199	127	326	234
FLAME	1140	60	411	172	114	286	443

5.7 VIRTUAL DESIGN TEAM (VDT)

The MASS C++ model can process 400 teams of engineers performing 1000 tasks in 20 minutes. The RepastHPC model is not completely implemented and currently only allows for a single team per rank, which is far from the limit that RepastHPC can handle. The FLAME model can process 64 teams performing 1000 tasks in 20 minutes.

5.7.1 *Performance Results*

Figure 5.17 and Figure 5.18 both show that the MASS C++ model's performance generally increases as the number of nodes increases. Interestingly, the performance of the model decreases when the number of threads increases. This is because the Virtual Design Team model operates with a specific workflow where tasks are passed between team members with an emphasis on intra-group communication. The introduction of multi-threading results in increased overhead during this communication process which does not benefit from the multi-threading.

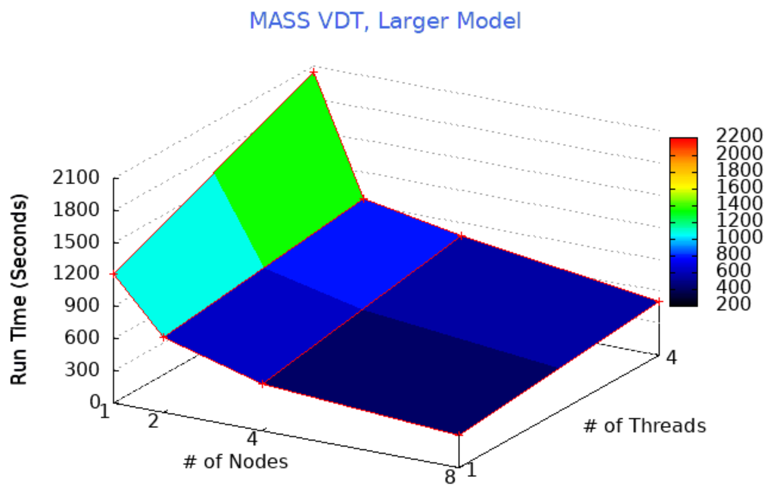

Figure 5.17. MASS C++ VDT using the higher parameters, execution time in seconds as the number of nodes and the number of threads increase

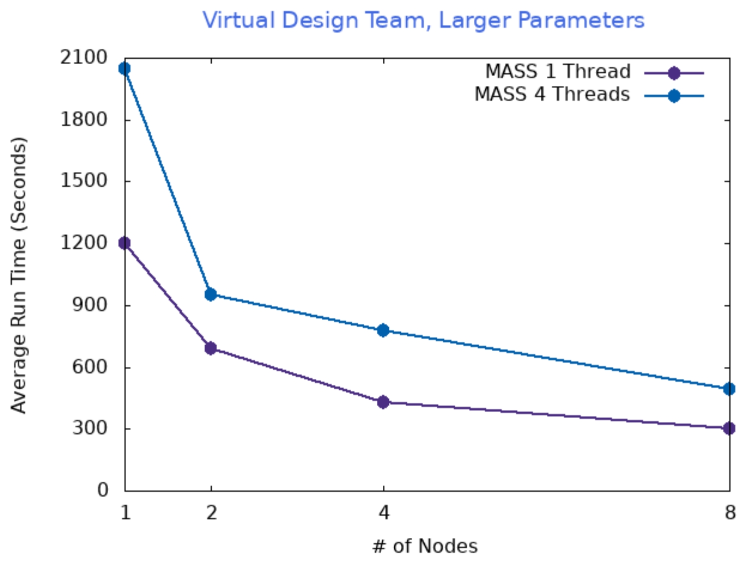

Figure 5.18. VDT models using the higher parameters, execution time in seconds as the number of nodes and the number of threads increase

Figure 5.19 shows the performance of the MASS C++, RepastHPC, and the FLAME VDT models with the smaller parameters. First, the FLAME model's execution time decreases as the number of nodes increases, but the performance appears to worsen when increasing from 4 to 8 nodes. The MASS VDT model also plateaus between 4 and 8 nodes with both the larger and smaller parameters, indicating that this is a normal pattern for the VDT model. Once again, this is the result of the increased communication overhead involved with increasing the number of processes in the model.

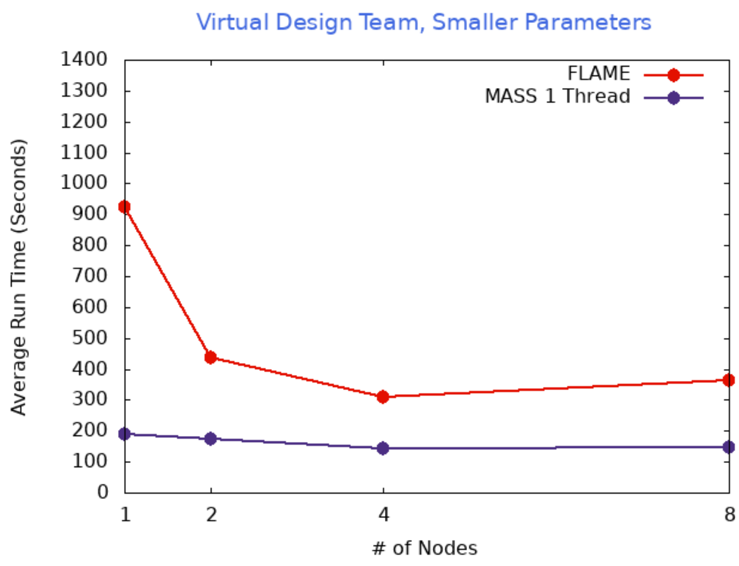

Figure 5.19. VDT models using the smaller parameters, execution time in seconds as the number of nodes increases

5.7.2 *Programmability Assessment*

The MASS C++ model instantiates Places as development teams, each containing 25 Engineer Agents. These Engineers continuously take new tasks from their supervisors (an Engineer higher in the hierarchy), spends time "working on the task", and passes the task to the task tray of the next Engineer in the workflow. The task trays must be handled as critical sections due to MASS C++'s multithreaded computation.

The RepastHPC model instantiates Observer contexts, each representing a team and populating Engineer Agents. Each Observer macroscopically manages task flows, centrally maintains task trays, and relays a task from one Agent to a lower-level Agent. Currently, since each Observer represents a team, the number of teams is limited to the number of Observers which is significantly limited compared to both MASS C++ and FLAME. This is incorrect behavior, so the performance of the RepastHPC model is not compared.

The FLAME model populates Member Agents, each with a different type and each with a given team ID. The FLAME model must use team IDs rather than other methods of grouping agents such as Places because of FLAME's limitations in defining space. The simulation also instantiates Task Agents, with each one keeping track of which Member is currently processing it. Members with the same team ID check-in and check-out the shared task, coordinating with hierarchical communication using the FLAME message boards.

Table 5.7 compares the lines of code used to write each ABM library's implementation of the Virtual Design Team model. The MASS C++ model has the fewest lines of code overall, while the RepastHPC model has the most. However, it uses Places on the simulation space to represent teams of engineers unlike RepastHPC and FLAME which are entirely agent-based.

Table 5.7. Quantitative comparison of benchmark implementations by library

Library	**Lines of Code**	**Control Flow Statements**	**Boilerplate LoC**	**Agent LoC**	**Space LoC**	**Model Design LoC**	**Model Management LoC**
MASS C++	593	86	68	72	227	299	227
RepastHPC	847	91	65	298	0	298	484
FLAME	842	48	155	405	0	405	282

5.8 Summary

5.8.1 *Performance Results*

The summaries of average execution times in both Table 5.8 and Table 5.9 indicate that the models written with MASS C++ generally had the best performance, especially when multi-threaded. An exception to this is the MATSim model where the RepastHPC version had superior performance. In general, even the single-threaded performance of the MASS C++ model surpassed the performance of most RepastHPC models. FLAME had the demonstrably worst performance and scalability of the three ABM libraries, requiring its own set of smaller parameters to be able to be compared with the other two libraries.

Table 5.8. Average execution times in seconds for each benchmark implementation using the higher parameters

# *Computing Nodes*	**Libraries**	**Bail-in, Bail-out**	**Brain Grid**	**Game of Life**	**MATSim**	**Social Network***	**Tuber-culosis**	**VDT****
1 node	**MASS 1**	1,203.913	1,191.925	1,163.152	1,154.488	1,097.882	1,096.603	1,201.433
	MASS 4	1,773.623	487.305	393.635	640.961	6.428	421.006	2,047.137
	Repast	6,812.577	6,213.978	4,647.147	956.689	1,229.729	2,308.043	0.031
2 nodes	**MASS 1**	1,193.632	678.524	654.490	646.178	446.501	603.602	692.947
	MASS 4	1,819.960	244.897	199.043	N/A	4,343.893	197.195	950.932
	Repast	4,686.141	4,381.844	2,897.910	509.679	1,240.832	1,184.286	0.025
4 nodes	**MASS 1**	934.339	339.261	328.800	383.815	239.960	325.989	432.124
	MASS 4	1,610.938	182.016	133.070	N/A	231.046	116.044	776.380
	Repast	2,666.279	2,434.674	1,605.113	293.210	1,353.733	588.439	0.022
8 nodes	**MASS 1**	1,058.749	170.401	165.606	359.572	129.332	164.175	301.425
	MASS 4	1,705.560	84.277	69.896	N/A	56.908	59.250	494.800
	Repast	1,493.997	1,425.412	751.137	219.274	1,369.767	289.893	0.016

* - *RepastHPC Social Network was limited to significantly fewer agents than MASS C++ and FLAME*
** - *RepastHPC VDT is incomplete and does not allow the correct number of teams in the simulation*

Table 5.9. Average execution times in seconds for each benchmark implementation using the smaller parameters

# *Computing Nodes*	**Libraries**	**Bail-in, Bail-out**	**Brain Grid**	**Game of Life**	**MATSim**	**Social Network**	**Tuber-culosis**	**VDT****
1 node	**MASS 1**	58.471	55.107	31.605	17.873	32.959	11.042	188.599
	Repast	338.467	301.935	97.737	115.263	N/A	64.221	0.032
	FLAME	1,212.226	1,299.290	1,284.601	1,088.153	1,218.784	1,353.899	923.952
2 nodes	**MASS 1**	53.530	32.138	17.850	20.575	16.760	6.328	173.533
	Repast	235.230	210.374	66.027	83.897	N/A	41.694	0.027
	FLAME	614.553	651.991	643.789	661.172	609.535	682.920	436.744
4 nodes	**MASS 1**	48.445	17.943	8.750	26.287	8.925	3.482	142.275
	Repast	134.152	117.525	34.167	70.57	N/A	24.566	0.023
	FLAME	358.160	371.556	339.997	346.157	348.528	378.398	309.336
8 nodes	**MASS 1**	62.317	10.093	4.825	61.510	4.811	2.082	147.935
	Repast	75.369	67.963	17.523	116.816	N/A	15.935	0.017
	FLAME	185.779	186.437	175.628	174.576	174.643	197.075	362.019

** - *RepastHPC VDT is incomplete and does not allow the correct number of teams in the simulation*

5.8.2 *Programmability Assessment*

Table 5.10 compares the programmability of MASS C++, RepastHPC, and FLAME using the averages of the quantitative measurements performed on each benchmark implementation. On average, MASS C++ appears to require the least lines of code, while FLAME requires the most. Particularly, MASS C++ requires the least boilerplate/setup code and the least code for defining agents. However, MASS C++ requires more code for defining the model space because of its Places constructs, and therefore the most agent and space code in total. The agents and space behave more autonomously than in the other libraries, so it requires the least code dedicated to model management. RepastHPC generally requires fewer lines than FLAME but requires more than MASS C++. FLAME requires the most lines of code, but most of those lines are classed as boilerplate/setup code because of how the library requires users to write XML files.

Table 5.10. Quantitative comparison of ABM libraries by averages of benchmark measurements

Library	**Lines of Code**	**Control Flow Statements**	**Boilerplate LoC**	**Agent LoC**	**Space LoC**	**Model Design LoC**	**Model Management LoC**
MASS C++	**592.3**	**79.7**	**164**	**188.9**	**220.4**	**409**	**89**
RepastHPC	789.9	68.1	175.25	297.6	71.9	369	320
FLAME	826.0	41.4	327	217.4	30.9	248	284

Table 5.11 and Table 5.12 contain the findings of the manual code review we performed on each benchmark implementation. In Table 5.11, we assessed how each ABM library approaches each benchmark model and therefore different computational models. In Table 5.12, we determined the relative strengths and weaknesses of each parallel ABM library for different aspects of agent-based modeling. The autonomy of the Agents and Places means that MASS C++ requires less code for managing the model, though the Agents and Places themselves require more code. RepastHPC's Observer allows the user to control the entire model and its agents more directly than MASS and FLAME, but this also means users must write more model management code. FLAME overall is the most limited in programmability because space cannot be directly defined, only mimicked by agents and because agent behavior is defined by states and message board communication.

Table 5.11. Comparison of each ABM library's approach to each benchmark models

Benchmark Model	**Computational Model**	**MASS C++**	**RepastHPC**	**FLAME**
Bail-In, Bail-Out	Inter-Group Communication	FinancialMarket Places act as shared memory for agents to communicate across model	Observers exchange Messenger objects with each other	All agents exchange transaction messages between each other.
Brain Grid	Agent Jump	Agent keeps moving with each neuron tip and hops back to its soma	Agents placed into the corresponding Observer.	Place agents mimics a 2D mesh and arbitrates neurons.
Game of Life	Cellular Automata	2D Places with von Neumann neighborhood.	2D Patch with von Neumann neighborhood	Each agent mimics a cell with von Neumann neighborhood.
MATSim	Agent Movement on Network	Agents migrate, with collision arbitration by a neighbor place	Observer maintains a traffic road and moves agents	Place agents mimics a road network and arbitrates cars.
Social Network	Network of Agents	1D array of Person Places, each with friendship lists	Each Agent maintains friendship lists.	Each Agent maintains friendship lists.
Tuberculosis	Agent Movement over 2D Space	Agents migrate, with collision arbitration by destination place	2D LungPlace patch adds to and takes agents from neighbors	Place Agents mimic a 2D mesh and arbitrates mobile Agents.
Virtual Design Team	Intra-Group Communication	Communication between Agents per Place via Place variables	Task object passed from one Agent to another by Observer	Task Agent communicates for Engineer Agents.

Table 5.12. Qualitative comparison of library capabilities

Metrics	**MASS C++**	**RepastHPC**	**FLAME**
Computational Model	+ Agents have autonomy of spawning, terminating, and migrating.	+ Good global space view. – Agents are centrally controlled by observers	– Space mimicked by agents have substantial semantic gaps.
Entities (space and/or agents)	+ Agents and places are separated. Quite intuitive.	+ Agents and places separated. Its shared space gives best global view.	– Space must be mimicked by agents
Space structure	– Network must be emulated by 1D Places.	+ Shared spaces such as patch and shared context are available	– Agents must maintain adjacency lists. Difficult to view the structure.
Agent population	– Agents spawned, and then must hop to their initial place.	+ Observer populates agents at initial location	– No agent population on a place. Agents must be associated with place agents
Multi-classes of agents	+ As many classes of agents as needed. Multiple places are allowed.	+ As many classes of agents as needed	+ As many classes of agents as needed.
Runtime agent management	+ Agents have behavioral autonomy, all supported by the library.	– Observer must control agents from the hawk's viewpoint	- State machines, message boards

Inter-agent communication	+ exchangeAll to remote Places deliver messages to agents in a remote place.	– Observers exchange Messenger objects with each other: can’t create teams.	- Indirect communication, based on message types.
Synchronization (lock/unlock)	– Agents on the same place must use lock/unlock at user level.	+ Observers centrally control agents	+ Each place agent behaves as a critical section.
Collision control	+ A destination place must arbitrate agent migration with exchangeBoundary.	+ Observers centrally control agent collisions	- Place agents must arbitrate other mobile agents through message exchange.

Chapter 6. CONCLUSION

6.1 SUMMARY

We have performed a comparative analysis of three parallel agent-based modeling libraries: MASS C++, RepastHPC, and FLAME. This analysis compared the performance and programmability of the libraries using a set of seven benchmark programs that cover problems within the social, behavioral, and economic sciences. We implemented each benchmark program three times, once for each of the three libraries, for a total of 21 programs. To compare performance, we defined two sets of parameters, one tuned for MASS C++ and one tuned for FLAME. We executed each of the benchmark implementations with the two sets of parameters and obtained average execution times. To compare programmability, we used static code analysis and manual code review to obtain quantitative and qualitative metrics for each benchmark implementation and extrapolated those findings to assess the ABM libraries.

Regarding the performance of the ABM libraries, we found that MASS C++ had the lowest average execution times across its benchmarks compared to RepastHPC and FLAME. The multi-threaded executions of MASS C++ generally had even lower average execution times, demonstrating another advantage of MASS C++. RepastHPC was able to handle the parameter set that was based on MASS C++ but was slower in all but one benchmark and more inconsistent overall. However, some benchmark implementations written for RepastHPC were incomplete and thus could not be completely compared. On the other hand, FLAME was entirely unable to handle the MASS C++ parameter set, and this necessitated a second parameter set based on FLAME's capabilities. The benchmarks for MASS C++ and RepastHPC executed this second parameter set faster than FLAME. Therefore, FLAME had definitively the worst performance of the three libraries.

In terms of programmability, each ABM library presents its own unique benefits and downsides which ABM researchers may prioritize differently. Therefore, no library can be said to be definitively the "most programmable" as this will depend on the user's needs and abilities. However, the quantitative and qualitative data can broadly indicate the strengths and weaknesses of each library. MASS C++ was found to generally require the fewest lines of code to program the benchmark models. Users must define agents and places in greater detail because of the focus on agent autonomy, but this autonomy means less code dedicated to managing the model. RepastHPC generally requires more lines than MASS C++ but fewer than FLAME. The RepastHPC Observer offers an improved view and control over the model at the cost of some autonomy. FLAME requires the most lines of code, but most of those lines are boilerplate/setup code from the XML file format that FLAME uses. FLAME overall is the most limited in programmability because space cannot be defined and must be mimicked by agents, and because agent behavior is defined by states and message board communication.

6.2 Reflection

The results of the performance and programmability comparison are promising for the Distributed Systems Lab and the MASS project. They indicate that researchers seeking to develop agent-based models with parallelization can use MASS C++ as a valid alternative to RepastHPC and FLAME which have been previously used in the industry.

Four different students in the Distributed Systems Lab implemented the benchmark programs, each at different times. This resulted in inconsistencies in coding style and documentation in the benchmark implementations. We attempted to minimize these inconsistencies before the performance and programmability measurements in case they affected the results, but time constraints limited this process. Ultimately, this resulted in two RepastHPC benchmark implementations which were incomplete or flawed (Social Network and VDT). In retrospect, some of the time spent on developing tools for the performance measurements should have been redirected towards ensuring code correctness and code consistency.

6.3 FUTURE DEVELOPMENT AND STUDY

The tools and methods used for this study can be adapted for future studies of the same libraries, or the studies can be expanded to include other parallel ABM libraries. In addition, measuring resource usage like CPU usage, memory usage, and network traffic while executing these models would provide more detailed and actionable performance metrics. Another useful way to utilize the existing tools and methods would be to determine the exact spatial and temporal limits of these ABM libraries. For the sake of limiting the time spent on measurements, we used parameters that reached a target execution time of twenty minutes on a single node. However, all three libraries can run models that are significantly larger, though they would take an extremely long time to complete.

The benchmark programs designed and implemented for this study can be further refined in terms of consistency and optimization to act as a testing suite for the three ABM libraries. For example, future researchers in the Distributed Systems Lab can use the MASS C++ benchmark implementations to measure performance improvements as they develop MASS C++.

BIBLIOGRAPHY

[1] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. O'Hare. 2017. Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Comput Sci Rev* 24, (May 2017), 13–33. DOI:https://doi.org/10.1016/J.COSREV.2017.03.001

[2] Chee Siang Ang and Panayiotis Zaphiris. 2009. Simulating social networks of online communities: Simulation as a method for sociability design. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5727 LNCS, PART 2 (2009), 443–456. DOI:https://doi.org/10.1007/978-3-642-03658-3_48/COVER

[3] Argonne National Laboratory. Repast Simphony. Retrieved November 18, 2022 from https://repast.github.io/repast_simphony.html

[4] Eric Bonabeau. 2002. Agent-based modeling: Methods and techniques for simulating human systems. *Proc Natl Acad Sci U S A* 99, SUPPL. 3 (May 2002), 7280–7287. DOI:https://doi.org/10.1073/PNAS.082080899/ASSET/15AB2074-4729-491A-B9E6-292A9C40AE31/ASSETS/GRAPHIC/PQ0820808004.JPEG

[5] Christopher Bowzer, Benjamin Phan, Kasey Cohen, and Munehiro Fukuda. Collision-Free Agent Migration in Spatial Simulation.

[6] Dennis L. Chao, M. Elizabeth Halloran, Valerie J. Obenchain, and Ira M. Longini. 2010. FluTE, a Publicly Available Stochastic Influenza Epidemic Simulation Model. *PLoS Comput Biol* 6, 1 (2010), e1000656. DOI:https://doi.org/10.1371/JOURNAL.PCBI.1000656

[7] Nick Collier. 2013. Repast HPC Manual. (2013). Retrieved November 18, 2022 from http://boost.org

[8] Roshan D'Souza, Mikola Lysenko, Simeone Marino, and Denise Kirschner. 2009. Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. DOI:https://doi.org/10.1145/1639809.1639831

[9] John Emau, Timothy Chuang, and Munehiro Fukuda. A Multi-Process Library for Multi-Agent and Spatial Simulation *.

[10] Munehiro Fukuda. MASS: A Parallelizing Library for Multi-Agent Spatial Simulation. Retrieved November 5, 2022 from http://depts.washington.edu/dslab/MASS/index.html

[11] Martin Gardner. 1970. MATHEMATICAL GAMES The fantastic combinations of John Conway's new solitaire game "life." *Sci Am* 223, (1970), 120–123. Retrieved October 22, 2022 from http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm

[12] Thomas E Gorochowski. 2016. Agent-based modelling in synthetic biology. *Essays Biochem* 60, (2016), 325–336. DOI:https://doi.org/10.1042/EBC20160037

[13] Andreas Horni, Kai Nagel, and Kay W Axhausen. The Multi-Agent Transport Simulation MATSim edited by. DOI:https://doi.org/10.5334/baw

[14] Fumitaka Kawasaki. 2012. Accelerating large-scale simulations of cortical neuronal network development. (September 2012). Retrieved November 19, 2022 from https://digital.lib.washington.edu:443/researchworks/handle/1773/20913

[15] Peter Klimek, Sebastian Poledna, J. Doyne Farmer, and Stefan Thurner. 2015. To bail-out or to bail-in? Answers from an agent-based model. *J Econ Dyn Control* 50, (January 2015), 144–154. DOI:https://doi.org/10.1016/J.JEDC.2014.08.020

[16] Raymond Levitt. 2000. VDT Computational Emulation Models of Organizations: State of the Art and Practice. (November 2000).

[17] Andreu Moreno, Juan J Rodríguez, · Daniel Beltrán, Anna Sikora, · Josep Jorba, and · Eduardo César. 2019. Designing a benchmark for the performance evaluation of agent-based simulation applications on HPC. *J Supercomput* 75, (2019), 1524–1550. DOI:https://doi.org/10.1007/s11227-018-2688-8

[18] Alban Rousset, Bénédicte Herrmann, Christophe Lang, and Laurent Philippe. 2016. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Comput Sci Rev* 22, (November 2016), 27–46. DOI:https://doi.org/10.1016/J.COSREV.2016.08.001

[19] Jose L. Segovia-Juarez, Suman Ganguli, and Denise Kirschner. 2004. Identifying control mechanisms of granuloma formation during M. tuberculosis infection using an agent-based model. *J Theor Biol* 231, 3 (December 2004), 357–376. DOI:https://doi.org/10.1016/J.JTBI.2004.06.031

[20] Craig Shih. Benchmarking and Evaluating ABM Parallel-Programming Features in Social, Behavioral and Economic Sciences.

[21] Craig Shih, Caleb Yang, and Munehiro Fukuda. Benchmarking the Agent Descriptivity of Parallel Multi-Agent Simulators. Retrieved November 5, 2022 from http://depts.washington.edu/dslab/MASS/index.html

[22] The Multi-Agent Transport Simulation MATSim on JSTOR. Retrieved October 22, 2022 from https://www.jstor.org/stable/j.ctv3t5r7p

[23] NetLogo 6.3.0 User Manual. Retrieved October 22, 2022 from http://ccl.northwestern.edu/netlogo/docs/

[24] MASON Multiagent Simulation Toolkit. Retrieved October 22, 2022 from https://cs.gmu.edu/~eclab/projects/mason/

[25] Repast Suite Documentation. Retrieved November 18, 2022 from https://repast.github.io/repast_hpc.html

[26] MATSim.org. Retrieved November 19, 2022 from https://www.matsim.org/

APPENDIX A. DETAILED PERFORMANCE RESULTS

Appendix Table A-1. Bail-in, Bail-out simulation, average execution times in seconds for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**	**MASS C++ (4 threads)**	**RepastHPC**
1 node	1,203.913	1,714.182	6,812.577
2 nodes	1,193.632	1,819.960	4,686.141
4 nodes	934.339	1,610.938	2,666.279
8 nodes	1,058.749	1,705.560	1,493.997

Appendix Table A-2. Bail-in, Bail-out simulation, average execution times in seconds for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**	**RepastHPC**	**FLAME**
1 node	58.471	338.467	1,212.226
2 nodes	53.530	235.230	614.553
4 nodes	48.445	134.152	358.160
8 nodes	62.317	75.369	185.779

Appendix Table A-3. Brain Grid simulation, average execution times in seconds for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**	**MASS C++ (4 threads)**	**RepastHPC**
1 node	1,191.925	487.305	6,213.978
2 nodes	678.524	244.897	4,381.844
4 nodes	339.261	182.016	2,434.674
8 nodes	170.401	91.943	1,425.412

Appendix Table A-4. Brain Grid simulation, average execution times in seconds for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**	**RepastHPC**	**FLAME**
1 node	55.107	301.935	1,299.290
2 nodes	32.138	210.374	651.991
4 nodes	17.943	117.525	371.556
8 nodes	10.093	67.963	186.437

Appendix Table A-5. Game of Life simulation, average execution times in seconds for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**	**MASS C++ (4 threads)**	**RepastHPC**
1 node	1,163.152	393.635	4,647.147
2 nodes	654.490	199.043	2,897.910
4 nodes	328.800	133.070	1,605.113
8 nodes	165.606	69.896	751.137

Appendix Table A-6. Game of Life simulation, average execution times in seconds for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**	**RepastHPC**	**FLAME**
1 node	31.605	97.737	1,284.601
2 nodes	17.850	66.027	643.789
4 nodes	8.750	34.167	339.997
8 nodes	4.825	17.523	175.628

Appendix Table A-7. MATSim, average execution times in seconds for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**	**MASS C++ (4 threads)**	**RepastHPC**
1 node	1,154.488	640.961	956.689
2 nodes	646.178	N/A	509.679
4 nodes	383.815	N/A	293.210
8 nodes	359.572	N/A	219.274

Appendix Table A-8. MATSim, average execution times in seconds for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**	**RepastHPC**	**FLAME**
1 node	17.873	115.263	1,188.153
2 nodes	20.575	83.897	661.172
4 nodes	26.287	70.57	346.157
8 nodes	61.510	116.816	174.576

Appendix Table A-9. Social Network simulation, average execution times in seconds for the higher parameters

# Computing Nodes	**MASS C++**	**MASS C++**	**RepastHPC**

	(1 thread)	(4 threads)	
1 node	1,102.592	N/A	N/A
2 nodes	446.501	N/A	N/A
4 nodes	239.960	N/A	N/A
8 nodes	129.332	N/A	N/A

Appendix Table A-10. Social Network simulation, average execution times in seconds for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**	**RepastHPC**	**FLAME**
1 node	32.959	N/A	1,218.784
2 nodes	16.760	N/A	609.535
4 nodes	8.925	N/A	348.528
8 nodes	4.811	N/A	174.643

Appendix Table A-11. Tuberculosis simulation, average execution times in seconds for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**	**MASS C++ (4 threads)**	**RepastHPC**
1 node	1,096.603	421.006	2,308.043
2 nodes	603.602	197.195	1,184.286
4 nodes	325.989	116.044	588.439
8 nodes	164.175	59.250	289.893

Appendix Table A-12. Tuberculosis simulation, average execution times in seconds for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**	**RepastHPC**	**FLAME**
1 node	11.042	64.221	1,353.899
2 nodes	6.328	41.694	682.920
4 nodes	3.482	24.566	378.398
8 nodes	2.082	15.935	197.075

Appendix Table A-13. Virtual Design Team simulation, average execution times in seconds for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**	**MASS C++ (4 threads)**	**RepastHPC**
1 node	1,201.433	2,047.137	N/A
2 nodes	692.947	950.932	N/A
4 nodes	432.124	776.380	N/A

8 nodes	301.425	494.800	N/A

Appendix Table A-14. Virtual Design Team simulation, average execution times in seconds for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**	**RepastHPC**	**FLAME**
1 node	188.599	N/A	923.952
2 nodes	173.533	N/A	436.744
4 nodes	142.275	N/A	309.336
8 nodes	147.935	N/A	362.019

APPENDIX B. STANDARD DEVIATION AND VARIANCE

Appendix Table B-1. Bail-in, Bail-out simulation, standard deviation and variance for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**		**MASS C++ (4 threads)**		**RepastHPC**	
1 node	5.970088999	35.64196266	5.253119675	27.59526632	6.831736309	46.672621
2 nodes	52.36718442	2742.322004	4.376489403	19.15365949	8.787145744	77.21393033
4 nodes	12.11449717	146.7610417	27.10323111	734.5851368	4.817508312	23.20838633
8 nodes	22.66815992	513.8454739	21.62904528	467.8155999	8.154871264	66.50192533

Appendix Table B-2. Bail-in, Bail-out simulation, standard deviation and variance for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**		**RepastHPC**		**FLAME**	
1 node	0.5652411451	0.3194975521	2.101749509	4.417351	1.689101635	2.853064333
2 nodes	0.155705755	0.02424428213	1.095840013	1.200865333	1.222392327	1.494243
4 nodes	0.133563036	0.01783908459	0.636858697	0.405589	0.3052982149	0.093207
8 nodes	0.2141813947	0.04587366983	0.3854983787	0.148609	0.01096965511	0.0001203333333

Appendix Table B-3. Brain Grid simulation, standard deviation and variance for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**		**MASS C++ (4 threads)**		**RepastHPC**	
1 node	4.412164685	19.4671972	0.6221648791	0.3870891368	7.145312893	51.05549633
2 nodes	2.488905298	6.194649582	1.309405783	1.714543504	18.10771356	327.8892903
4 nodes	1.080930051	1.168409776	4.963443476	24.63577114	5.424810534	29.42856933
8 nodes	0.6246955695	0.3902445546	4.474295412	20.01931943	5.749930811	33.06170433

Appendix Table B-4. Brain Grid simulation, standard deviation and variance for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**		**RepastHPC**		**FLAME**	
1 node	0.4629389292	0.2143124522	2.885132984	8.323992333	3.171322174	10.05728433
2 nodes	0.03497020575	0.00122291529	1.76594479	3.118561	0.7149442869	0.5111453333

4 nodes	0.1961560423	0.03847719292	3.424716193	11.728681	0.1311830782	0.017209
8 nodes	0.2146711772	0.0460837143	0.5553488393	0.3084123333	0.3115300949	0.097051

Appendix Table B-5. Game of Life simulation, standard deviation and variance for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**		**MASS C++ (4 threads)**		**RepastHPC**	
1 node	9.106575763	82.92972213	0.6761025061	0.4571145988	14.71000453	216.3842333
2 nodes	6.416661219	41.1735412	1.108475627	1.228718217	11.38415126	129.5989
4 nodes	4.993694924	24.936989	7.453765516	55.55862036	9.03848623	81.69423333
8 nodes	1.505706589	2.267152331	10.41005865	108.369321	43.55155604	1896.738033

Appendix Table B-6. Game of Life simulation, standard deviation and variance for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**		**RepastHPC**		**FLAME**	
1 node	0.1638079534	0.02683304561	0.5650073746	0.3192333333	60.27898545	3633.556086
2 nodes	0.05108569897	0.002609748639	0.6213158081	0.3860333333	7.029246712	49.41030933
4 nodes	0.02968798101	0.0008813762163	0.1795364401	0.03223333333	4.45150615	19.815907
8 nodes	0.05551084486	0.003081453897	0.1193035345	0.01423333333	4.00778559	16.06234533

Appendix Table B-7. MATSim simulation, standard deviation and variance for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**		**MASS C++ (4 threads)**		**RepastHPC**	
1 node	1.620669373	2.626569218	2.108060172	4.443917691	0.4218850554	0.177987
2 nodes	4.699179335	22.08228642	N/A	N/A	1.085740761	1.178833
4 nodes	2.110590892	4.454593915	N/A	N/A	0.5664594719	0.3208763333
8 nodes	1.427274454	2.037112367	N/A	N/A	1.9705687	3.883141

Appendix Table B-8. MATSim simulation, standard deviation and variance for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**		**RepastHPC**		**FLAME**	
1 node	0.1648442855	0.02717363845	0.04398105653	0.001934333333	13.21586661	174.6591303
2 nodes	0.2754138019	0.07585276225	0.1694963126	0.028729	10.42059619	108.588825
4 nodes	1.059111423	1.121717007	0.030022214	0.0009013333333	3.951618951	15.61529233
8 nodes	2.410084464	5.808507123	2.211415911	4.890360333	8.204870444	67.319899

Appendix Table B-9. Social Network simulation, standard deviation and variance for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**		**MASS C++ (4 threads)**		**RepastHPC**	
1 node	6.66119973	44.37158184	N/A	N/A	N/A	N/A
2 nodes	1.23076902	1.51479238	N/A	N/A	N/A	N/A
4 nodes	1.799761904	3.239142912	N/A	N/A	N/A	N/A
8 nodes	1.118547032	1.251147464	N/A	N/A	N/A	N/A

Appendix Table B-10. Social Network simulation, standard deviation and variance for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**		**RepastHPC**		**FLAME**	

1 node	0.05737695302	0.003292114737	N/A	N/A	0.1512789917	0.02288533333
2 nodes	0.07082887327	0.005016729289	N/A	N/A	0.2204699526	0.048607
4 nodes	0.04779433433	0.002284298394	N/A	N/A	0.4361299501	0.1902093333
8 nodes	0.007991702697	0.000063867312	N/A	N/A	0.1882454072	0.03543633333

Appendix Table B-11. Tuberculosis simulation, standard deviation and variance for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**		**MASS C++ (4 threads)**		**RepastHPC**	
1 node	23.65394405	559.5090692	47.69546615	2274.857491	3.218975199	10.36180133
2 nodes	3.685092543	13.57990705	0.9805936773	0.9615639599	6.271533385	39.332131
4 nodes	1.81599069	3.297822187	0.470364652	0.2212429059	2.728694254	7.445772333
8 nodes	0.9522600244	0.9067991541	0.1544113453	0.02384286354	3.478106525	12.097225

Appendix Table B-12. Tuberculosis simulation, standard deviation and variance for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**		**RepastHPC**		**FLAME**	
1 node	0.08314539536	0.00691315677	1.683804027	2.835196	7.91522137	62.65072933
2 nodes	0.01891779084	0.0003578828103	1.054995893	1.113016333	4.986712344	24.8673
4 nodes	0.03942357807	0.001554218508	0.725265009	0.5260093333	22.19212879	492.4905803
8 nodes	0.03769246649	0.00142072203	0.7717592457	0.5956123333	4.373916132	19.13114233

Appendix Table B-13. Virtual Design Team simulation, standard deviation and variance for the higher parameters

# Computing Nodes	**MASS C++ (1 thread)**		**MASS C++ (4 threads)**		**RepastHPC**	
1 node	2.214982268	4.906146449	15.87306039	251.9540462	N/A	N/A
2 nodes	1.313847138	1.726194301	7.32752875	53.69267757	N/A	N/A
4 nodes	0.4167677557	0.1736953622	4.294514685	18.44285638	N/A	N/A
8 nodes	0.9343829168	0.8730714353	4.090750121	16.73423655	N/A	N/A

Appendix Table B-14. Virtual Design Team simulation, standard deviation and variance for the smaller parameters

# Computing Nodes	**MASS C++ (1 thread)**		**RepastHPC**		**FLAME**	
1 node	1.052885528	1.108567935	N/A	N/A	49.45034435	2445.336556
2 nodes	0.1599111162	0.02557156508	N/A	N/A	50.52742452	2553.020629
4 nodes	0.1305386936	0.01704035052	N/A	N/A	28.22941121	796.8996573
8 nodes	0.02227391102	0.0004961271123	N/A	N/A	103.3355151	10678.22868