Performance and Programmability Comparison of MASS and MadMPI

Kyryll Kotyk

CSS 497 Term Report
Spring 2026

University of Washington Bothell

Munehiro Fukuda

Table of Contents

Abstract 4
1. Introduction 5
2. Background 8
2.1 Distributed Runtime Models 8
2.2 MadMPI 8
2.3 MASS C++ 8
2.4 Terminology Used in This Report 9
2.5 Evaluation Terms 10
3. Related Work 11
3.1 Existing HPC Benchmark Suites 11
3.2 Prior MASS C++ Evaluations 11
3.3 MadMPI and NewMadeleine Work 11
3.4 Position of This Project 12
4. Benchmarks and Implementations 13
4.1 Heat3D 13
4.2 DGEMM 14
4.3 Graph Motif Search 15
4.4 Bail-In/Bail-Out 16
4.5 SugarScape 18
5. Experiment Setup and Methodology 20
5.1 Execution Environment 20
5.2 Configuration Strategy 20
5.3 Timing Methodology 20
5.4 Correctness and Validation 21
5.5 Programmability Evaluation 21
5.6 Fairness and Limitations 21
6. Performance 22
6.1 Heat3D 24
6.2 DGEMM 25
6.3 Graph Motif Search 27
6.4 Bail-In/Bail-Out 28
6.5 SugarScape 29
6.6 Performance Summary 30
7. Programmability 31
7.1 Overall Results 33
7.2 Heat3D 34

7.3 DGEMM........ 34
7.4 Graph Motif Search........ 34
7.5 Bail-In/Bail-Out........ 34
7.6 SugarScape........ 35
8. Conclusion........ 36
Bibliography........ 37

Abstract

Choosing the right runtime for distributed computing applications in C++ requires evaluating both performance and programmability. This report compares MASS C++ and MadMPI on equivalent implementations in five benchmarks: Heat3D, DGEMM, Graph Motif Search, Bail-In/Bail-Out, and SugarScape. They cover structured stencil computation, dense matrix multiplication, irregular graph analysis, phased financial simulation, and agent-based spatial modeling. MASS represents a higher-level model based on Places and Agents, while MadMPI represents a lower-level message-passing model built around explicit communication.

The implementations were evaluated on the Hermes cluster using wall-clock time, speedup, correctness checks, and programmability metrics. Performance results show that neither runtime is always faster. MadMPI had better peak performance in Heat3D, DGEMM SUMMA, Bail-In/Bail-Out, and SugarScape, where explicit communication control was valuable. MASS performed better in DGEMM Cannon's algorithm and Graph Motif Search, which shows that its structure can still compete or be faster when the workload fits its model or communication is less dominant than computation.

Programmability followed an opposite trend. MASS was generally better in terms of programmability, as the implementations were easier to organize. At the same time, MadMPI was very close behind MASS as it was more compact, but required more manual handling.

Overall, there is a clear tradeoff present. MadMPI is usually better for performance-critical communication, and MASS is usually better for structured simulations where maintainability and clarity matter more.

1. Introduction

Large scientific simulations and applications that require high amounts of data often require more computation and memory than a single machine can provide. HPC (High-Performance Computing) addresses this need by distributing the work across multiple machines (nodes), which allows the problem to be divided into smaller pieces executed in parallel. However, achieving the best parallel performance is not just a hardware problem. Rather, the programming model used greatly affects the efficiency, performance, and difficulty of writing, debugging, and maintaining the program.

A common model in distributed computing is the standardized Message Passing Interface (MPI). Hammond et al. state that "MPI is the most widely used interface for high-performance computing (HPC) workloads" [1, p. 1], so it is an important baseline for evaluating other runtimes and models. However, MPI's strength also comes from its low-level control. As Basili et al. describe it, MPI is "a message-passing library where the programmer explicitly specifies all communication" [2, p. 30]. In this model, the programmer is directly in control of how data is divided across processes, when messages are sent, results are received, and where synchronization happens.

MPI's low-level approach gives the programmer a lot of freedom and control over data movement, allowing for unique optimizations per workload. At the same time, this extra control has a cost: the implementation must manage process ownership, routing, communication timing, and more.

MASS C++ represents a much different design, where instead of exposing the messages directly, it uses higher-level abstractions such as Places and Agents. Places represent distributed state, often arranged over a grid, while Agents represent mobile objects that can act and move through that space. Computation is expressed through bulk-synchronous phases, where operations are invoked across the distributed objects, and the runtime handles the communication and synchronization. This can make MASS programs easier to structure for simulations with clear phase boundaries or spatial organization. However, this abstraction also limits the explicit control the programmer has and may introduce overhead when the workload doesn't match the Place and Agent model naturally.

This project compares MASS C++ and MadMPI as two contrasting approaches to distributed C++ programming. The comparison focuses on the tradeoff between programmability and performance. MASS is expected to be at an advantage when the abstractions match the workload, especially for structured simulations and agent-based programs. MadMPI is expected to perform well when explicit communication control is important, especially when message timing, process layout, and communication overlap are tunable. The central question is where

these expectations match the results, where they don't, and how the answer changes for the workload types used.

To make the comparison meaningful, multiple benchmarks with different workloads and stress points were used. The benchmarks include Heat3D, a structured stencil computation in 3D; DGEMM, dense matrix multiplication implemented separately with SUMMA and Cannon's algorithm; Graph Motif Search, an irregular graph workload; Bail-In/Bail-Out, a phased financial simulation with a focus on frequent and variable communication with many factors; and SugarScape, an agent-based model with movement and competition for resources on a grid. Together, these benchmarks test several important patterns in distributed computing: Regular neighbor exchange, broadcasting, shifting, heavy computation, irregular search, phased variable communication, and dynamic agent movement.

These workloads were not chosen only to test the runtimes in isolation. They were provided by the UWB School of Business because they represent common application patterns that appear in business and scientific computing. Heat3D and DGEMM represent scientific modeling and numerical computation, Graph Motif Search represents network analysis, Bail-In/Bail-Out models financial interaction and systemic risk, and SugarScape represents agent-based resource competition and emergent behavior. Because these patterns are relevant outside the project itself, the comparison is meant to show how MASS and MadMPI behave on realistic workload types, not just artificial test cases.

The implementations are designed to preserve the same high-level behavior in both runtimes. Each version uses the same goals, inputs, validation, and structure wherever practical. Without this, the comparison would measure differences in algorithm design, rather than the differences in MASS and MadMPI. Some implementation details differ, as the runtimes use different models; however, these differences are part of the comparison. They show what each system makes easy, difficult, and what code structure it encourages.

This comparison uses MASS implementations originally developed by Ahmed Bera Pay as part of his Master's thesis [12]. For this report, those implementations went through finalization and cleanup, including organization into the final project structure and preparation for consistent evaluation alongside the MadMPI versions.

To compare performance, wall-clock time and speedup vs one execution unit were measured for different node and per-node execution units configurations. The measurements show how the runtimes scale as more units are added, and whether the additional parallelism helps or harms the performance.

Programmability was evaluated through extracted code metrics such as the code size, complexity, function size, and more. This aspect is important, as a faster implementation may not

always be the best choice, as the code may be more complex, less maintainable and readable. At the same time, a cleaner implementation is not acceptable either if the abstractions prevent scaling or hold back performance significantly.

Due to the nature of MadMPI and MASS, the expected tradeoff is that the former should provide better control over performance-critical communication, while the latter should provide more maintainable implementation in workloads that match its model. The results of this comparison are intended to clarify that tradeoff across multiple workload types. Rather than treating performance and programmability as separate concerns, both are evaluated together to determine which runtime is better suited to which kind of application.

2. Background

2.1 Distributed Runtime Models

There is more to compare between MASS and MadMPI than the syntax. They encourage different thinking about the distributed program being written. MadMPI is closer to writing the communication directly, and MASS gives the programmer a higher-level structure built around distributed objects and bulk synchronization. Because of this, the same workload can look very different depending on which runtime is used.

In a distributed program, the data is split across multiple nodes. Each node owns only part of the full problem, so work that only uses local data is usually much easier to scale than work that needs to send to and receive data from other nodes. When remote data is needed, the program must communicate, and that communication has a cost. So, the performance doesn't only depend on how much computation there is to do, but also on how the data is divided, the frequency of communication, and synchronization.

2.2 MadMPI

MadMPI uses an explicit message-passing style. Each process runs the same program, but works on a different part of the data. This is called SPMD, or Single Program Multiple Data. In this model, the code is usually written based on the process rank, range of ownership, send/receive operations, and collectives. For example, when matrix multiplication needs row and column broadcasts, the programmer is responsible for creating that communication structure.

This makes MadMPI flexible, as there is direct control over communication, so there aren't many restrictions on what the programmer can do. Data can be packed as the program needs, messages can be grouped or sent by themselves, communication can be overlapped with computation, and the process layout can be changed if needed. This is useful when performance depends on a very specific pattern. At the same time, this control increases the complexity of the program, requiring handling ownership, buffers, message orders, barriers, and correctness details directly, rather than letting the runtime abstract these away.

2.3 MASS C++

MASS C++ takes a different approach. Instead of making the programmer write every message directly, MASS uses Places and Agents as an abstraction. Places are distributed objects that are usually arranged as a distributed grid, while Agents are mobile objects that act and live on Places and move through the distributed space.

MASS programs are usually written as a sequence of phases. A phase may call a method across many Places or Agents, exchange boundary information, move agents, or apply updates. Importantly, the programmer is usually describing what the objects should do, and MASS handles the communication and synchronization needed to make that happen. When a workload has a clear update structure, this can make the code easier to follow as compared to the more manual MadMPI implementation.

MASS's main advantage is that it can make distributed simulations easier to organize and follow. There is no need to manually write individual sends and receives, and the bulk-synchronous structure makes the order of operations easier to reason about and track. If one phase finishes before the next begins, then it is clearer which state belongs to the current and the next timesteps. This is useful when correctness depends on the distributed state being updated in a consistent order.

Another practical advantage is that MASS can reduce some of the mistakes that are easier to make in lower-level message-passing code. As mentioned earlier, MadMPI requires the programmer to manage communication and related details, so correctness and performance are heavily dependent on how well they are written. Basili et al. describe this skill gap clearly, noting that "Teaching people to use MPI is not very hard," but "Teaching people to write MPI effectively ... is extremely difficult" [2, p. 35]. This shows that MadMPI has a high ceiling for expert optimization, and a less experienced programmer is much closer to its performance and efficiency floor. MASS does not necessarily have the same peak performance ceiling, but its abstractions can make it easier to reach a good implementation by guiding the program toward structured phases, consistent synchronization, and runtime-managed data movement.

The tradeoff is that MASS gives less direct control over the communication details. If the workload naturally matches Places, Agents, and phase-based execution, this may not be a problem. However, when the workload needs a very specific communication pattern, the programmer has to express that pattern through the MASS model. This can add extra structure, extra packing, or extra routing logic that would be more direct in MadMPI.

2.4 Terminology Used in This Report

A node refers to a physical machine used in the cluster. Execution units per node refers to how many processes ran on each node for MadMPI, and the number of threads that ran on each node for MASS. Total execution units refers to the node count multiplied by the execution units per node. For convenience, instances per node (ipn) may be used in place of execution units per node (eupn). When referring to MadMPI only, processes per node (ppn) may be used as well. This wording is used because MASS and MadMPI do not expose parallelism in the same way.

MASS stands for Multi-Agent Spatial Simulation. DGEMM refers to double-precision general matrix multiplication, and SUMMA refers to the Scalable Universal Matrix Multiplication Algorithm. Bail-In/Bail-Out may also be shortened to BIBO. CSR means Compressed Sparse Row, which is the graph storage format used for efficient adjacency access in Graph Motif Search. FNV-1a refers to the Fowler-Noll-Vo hash variant used for checksum-based validation. LCOM means Lack of Cohesion of Methods, a code metric used to estimate how closely related the methods in a class are.

2.5 Evaluation Terms

Performance is measured using wall-clock time and speedup. Wall-clock time measures the main computation, while speedup compares each run to the one node, one execution unit baseline for the same runtime. Programmability is evaluated through implementation structure, including code size, function size, complexity, and runtime-specific communication or synchronization code. These metrics do not capture all programmer effort, but they provide a practical comparison of implementation complexity.

Overall, this background leads to the main question of the report: for the real-world workload patterns provided by the UWB School of Business, when does MadMPI's extra communication control lead to better performance, and when does the higher-level structure of MASS lead to a cleaner implementation without giving up too much speed?

3. Related Work

3.1 Existing HPC Benchmark Suites

Many existing suites are commonly used to evaluate parallel performance, but they usually answer a different question than this project. The NAS Parallel Benchmarks are a standard example, focusing on kernels and simulated applications that represent large scientific workloads [3]. PolyBench, Rodinia, and BOTS also provide useful workloads for evaluating kernels, compiler behavior, heterogeneous systems, and task-based parallelism [4]-[6]. These suites are great for measuring performance, but they usually assume the programming model instead of comparing programming models directly. They also tend to focus on general workload collections, rather than asking how different runtimes behave on a set of applied business and science workloads chosen for a specific project context.

Due to that, they do not answer what this report asks. This project is more than just measuring performance. It is also asking how the code's structure, communication logic, and control change based on the runtime, and what the result of that is on programmability and performance.

3.2 Prior MASS C++ Evaluations

Most prior MASS C++ work focuses on agent-based or spatial simulations, where the Places and Agents model is a natural fit. Fukuda et al. compare MASS C++ with FLAME and Repast HPC across multiple agent-based models, measuring both performance and programmability metrics like lines of code, boilerplate, LCOM, and scaling behavior [7]. This is close to the goals of this report because it treats programmability as part of the evaluation, rather than only reporting runtime performance.

However, that work is only within the agent-based library space. It does not compare MASS against an MPI implementation on different workload types. Kipps, Kim, and Fukuda also show that the MASS model can be applied to graph motif search, comparing MASS graph traversal with MPI-based and MASS place-based versions [8]. That is relevant as motif search is less regular than a simple grid simulation, but it is still a narrower comparison than the one used here.

3.3 MadMPI and NewMadeleine Work

Related work on MadMPI and NewMadeleine is mostly focused on communication performance and MPI support, rather than programmability comparison. MadMPI is described by the PM2 project as the MPI interface for NewMadeleine, allowing MPI applications to use the NewMadeleine communication engine [9].

NewMadeleine itself has been studied as a communication scheduling engine for high-performance networks [10]. Later work also evaluates its scalability with many communication threads and describes MadMPI as a thin MPI layer over NewMadeleine [11]. These works do not focus on the main tradeoff between control and complexity studied here.

3.4 Position of This Project

This project connects parts of the previous work, but uses a different comparison. Existing suites provide useful workload patterns, MASS studies show the value of higher-level simulation abstractions, and MadMPI/NewMadeleine work shows the value of explicit communication control. What is missing is a focused comparison between MASS and MadMPI on multiple workload types, using both performance and programmability as evaluation goals.

Another difference is the applied purpose of the workloads used in the comparison. Prior work evaluates either general HPC kernels, agent-based simulation, or communication systems separately. So, they do not connect the runtime behavior with a mixed set of applied business and science workload types. Here, the applicability is part of the comparison, as the question is not just the performance of the two libraries but how useful the models are for the specific patterns that reflect real application needs.

Therefore, the goal is not to replace existing suites or prior benchmarking of MASS and MadMPI. Rather, this project introduces a focused comparison of two distributed C++ models on shared, applied workload patterns. The contribution is in measuring how each model affects both the performance results and the implementation structure, especially when the workload has real business or science relevance instead of being only a synthetic test case.

4. Benchmarks and Implementations

Before describing each implementation, Table 1 provides a compact overview of the five workloads, their applications, communication patterns, and decomposition strategies used. This helps show why the workloads are different enough to test multiple behaviors in distributed computing.

Table 1. Benchmark Overview

Benchmark	Domain	Communication Pattern	Decomposition
Heat3D	Heat diffusion / CFD	Nearest-neighbor halo exchange (6 face directions per timestep)	3D block ($P_x \times P_y \times P_z$)
DGEMM – SUMMA	Dense linear algebra	Row and column panel broadcasts at each step	2D process grid ($P_r \times P_c$)
DGEMM – Cannon's	Dense linear algebra	Cyclic nearest-neighbor panel shifts (left/up)	Square 2D grid ($P \times P$ only)
Graph Motif Search	Network analysis	Graph broadcast at init; count reduction at end; none during search	Root-vertex range per execution unit
Bail-In/Bail-Out	Financial simulation	Phased all-to-all message exchange; one barrier per phase	Contiguous entity ID ranges
SugarScape	Agent-based model	Boundary halo exchange; centralized target selection and conflict resolution	Horizontal row slabs

The subsections describe how each benchmark was implemented in MASS and MadMPI, focusing on the implementation details that affect performance, communication, or programmability.

4.1 Heat3D

Heat3D models heat diffusion in a 3D grid via repeated stencil updates. Each execution unit owns a part of the grid, and at each timestep, the neighboring boundary values are exchanged before the next update.

In MASS, the grid is divided into Places. Each Place has a chunk of the global grid, including the boundary information needed for neighbor exchange. It is implemented as a phase pipeline, where the boundary faces are packed, then exchanged through MASS communication, received into halo regions, and used for stencil updates. Unlike MadMPI, this implementation also includes sender coordinate information, so each received face can be placed into the right halo region. This is necessary due to MASS's communication system.

In MadMPI, the global grid is split into a 3D block decomposition based on rank. Each process allocates owned cells plus ghost layers. Neighbor ranks are computed from the 3D decomposition, and six halo directions are exchanged with specific message tags. The implementation uses persistent send and receive requests, so the halo communication setup is only created once and reused per timestep. When possible, the interior computation is performed while halo communication is still in progress, and the boundary computation is completed after the halo data arrives.

Table 2 summarizes the main differences in the Heat3D implementation.

Table 2. Heat3D Implementation Comparison

Dimension	MASS C++	MadMPI
Grid decomposition	Places dimensions; each Place owns one contiguous grid chunk	3D block decomposition by MPI rank ($P_x \times P_y \times P_z$)
Halo exchange	placeExchangeAll with a sender-coordinate header on each face message	Persistent `MPI_Send_init` / `MPI_Recv_init`; restarted each timestep with `MPI_Startall` / `MPI_Waitall`
Face identification	Four-integer coordinate header; the receiver infers direction from the coordinate difference	Direction is encoded in the MPI tag; no separate header is needed
Synchronization	Implicit bulk-synchronous barrier at each phase	`MPI_Waitall` at each timestep
Compute / communication overlap	None; faces are packed, exchanged, then used for the stencil update	Interior stencil work runs while halo communication is in flight; boundary cells are computed after `MPI_Waitall`
Correctness criterion	Global temperature checksum compared with a serial reference	Global temperature checksum compared with a serial reference

4.2 DGEMM

DGEMM performs dense matrix multiplication. Both versions generate matrix values deterministically from indices instead of storing full global matrices. This keeps the focus on distributed computation and communication rather than file input reading and storage.

SUMMA and Cannon's algorithm were implemented separately, as they use different communication styles. SUMMA uses broadcasts, while Cannon's algorithm shifts matrix blocks between neighboring processes.

In MASS, the matrix work is mapped to a grid of Places. Each Place owns part of the output matrix and participates in panel communication through MASS operations. As MASS does not expose communicators directly, the implementation uses message metadata to identify the received panel.

In MadMPI SUMMA, processes are arranged into a 2D process grid. Row and column communicators are created with `MPI_Comm_split`. At each step, the needed A and B panels get broadcast across the row and column using non-blocking broadcasts. The implementation uses double buffering so the next panel communication can start while the current panel is being used.

In MadMPI Cannon, the process grid must be square. Each process owns a block of the output and starts with a skewed A and B panel. During each step, the process multiplies its current panels, then shifts A horizontally and B vertically using non-blocking sends and receives. For shifted messages to have consistent sizes, padded buffers are used.

Table 3 summarizes the DGEMM implementation details for both SUMMA and Cannon's algorithm.

Table 3. DGEMM Implementation Comparison

Dimension	MASS C++	MadMPI
Process grid	Places dimensions; node-granular only, so ipn does not subdivide the matrix	`MPI_Comm_split` creates row and column communicators
SUMMA panel broadcast	`placeExchangeAll` to all neighbors; a step-number header filters out panels not meant for the current row or column	Non-blocking `MPI_Ibcast` on row and column communicators
SUMMA double buffering	Not used	Yes; the next step's broadcast is posted while the current step computes
SUMMA overlap	None	The next broadcast overlaps the current computation
Cannon panel shift	`placeExchangeAll` with a step-number header; panels are padded to the maximum block size so message sizes stay uniform	`MPI_Isend` / `MPI_Irecv` on row and column communicators; the shift is posted before computation
Cannon overlap	None	The shift overlaps the current matrix multiply; `MPI_Waitall` completes after computation
Grid constraint	Requires a square sqrt(P) × sqrt(P) process grid; valid for ranks `{1, 4, 16}`	Requires a square sqrt(P) × sqrt(P) process grid; valid for ranks `{1, 4, 16}`
Correctness	Per-entry checksum of C against a serial reference using the same `valueAt(row, col, seed)` hash	Per-entry checksum of C against a serial reference using the same `valueAt(row, col, seed)` hash

4.3 Graph Motif Search

Graph Motif Search counts small graph patterns inside a larger graph. The irregularity comes from different root vertices leading to different amounts of search work.

In both runtimes, the graph is replicated rather than distributed by edges or vertices. The search space is divided by root vertex. So, each execution unit only searches its assigned root range, and communication is only needed for setup and result aggregation at the end.

In MASS, the graph is loaded into the MASS structure, and Places divide the root work. Search is invoked across Places, and each Place returns its local count. The final result is produced by collecting and summing those local counts.

In MadMPI, each process either loads the same input graph or generates the same deterministic graph. The graph is stored in CSR format for efficient adjacency checks. Each process receives a contiguous range of root vertices, runs the backtracking search locally, and then uses reductions to combine the raw embedding count and the slowest process time.

Table 4 summarizes the Graph Motif Search implementation. The main comparison is not graph distribution, but how each runtime partitions the search work and combines the final counts.

Table 4. Graph Motif Search Implementation Comparison

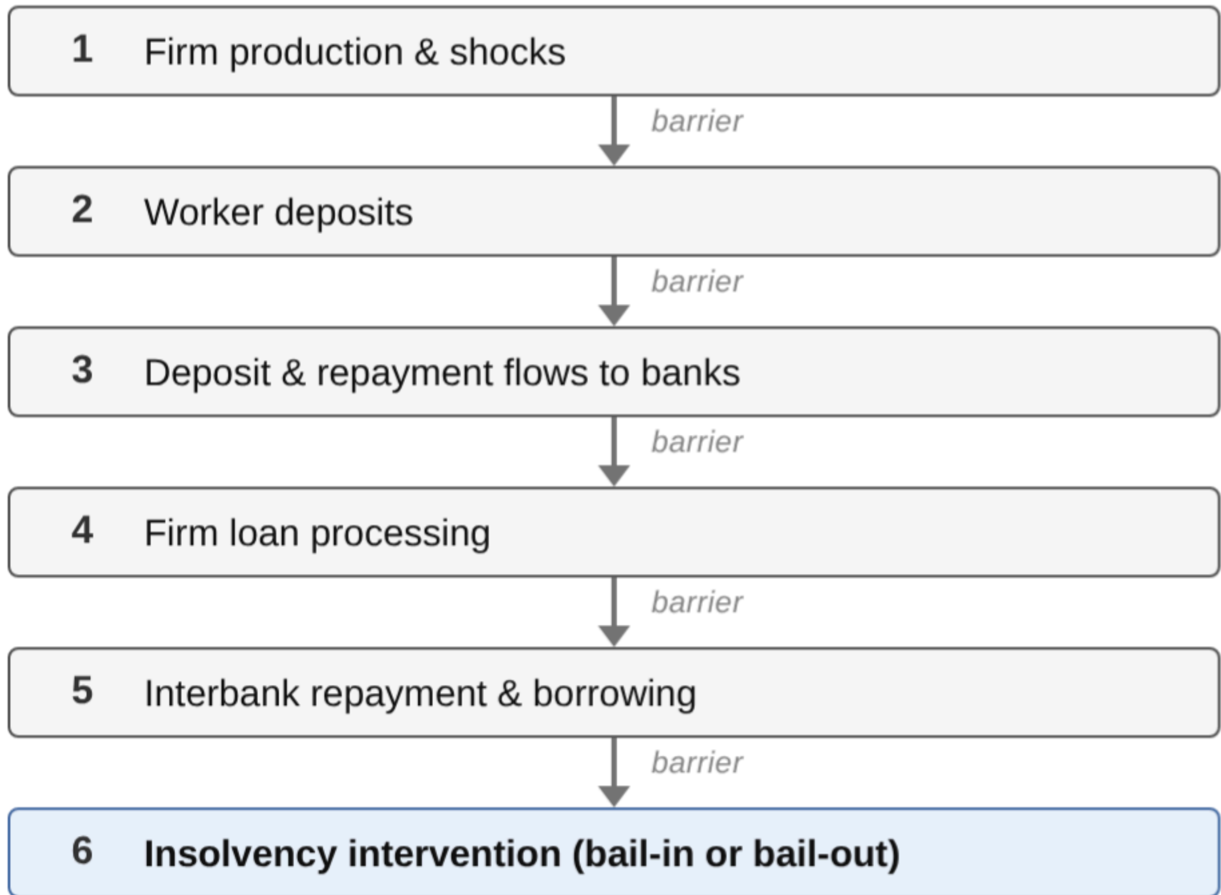
Dimension	MASS C++	MadMPI
Graph initialization	The host loads and serializes the graph, then broadcasts it to all Places through `callAll(init)`	Each process independently loads or generates the same graph from the same seed; no broadcast is needed
Graph storage	Unpacked into shared `CSR` arrays per process; access is mutex-guarded across local threads	`CSR` arrays are stored per process; no sharing across ranks
Work partition	Places divide root-vertex work across local threads	Each process owns one contiguous range of root vertices
Communication during search	None; all search work is local	None; all search work is local
Result aggregation	The host sums local counts returned by `callAll(getCount)`	`MPI_Reduce(SUM)` combines total count; `MPI_Reduce(MAX)` combines wall time
Correctness criterion	Total embedding count compared against a serial reference on the same graph	Total embedding count compared against a serial reference on the same graph

4.4 Bail-In/Bail-Out

Bail-In/Bail-Out is a financial simulation with banks, firms, and workers. Each timestep is divided into phases for firm activity, worker deposits, loan processing, interbank activity, and bail-in or bail-out intervention. The workload is communication-heavy as entities often interact with other non-local entities.

Figure 1 shows the ordered phase structure used in Bail-In/Bail-Out. It is important to illustrate, as the workload depends on strict phase ordering. Each phase must complete before the next starts, so synchronization is required.

Figure 1. Bail-In/Bail-Out Phase Order



In MASS, entity state is partitioned across Places, and each phase packs outgoing messages, exchanges them, and applies the received messages.

In MadMPI, each process owns ranges of banks, firms, and workers. Message routing is based on the owner process of the target. The implementation uses reusable per-destination message buckets and an all-to-all exchange for routed messages. Bank liquidity information needed for interbank borrowing is shared using `MPI_Allgatherv`.

Table 5 summarizes the Bail-In/Bail-Out implementation differences. MASS keeps the program closer to a phase-based simulation structure, while MadMPI makes the routing and synchronization more explicit.

Table 5. Bail-In/Bail-Out Implementation Comparison

Dimension	MASS C++	MadMPI
Entity partitioning	Entity state is distributed across Places by global ID	Contiguous global ID ranges per rank, using shared `biboOwnerRankFromGlobalId`
Message routing	Places pack per-destination byte buckets; these are exchanged through `GraphPlaces::exchangeNeighbors` and applied with `callAll(applyMethod)`	Per-destination typed vectors; counts are exchanged with `MPI_Alltoall`, and payloads are exchanged with `MPI_Isend` / `MPI_Waitall`
Routing locus	Routing is coordinated by the console or master	Each rank routes its own messages independently
Global bank liquidity (phase 5)	`callAll("reportBankLiquidity");` the console concatenates the results and broadcasts them back	`MPI_Allgatherv`
Phase synchronization	Implicit `callAll` bulk-synchronous barrier between phases	Explicit `MPI_Barrier` between phases
Correctness criterion	FNV-1a checksum of seven financial aggregates, compared across runtimes	FNV-1a checksum of seven financial aggregates, compared across runtimes

4.5 SugarScape

SugarScape is an agent-based grid simulation. Agents move across a sugar grid, compete for resources, and update their state each timestep. This workload combines distributed grid state with dynamic movement.

In MASS, the grid is divided into SugarChunk Places. Each chunk gets a part of the grid and the agents in it. Movement and boundary interactions happen through MASS communication between neighbors. This is a natural fit for MASS as the workload is spatial and based on phases.

In MadMPI, the grid is divided into row slabs across processes. Each process owns its rows and agents. Halo rows are exchanged with neighbors before movement decisions are made. Movement targets are chosen locally, boundary conflicts are exchanged and resolved, and agents that move across process boundaries are moved using custom MPI datatypes.

Table 6 summarizes the SugarScape implementation. The main difference is that MASS maps naturally onto distributed spatial chunks, while MadMPI implements the row-slab layout, halo exchange, conflict handling, and agent migration.

Table 6. SugarScape Implementation Comparison

Dimension	MASS C++	MadMPI
Grid partitioning	SugarChunk Places arranged as horizontal strips	Row slabs by MPI rank
Agent representation	Plain AgentRec structs stored in Place chunks; the native MASS Agent API is not used	Plain Agent structs with a custom MPI_Type_create_struct datatype
Halo exchange	placeExchangeBoundary called four times per timestep	Non-blocking MPI_Isend / MPI_Irecv
Agent migration	AgentRec structs are packed manually into Place boundary messages	Typed MPI sends through the exchangeNeighborVectors<T> template
Move protocol	Three-phase distributed flow: broadcast candidates, resolve conflicts on the owning chunk, then send winner replies	Three-phase distributed flow: broadcast candidates, resolve conflicts on the owning chunk, then send winner replies
Centralized phases	Target selection and conflict resolution are coordinated at one rank to avoid per-step inter-node dependency resolution	Target selection and conflict resolution are coordinated at one rank to avoid per-step inter-node dependency resolution
Correctness criterion	Final agent count, total agent wealth, and total grid sugar compared with a serial reference	Final agent count, total agent wealth, and total grid sugar compared with a serial reference

5. Experiment Setup and Methodology

This section describes the testing and comparison process used. The goal of the setup was to keep the workload behavior consistent between MASS and MadMPI, while still allowing each runtime to use its own natural style.

5.1 Execution Environment

All runs were performed on the Hermes cluster at the University of Washington Bothell. Hermes is a heterogeneous cluster. Nodes 1-12 use Intel Xeon Gold 5315Y processors, while nodes 13-24 use AMD EPYC 7252 processors. Each node has one socket, 4 physical cores, and about 9.4 GiB of RAM. The nodes are connected through 10 GbE Ethernet.

The software environment used Rocky Linux 9.7, GCC 11.5.0, C++17, MASS C++, and MadMPI/NewMadeleine git build 2026-01-21+1805f140ea794afbf38e0649a1de9d4793e65f4f. Runs used multiple physical nodes, with each configuration varying both the number of nodes and the number of execution units per node. In MASS, execution units refer to threads used by the runtime. In MadMPI, execution units refer to MPI processes.

This hardware difference matters as larger runs use both CPU types. As a result, performance is affected not only by the runtime, but also by the cluster’s mixed hardware. This is treated as a limitation when interpreting small timing differences.

5.2 Configuration Strategy

The experiments use strong scaling. This means the problem size is fixed, and more resources are added. For each workload, the node count and execution units per node were varied to measure whether additional parallelism improved performance or introduced overhead.

Each runtime was compared against its own baseline run with 1 total execution unit. This keeps the speedup comparison fair, as MASS and MadMPI do not expose parallelism in the same way. The comparison therefore includes how well the runtimes scale against their own starting points, so as not to claim that execution units are directly equivalent in both libraries.

5.3 Timing Methodology

The main performance measurement is wall-clock time for the core computation. Initialization, input setup, and final output were kept outside the measured region where practical, so the reported timing focuses on the actual workload. Usually, this means measuring the timestep loop or the main computation.

Failed runs, missing output, or invalid results were excluded from the performance comparison and treated as failed configurations.

5.4 Correctness and Validation

Correctness was checked by the appropriate validation methods. Heat3D uses a final temperature checksum. DGEMM uses a checksum of the output matrix. Graph Motif Search checks the final motif count. Bail-In/Bail-Out validates final aggregate checksums of the simulation state. SugarScape checks final agent count, total wealth, and total grid sugar.

These checks are vital, as speedup is only meaningful if the implementation still produces the expected result, and doesn't deviate from the baseline. A faster run that fails validation is not considered valid.

5.5 Programmability Evaluation

Programmability was evaluated from the finalized source code. The comparison focused on program size, function size, complexity, and runtime-specific communication and synchronization. This does not capture every part of programming effort, but it provides a practical and measurable way to compare the implementation complexity added by each runtime.

The programmability comparison also considers where the complexity appears. In MadMPI, complexity often comes from explicit ownership, routing, buffers, and synchronization. In MASS, complexity often comes from adapting the workload to Places, phases, and data passing. This makes the comparison more useful than only counting lines of code.

5.6 Fairness and Limitations

The implementations preserve the same high-level behavior wherever practical. Inputs, seeds, update rules, and timing boundaries were kept consistent so the comparison reflects runtime differences rather than unrelated program changes.

Some differences were unavoidable, as MASS and MadMPI use different programming models. These differences are part of the comparison, so they were kept, instead of forcing one or the other to use an unnatural implementation.

6. Performance

Performance was compared using wall-clock time and speedup from each runtime's own one-node, one-execution-unit baseline. Since MASS and MadMPI expose parallelism differently, speedup is interpreted within each runtime first, then compared against the other.

Overall, MadMPI reached stronger peak performance on Heat3D, DGEMM SUMMA, Bail-In/Bail-Out, and SugarScape. MASS performed better on DGEMM Cannon's algorithm and Graph Motif Search. This supports the main expected tradeoff: MadMPI is strongest when explicit communication control is useful, while MASS is more competitive when the workload fits its structured, phase-based model, or when minimal communication is needed.

The peak-performance results are summarized in Table 7. This table gives the best wall-clock time, peak speedup, and best configuration for each workload in both runtimes.

Table 7. Peak Performance Summary

Benchmark	Dataset	MadMPI			MASS C++			Winner
		Best Wall (s)	Peak Speedup	Config	Best Wall (s)	Peak Speedup	Config	
Heat3D	832^3, 512 steps	**72.771**	**8.77×**	24 nodes, 2/node	166.980	3.60×	8 nodes, 4/node	MadMPI
DGEMM SUMMA	$11{,}520^3$ matrix	**29.691**	**14.76×**	24 nodes, 1/node	45.032	8.49×	24 nodes, 1/node	MadMPI
DGEMM Cannon's	$11{,}520^3$ matrix	42.252	10.23×	4 nodes, 4/node	**26.732**	**14.89×**	24 nodes, 6/node	MASS C++
Graph Motif Search	triangle, V=250,000	45.395	14.50×	16 nodes, 2/node	**30.458**	**18.82×**	16 nodes, 6/node	MASS C++
Bail-In/Bail-Out	450 timesteps	**41.633**	**6.81×**	4 nodes, 4/node	239.524	2.64×	8 nodes, 1/node	MadMPI
SugarScape	$10{,}000^2$ grid, 10M agents, 100 steps	**24.000**	**10.89×**	24 nodes, 2/node	43.508	7.99×	4 nodes, 6/node	MadMPI

Figure 2 shows the same result visually by comparing the maximum speedup reached by each runtime. This makes the workload-dependent pattern easier to see: MadMPI leads in most workloads, while MASS leads in DGEMM Cannon's and Graph Motif Search.

Figure 2. Peak Speedup Comparison

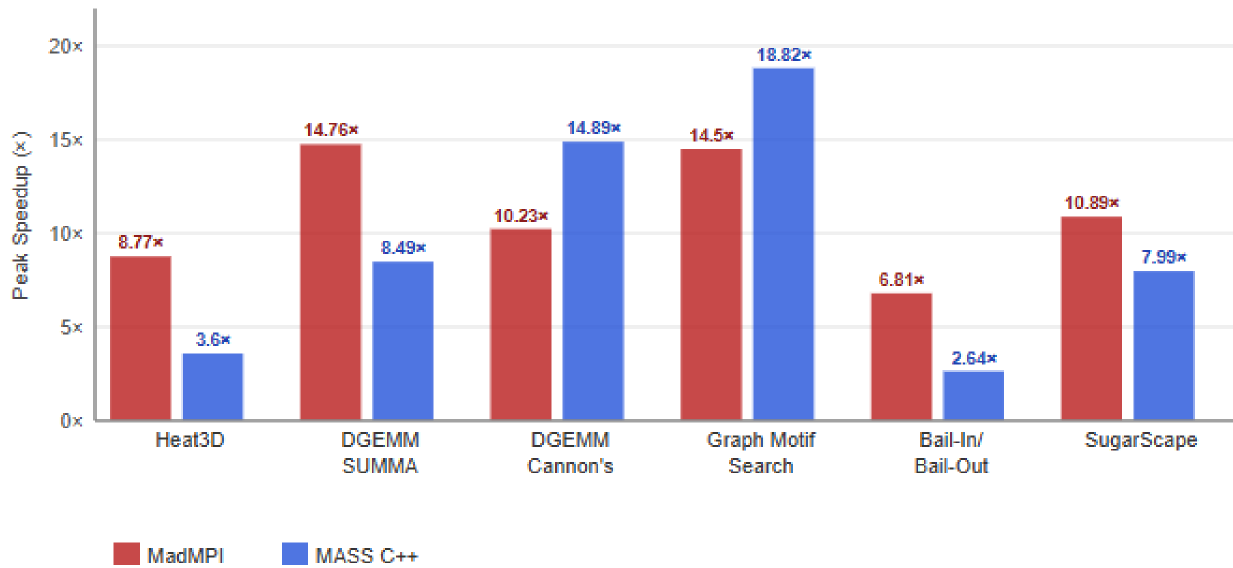
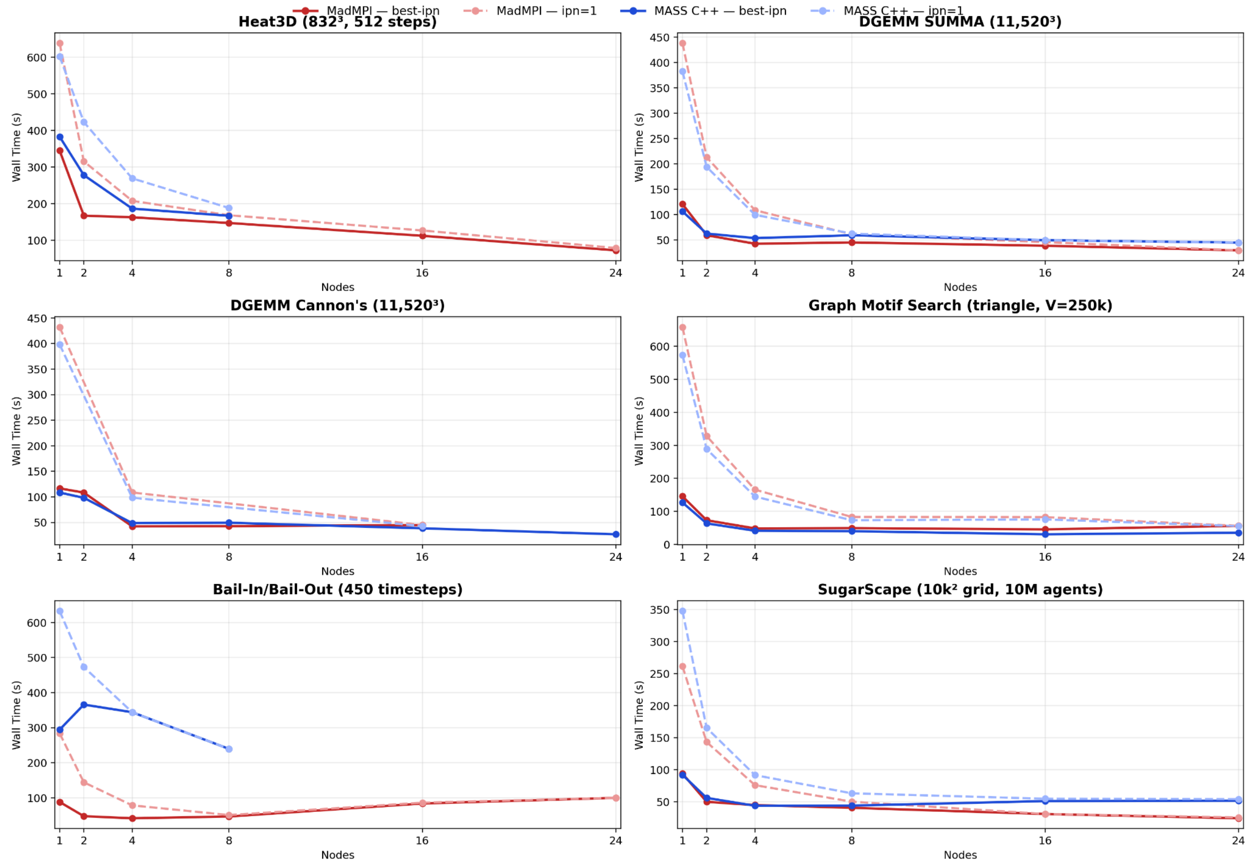


Figure 3 shows how runtime changes as node count increases for the workloads with fully plotted scaling data.

Figure 3. Scaling Trends for All Benchmarks



6.1 Heat3D

Heat3D shows a clear performance advantage for MadMPI. It reached a best time of 72.771 seconds with a maximum speedup of 8.77x at 24 nodes and 2 execution units per node. MASS reached a best time of 166.980 seconds with a maximum speedup of 3.60x at 8 nodes and 4 execution units per node.

This result matches the communication required, as Heat3D repeatedly exchanges halos in six directions. MadMPI allows persistent requests and overlap with interior computation, and while MASS can express the same halo pattern through Places, the higher-level structure adds synchronization and communication overhead that limits scaling earlier.

Table 8 gives the full Heat3D timing and speedup data. It shows that MadMPI continues improving through the largest tested configuration, and MASS reaches its best result earlier.

Table 8. Heat3D Performance Results

MASS C++ | 832^3 grid, 512 timesteps | baseline: 601.833 s				
Nodes	**Exec/Node**	**Total**	**Wall Time (s)**	**Speedup**
1	1	1	601.833	1.00×
1	2	2	555.740	1.08×
1	4	4	467.627	1.29×
1	8	8	382.702	1.57×
2	1	2	422.722	1.42×
2	2	4	313.755	1.92×
2	4	8	292.154	2.06×
2	8	16	277.957	2.17×
4	1	4	269.087	2.24×
4	2	8	186.627	3.22×
4	4	16	188.008	3.20×
4	8	32	189.726	3.17×
8	1	8	188.421	3.19×
8	2	16	185.901	3.24×
8	4	32	**166.980**	**3.60×**
MadMPI | 832^3 equivalent (Large), 512 timesteps | baseline: 637.919 s				
Nodes	**Exec/Node**	**Total**	**Wall Time (s)**	**Speedup**
1	1	1	637.919	1.00×
1	2	2	432.334	1.48×
1	4	4	344.816	1.85×
1	6	6	354.403	1.80×
2	1	2	315.489	2.02×
2	2	4	207.796	3.07×
2	4	8	167.521	3.81×
2	6	12	176.018	3.62×
4	1	4	207.690	3.07×
4	2	8	167.536	3.81×
4	4	16	168.822	3.78×
4	6	24	162.932	3.92×
8	1	8	168.653	3.78×
8	2	16	169.627	3.76×
8	4	32	147.386	4.33×
8	6	48	151.070	4.22×
16	1	16	126.975	5.02×
16	2	32	112.504	5.67×
16	4	64	112.446	5.67×
24	1	24	79.106	8.06×
24	2	48	**72.771**	**8.77×**

6.2 DGEMM

For SUMMA, MadMPI was faster. MadMPI reached 29.691 seconds and 14.76x speedup at 24 nodes and 1 execution unit per node. MASS reached 45.032 seconds and 8.49x speedup, also at the 24 nodes and 1 execution unit per node. SUMMA relies on row and column broadcasting, which maps well to MadMPI's communicator and non-blocking broadcast options. MASS still scales, but not as well, despite starting off with a faster baseline.

Table 9. DGEMM SUMMA Performance Results

MASS C++ | Large dataset (11,520³) | baseline: 382.451 s				
Nodes	**Exec/Node**	**Total**	**Wall Time (s)**	**Speedup**
1	1	1	382.451	1.00×
1	2	2	199.602	1.92×
1	4	4	108.331	3.53×
1	6	6	106.460	3.59×
2	1	2	194.174	1.97×
2	2	4	100.470	3.81×
2	4	8	62.747	6.10×
2	6	12	64.122	5.96×
4	1	4	99.735	3.83×
4	2	8	60.746	6.30×
4	4	16	53.774	7.11×
4	6	24	59.920	6.38×
8	1	8	62.739	6.10×
8	2	16	59.429	6.44×
16	1	16	49.746	7.69×
16	2	32	67.784	5.64×
24	1	24	**45.032**	**8.49×**
24	2	48	53.793	7.11×
24	4	96	89.287	4.28×
24	6	144	113.117	3.38×
MadMPI | Large dataset (11,520³) | baseline: 438.151 s				
Nodes	**Exec/Node**	**Total**	**Wall Time (s)**	**Speedup**
1	1	1	438.151	1.00×
1	2	2	219.001	2.00×
1	4	4	120.933	3.62×
1	6	6	132.158	3.32×
2	1	2	213.121	2.06×
2	2	4	109.948	3.99×
2	4	8	59.488	7.37×
2	6	12	67.374	6.50×
4	1	4	108.814	4.03×
4	2	8	59.704	7.34×
4	4	16	42.700	10.26×
4	6	24	43.827	10.00×
8	1	8	61.260	7.15×
8	2	16	45.267	9.68×
16	1	16	45.888	9.55×
16	2	32	38.775	11.30×
24	1	24	**29.691**	**14.76×**
24	2	48	30.396	14.41×

Cannon's algorithm showed the opposite result. MASS reached 26.732 seconds and 14.89x speedup at 24 nodes and 6 execution units per node, while MadMPI reached 42.252 seconds and 10.23x speedup at 4 nodes and 4 execution units per node. Cannon's algorithm requires square process counts, so the valid configurations are more restricted. In this case, MASS achieved the better best time and peak speedup.

Table 10. DGEMM Cannon's Performance Results

MASS C++ | Large dataset (11,520³) | baseline: 398.016 s | Square process grids only				
Nodes	Exec/Node	Total (√P×√P)	Wall Time (s)	Speedup
1	1	1 (1×1)	398.016	1.00×
1	4	4 (2×2)	108.403	3.67×
2	2	4 (2×2)	98.133	4.06×
4	1	4 (2×2)	98.330	4.05×
4	4	16 (4×4)	48.630	8.18×
8	2	16 (4×4)	49.406	8.06×
16	1	16 (4×4)	42.632	9.34×
16	4	64 (8×8)	38.459	10.35×
24	6	144 (12×12)	**26.732**	**14.89×**
MadMPI | Large dataset (11,520³) | baseline: 432.104 s | Square process grids only; non-square configs omitted				
Nodes	Exec/Node	Total (√P×√P)	Wall Time (s)	Speedup
1	1	1 (1×1)	432.104	1.00×
1	4	4 (2×2)	116.605	3.71×
2	2	4 (2×2)	108.362	3.99×
4	1	4 (2×2)	108.588	3.98×
4	4	16 (4×4)	**42.252**	**10.23×**
8	2	16 (4×4)	42.625	10.14×
16	1	16 (4×4)	44.752	9.66×

MASS achieves additional valid configurations at 64 and 144 total processes (8×8 and 12×12 grids) by using more nodes with higher ipn; MadMPI is constrained to square total rank counts available within the 64-rank limit.

6.3 Graph Motif Search

Graph Motif Search was also better for MASS. MASS reached 30.458 seconds and 18.82x speedup at 16 nodes and 6 execution units per node. MadMPI reached 45.395 seconds and 14.50x speedup at 16 nodes and 2 execution units per node.

This workload is different from Heat3D and DGEMM, as the graph is replicated, so all of the main work is local. Communication is only needed for final aggregation. Since communication is not the main cost, MadMPI's main advantage of control is mainly unused. The result depends more on how the work is divided, and how well the runtime handles the irregular local computation.

Table 11. Graph Motif Search Performance Results

MASS C++ | Triangle motif, V=250,000 | baseline: 573.340 s				
Nodes	**Exec/Node**	**Total**	**Search Time (s)**	**Speedup**
1	1	1	573.340	1.00×
1	2	2	287.232	2.00×
1	4	4	144.981	3.95×
1	6	6	126.825	4.52×
2	1	2	288.686	1.99×
2	2	4	144.732	3.96×
2	4	8	73.006	7.85×
2	6	12	63.509	9.03×
4	1	4	144.816	3.96×
4	2	8	73.222	7.83×
4	4	16	46.742	12.27×
4	6	24	41.217	13.91×
8	1	8	73.035	7.85×
8	2	16	49.922	11.48×
8	4	32	48.607	11.80×
8	6	48	39.858	14.38×
16	1	16	75.481	7.60×
16	2	32	45.742	12.53×
16	4	64	36.373	15.76×
16	6	96	**30.458**	**18.82×**
24	1	24	55.341	10.36×
24	2	48	49.633	11.55×
24	4	96	42.489	13.49×
24	6	144	35.331	16.23×
MadMPI | Triangle motif, V=250,000 | baseline: 658.267 s				
Nodes	**Exec/Node**	**Total**	**Search Time (s)**	**Speedup**
1	1	1	658.267	1.00×
1	2	2	334.834	1.97×
1	4	4	168.266	3.91×
1	6	6	145.595	4.52×
2	1	2	328.046	2.01×
2	2	4	165.320	3.98×
2	4	8	84.169	7.82×
2	6	12	73.050	9.01×
4	1	4	165.761	3.97×
4	2	8	83.304	7.90×
4	4	16	59.955	10.98×
4	6	24	48.327	13.62×
8	1	8	83.071	7.92×
8	2	16	53.186	12.38×
8	4	32	49.151	13.39×
8	6	48	49.352	13.34×
16	1	16	82.571	7.97×
16	2	32	**45.395**	**14.50×**
16	4	64	47.047	13.99×
24	1	24	56.182	11.72×
24	2	48	90.666	7.26×

6.4 Bail-In/Bail-Out

Bail-In/Bail-Out was the strongest MadMPI result in relation to MASS. MadMPI reached 41.633 seconds and 6.81x speedup at 4 nodes and 4 execution units per node. MASS reached 235.631 seconds and 2.68x speedup at 8 nodes and 1 execution unit per node.

This workload has clear phases, which makes it structurally suitable for MASS, but it also has a large amount of irregular communication. MadMPI greatly benefits from the low-level control in this case. Even though MASS keeps the simulation easier to express as ordered phases, the communication volume and synchronization cost limit its performance.

Table 12. Bail-In/Bail-Out Performance Results

MadMPI | Avg policy run time (bail-in and bail-out averaged) | baseline: 283.703 s				
Nodes	Exec/Node	Total	Avg Run (s)	Speedup
1	1	1	283.703	1.00×
1	2	2	148.791	1.91×
1	4	4	88.120	3.22×
1	6	6	90.037	3.15×
2	1	2	144.185	1.97×
2	2	4	79.120	3.59×
2	4	8	47.543	5.97×
2	6	12	50.179	5.65×
4	1	4	78.424	3.62×
4	2	8	49.736	5.70×
4	4	16	**41.633**	**6.81×**
4	6	24	44.505	6.37×
8	1	8	50.852	5.58×
8	2	16	46.692	6.08×
8	4	32	97.927	2.90×
8	6	48	129.738	2.19×
16	1	16	86.120	3.29×
16	2	32	83.777	3.39×
16	4	64	157.391	1.80×
24	1	24	99.846	2.84×
24	2	48	271.303	1.05×
24	4	96	271.249	1.05×
MASS C++ | Avg policy run time [(bail-in + bail-out) ÷ 2] | baseline: 632.346 s | Data beyond 8 nodes not collected				
Nodes	Exec/Node	Total	Avg Run (s)	Speedup
1	1	1	632.346	1.00×
1	2	2	419.833	1.51×
1	4	4	298.254	2.12×
1	6	6	294.336	2.15×
2	1	2	472.753	1.34×
2	2	4	395.534	1.60×
2	4	8	365.784	1.73×
2	6	12	515.698	1.23×
4	1	4	343.979	1.84×
4	2	8	452.593	1.40×
4	4	16	680.329	0.93×
4	6	24	779.488	0.81×
8	1	8	**239.524**	**2.64×**

6.5 SugarScape

SugarScape also favored MadMPI. MadMPI reached 24.000 seconds and 10.89x speedup at 24 nodes and 2 execution units per node. MASS reached 43.508 seconds and 7.99x speedup at 4 nodes and 6 execution units per node.

This result is important, as on paper, SugarScape is a good fit for MASS. The workload is spatial, phase-based, and agent-oriented. However, the MadMPI implementation uses a direct row-slab layout, halo exchange, conflict handling, and agent migration, which performed better in this case. MASS still scales, but is still outperformed by MadMPI.

Table 13. SugarScape Performance Results

MASS C++ | 10,000² grid, 10M agents, 100 timesteps | end-to-end time | baseline: 347.490 s				
Nodes	**Exec/Node**	**Total**	**End-to-End (s)**	**Speedup**
1	1	1	347.490	1.00×
1	2	2	216.485	1.61×
1	4	4	118.507	2.93×
1	6	6	110.458	3.15×
1	8	8	91.807	3.79×
2	1	2	165.031	2.11×
2	2	4	89.648	3.88×
2	4	8	58.359	5.95×
2	6	12	55.783	6.23×
4	1	4	91.331	3.80×
4	2	8	61.074	5.69×
4	4	16	46.978	7.40×
4	6	24	**43.508**	**7.99×**
8	1	8	62.975	5.52×
8	2	16	49.294	7.05×
8	4	32	46.002	7.55×
8	6	48	43.741	7.94×
16	1	16	54.522	6.37×
16	2	32	50.842	6.83×
16	4	64	51.056	6.81×
24	1	24	53.895	6.45×
24	2	48	52.660	6.60×
24	4	96	51.400	6.76×
MadMPI | 10,000² grid, 10M agents, 100 timesteps | total time incl. init | baseline: 261.286 s				
Nodes	**Exec/Node**	**Total**	**Total Time (s)**	**Speedup**
1	1	1	261.286	1.00×
1	2	2	159.039	1.64×
1	4	4	93.790	2.79×
1	6	6	100.272	2.61×
2	1	2	142.994	1.83×
2	2	4	75.835	3.45×
2	4	8	49.685	5.26×
2	6	12	51.112	5.11×
4	1	4	75.721	3.45×
4	2	8	49.873	5.24×
4	4	16	44.596	5.86×
4	6	24	45.358	5.76×
8	1	8	49.927	5.23×
8	2	16	42.160	6.20×
8	4	32	40.350	6.48×
8	6	48	44.136	5.92×
16	1	16	30.670	8.52×
16	2	32	32.220	8.11×
16	4	64	39.604	6.60×
24	1	24	25.342	10.31×
24	2	48	**24.000**	**10.89×**

MASS uses end-to-end time (avgEndToEndMs); MadMPI uses total time including initialization. Speedup computed from each runtime's own 1-node, 1-execution-unit baseline. MASS best-ipn plateaus after 4 nodes; MadMPI continues scaling through 24 nodes.

6.6 Performance Summary

The performance results show that neither runtime is always faster. When the workload benefits from explicit communication control, MadMPI tends to perform better. This is most visible in Bail-In/Bail-Out and DGEMM SUMMA.

MASS is at its best when the workload can use its model without paying too much overhead for communication and synchronization. This is clearly seen in DGEMM Cannon's algorithm and Graph Motif Search. These results show that the best runtime depends on the workload pattern, not just the general level of abstraction.

7. Programmability

Programmability was evaluated from the finalized MASS and MadMPI source code. The goal was to measure how difficult each implementation was to write, understand, and maintain. The metrics were grouped into categories like code footprint, function structure, control complexity, organization, and duplication. Scores were normalized so that higher values indicate better results.

Table 14 gives the raw code metrics behind the programmability comparison. These values show where the implementation differences come from, including code size, function size, cyclomatic complexity, and duplication.

Table 14. Raw Programmability Metrics by Workload

MASS C++								
Benchmark	**LoC**	**Funcs**	**Avg Nloc**	**Max Nloc**	**Avg CCN**	**Max CCN**	**HiCCN>10**	**Dup%**
Bail-In/Bail-Out	**3210**	47	**57.2**	**234**	**5.7**	**28**	**6**	11.19
DGEMM Cannon's	900	20	**36.8**	**97**	**3.25**	**7**	**0**	**0**
DGEMM SUMMA	1125	22	**42.2**	**172**	**3.27**	**9**	**0**	**0**
Heat3D	**987**	**20**	**42.3**	**162**	5.4	**21**	**2**	**0**
Motif Search	1539	33	**39.3**	297	**3.73**	17	**1**	**0**
SugarScape	3131	60	**45.9**	**266**	**4.62**	**26**	4	**0**
Average	**1815**	**33.7**	**43.9**	**205**	**4.33**	**18**	**2.2**	**1.86**
MadMPI								
Benchmark	**LoC**	**Funcs**	**Avg Nloc**	**Max Nloc**	**Avg CCN**	**Max CCN**	**HiCCN>10**	**Dup%**
Bail-In/Bail-Out	3531	**44**	69.6	832	6.59	37	8	**0**
DGEMM Cannon's	**597**	**4**	135.8	470	9.25	25	1	**0**
DGEMM SUMMA	**632**	**4**	144.5	505	9.5	26	1	**0**
Heat3D	1569	23	62.8	324	**4.52**	23	**2**	**0**
Motif Search	**737**	**12**	53.3	**133**	5.25	**16**	**1**	**0**
SugarScape	**2077**	**22**	87.1	616	9.14	64	**3**	**0**
Average	**1524**	**18.2**	**92.2**	**480**	**7.38**	**31.8**	**2.7**	**0**

Bold values indicate the better result for that metric. LoC = non-comment lines of code; Funcs = function count; Avg/Max Nloc = average/maximum lines per function; Avg/Max CCN = average/maximum cyclomatic complexity; HiCCN>10 = functions with CCN above 10; Dup% = code duplication percentage. Lower values are better for all metrics.

Figure 4 summarizes the category-level programmability profile. It shows that MASS has more manageable functions and better control complexity, while MadMPI is more compact.

Figure 4. Normalized Programmability Category Profile

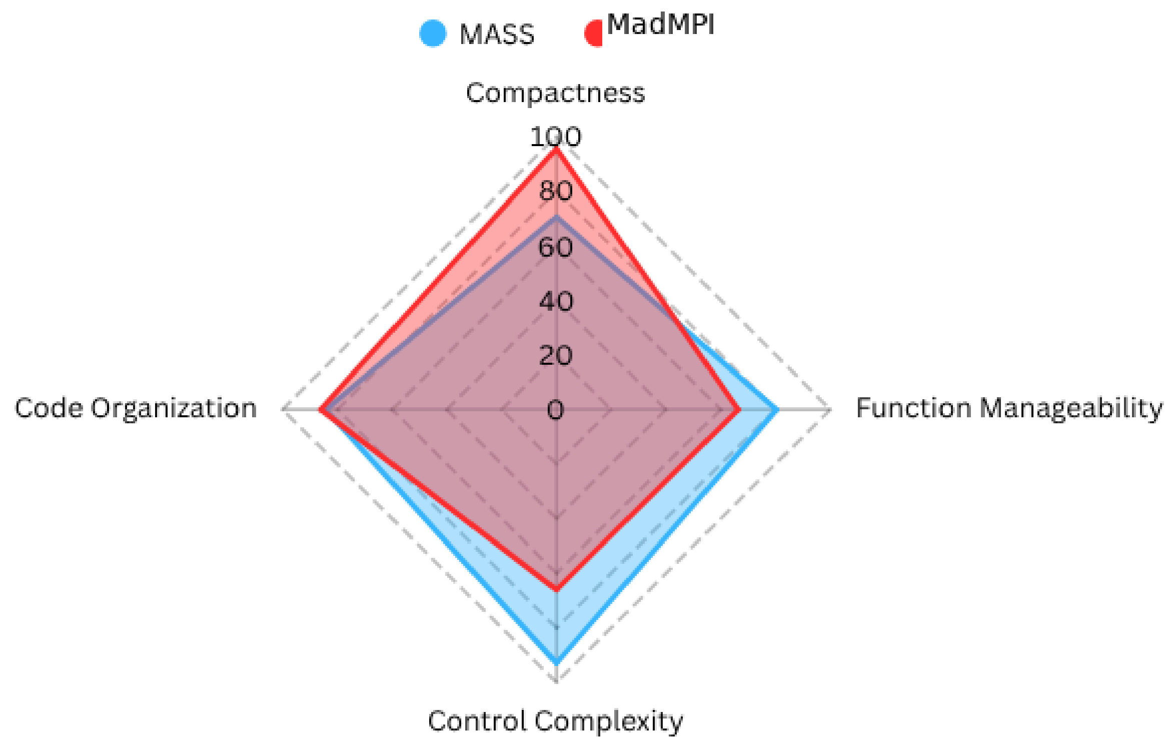


Figure 5 shows the normalized programmability score for each workload. MASS scores higher in five of the six workloads, while MadMPI scores higher on Graph Motif Search.

Figure 5. Normalized Raw Programmability Score by Workload

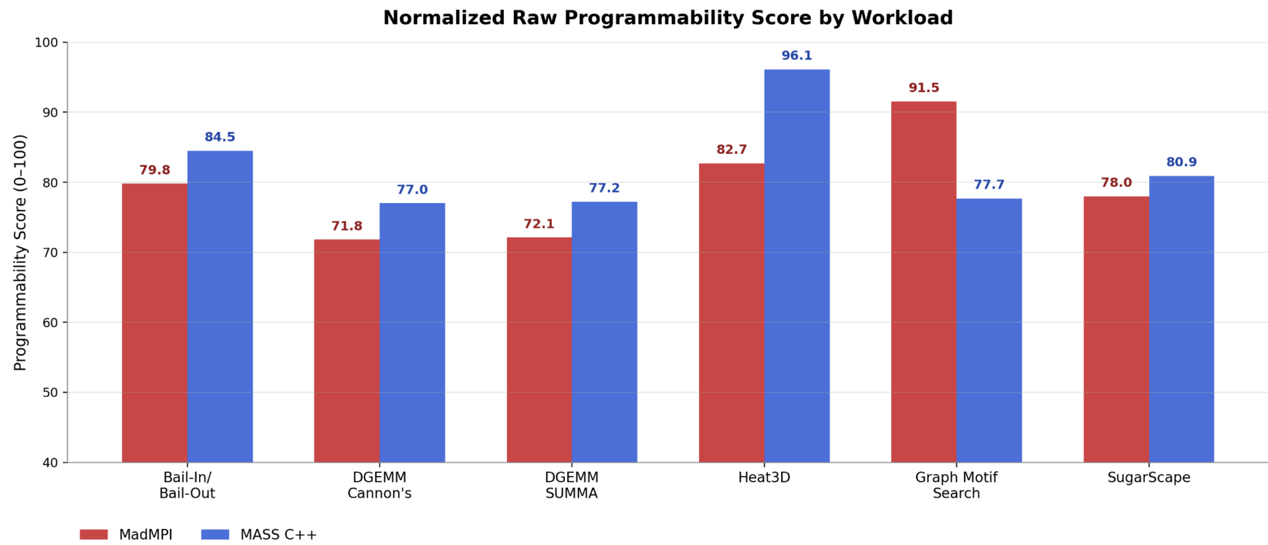


Table 15 lists the exact normalized raw scores shown in Figure 5. It is included to show the exact difference and the winner in a more direct manner.

Table 15. Normalized Raw Programmability Score Summary

Benchmark	MASS C++ Score	MadMPI Score	Difference	Winner
Bail-In/Bail-Out	84.5	79.8	+4.7	MASS C++
DGEMM Cannon's	77.0	71.8	+5.2	MASS C++
DGEMM SUMMA	77.2	72.1	+5.1	MASS C++
Heat3D	96.1	82.7	+13.4	MASS C++
Graph Motif Search	77.7	91.5	-13.8	MadMPI
SugarScape	80.9	78.0	+2.9	MASS C++
Average (unweighted)	**82.2**	**79.3**	**+2.9**	**MASS C++**

Normalized raw programmability scores from the Graphs sheet (0–100). Higher is better. Average row is an unweighted mean of the six workload scores.

7.1 Overall Results

Overall, MASS scored better than MadMPI on most workloads. MadMPI only did better on Graph Motif Search.

However, in several implementations, MASS required more setup or runtime-specific structure, leading to a less compact implementation. The main advantage was that the code was usually more organized, contained smaller pieces of simulation logic, and had less direct communication management. MadMPI was often more performance-oriented, but the code had to manage a lot of details more directly.

7.2 Heat3D

Heat3D showed one of the clearest programmability advantages for MASS. The workload naturally fits MASS, and the Place-based structure maps to grid chunks exchanging data well.

The MadMPI version exposes more implementation details. It must compute neighbor ranks, manage ghost layers, assign message tags, create communication requests, and wait for halo data before completing the boundary update. This increases the amount of communication-specific code the programmer has to track.

7.3 DGEMM

For both SUMMA and Cannon's algorithm, MASS scored higher in the programmability results. The MASS versions express matrix work through distributed Places and runtime-managed communication. This makes the implementation easier to follow, especially when looking at the computation as repeated distributed phases.

MadMPI is more explicit. In SUMMA, the implementation creates communicators, manages non-blocking broadcasts, and uses double buffering. In Cannon's algorithm, it handles panel shifting and message coordination. This makes the implementation communication-heavy and harder to read as a result.

7.4 Graph Motif Search

Unlike other benchmarks, MadMPI had a winning programmability score for Graph Motif Search. This is likely because the MadMPI implementation is relatively simple for this workload. There is little communication, which means there is less management.

For MASS, the same workload requires fitting the replicated graph and root-vertex partitioning into the Place model. Since the graph search is not naturally a grid-based or agent-based simulation, MASS does not provide as much structural benefit here. The extra MASS setup makes the implementation less direct than the MadMPI version.

7.5 Bail-In/Bail-Out

Bail-In/Bail-Out favored MASS in programmability. The simulation is divided into ordered phases, and that structure fits the way MASS programs are usually written. The implementation can be read as a sequence of phase updates over the distributed state.

MadMPI makes the same logic more explicit. Each process owns entity ranges, routes messages to other owner processes, and uses collectives and reductions for shared state. This made the MadMPI version faster, but also harder to write and reason about.

This workload shows that performance and programmability can point in different directions. MASS better matches the conceptual structure of the simulation, while MadMPI gives more control over the communication path.

7.6 SugarScape

SugarScape also scored slightly better for MASS. This makes sense, as the workload is spatial and phase-based. So, it maps naturally to the Place-based model.

However, the gap was smaller than in Heat3D or Bail-In/Bail-Out. SugarScape still requires careful handling of agent movement, boundary cases, and conflicts. MASS helps organize the simulation, but it does not remove all implementation complexity. MadMPI must implement row slabs, halo exchange, migration, and reductions directly, but the final structure is still reasonably understandable, as the grid decomposition is simple.

8. Conclusion

This report compared MASS C++ and MadMPI using five benchmarks, taking both performance and programmability results into account. The goal was to understand where one is a better fit than the other, and in what way.

The performance results showed that MadMPI usually achieved better peak performance when explicit communication control mattered. Direct control over message routing, collectives, synchronization, or communication overlap gave MadMPI an advantage. MASS's competitive performance showed that its higher-level model can still be competitive or faster when the workload fits the runtime structure or when communication is not the dominant cost.

The programmability results showed the opposite. MASS usually produced code that was easier to organize and reason about, especially for spatial or phase-based simulations. When the workload did not match MASS well, MadMPI was more direct and scored better. In general, MadMPI was the more compact implementation, but required more manual handling, which led to more complex code.

Overall, the results support the expected tradeoff. MadMPI provides more control and often better peak performance, but that control comes with more implementation complexity. MASS provides a cleaner structure for many distributed simulations, but its abstractions can add overhead or make certain communication patterns less direct. The best choice depends on the workload and on whether the priority is peak performance or a more maintainable implementation.

These results should also be interpreted within the context of the experiment. The comparison depends on the selected workloads, input sizes, final implementations, and the environment. Different problem sizes or tuning could change some results. However, the broader pattern is clear: runtime choice matters, and performance should be evaluated together with programmability rather than treated as a separate concern.

Bibliography

[1] J. R. Hammond, L. Dalcin, E. Schnetter, M. Pérache, J.-B. Besnard, J. Brown, G. Brito Gadeschi, J. Schuchart, S. Byrne, and H. Zhou, "MPI Application Binary Interface Standardization," in *Proceedings of EuroMPI2023: The 30th European MPI Users' Group Meeting*, Bristol, United Kingdom, 2023, pp. 1–12, doi: 10.1145/3615318.3615319.

[2] V. R. Basili, J. Carver, D. Cruzes, L. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz, "Understanding the High-Performance-Computing Community: A Software Engineer's Perspective," *IEEE Software*, vol. 25, no. 4, pp. 29–36, 2008, doi: 10.1109/MS.2008.103.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991, doi: 10.1177/109434209100500306. Link: https://doi.org/10.1177/109434209100500306

[4] L.-N. Pouchet, "PolyBench/C: The Polyhedral Benchmark Suite," 2012. Link: https://www.cs.colostate.edu/~pouchet/software/polybench/

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009, pp. 44–54, doi: 10.1109/IISWC.2009.5306797. Link: https://doi.org/10.1109/IISWC.2009.5306797

[6] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *Proceedings of the International Conference on Parallel Processing*, 2009, pp. 124–131, doi: 10.1109/ICPP.2009.64. Link: https://doi.org/10.1109/ICPP.2009.64

[7] M. Fukuda, K. Wang, S. Panther, N. Wong, I. Dudder, and Q. Shao, "An Analysis of Three C/C++ Cluster-Computing Libraries for Agent-Based Models," *ACM Transactions on Modeling and Computer Simulation*, vol. 35, no. 4, Article 28, 2025, doi: 10.1145/3732778. Link: https://doi.org/10.1145/3732778

[8] M. Kipps, W. Kim, and M. Fukuda, "Agent and Spatial Based Parallelization of Biological Network Motif Search," 2015. Link: https://faculty.washington.edu/mfukuda/papers/bionet.pdf

[9] PM2 Project, "MadMPI," *NewMadeleine - PM2 - Inria*. Link: https://pm2.gitlabpages.inria.fr/newmadeleine/

[10] O. Aumage, E. Brunet, N. Furmento, and R. Namyst, “NewMadeleine: A Fast Communication Scheduling Engine for High Performance Networks,” 2007, doi: 10.1109/IPDPS.2007.370476. Link: https://doi.org/10.1109/IPDPS.2007.370476

[11] A. Denis, M. Dreher, A. Gainaru, T. Herault, G. Mercier, R. Namyst, and E. Slaughter, “Scalability of the NewMadeleine Communication Library for Large Numbers of Threads,” 2019. Link: https://inria.hal.science/hal-02103700/file/article.pdf

[12] Ahmed Bera Pay. A Benchmark of C++ Cluster-Computing Libraries: MASS C++, HPX, and PM2. Master’s thesis, University of Washington Bothell, June 2026.