Feature Extension of MASS C++ towards a General Purpose Library

Rishabh Pratap Singh

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington
2026

Committee:

Munehiro Fukuda
Michael Stiber
Kelvin Sung

Program Authorized to Offer Degree:
Computing and Software Systems

University of Washington

Abstract

Feature Extension of MASS C++ towards a General Purpose Library

Rishabh Pratap Singh

Chair of the Supervisory Committee:
Munehiro Fukuda
Computing and Software Systems

MASS (Multi-Agent Spatial Simulation) C++ is a parallel computing library for agent-based simulations on distributed memory clusters, organised around the Bulk Synchronous Parallel model and a master-worker coordination scheme. Three structural limitations have constrained its applicability as a general-purpose runtime. An integer-based method dispatch scheme couples user-defined Place and Agent classes to the framework and propagates renumbering errors silently. A per-iteration master coordination cost of $K \times N$ (K operations $\times$ N iterations) barrier round-trips dominates wall-clock time on communication-intensive workloads. And a `Places` abstraction restricted to regular grids leaves social, biological, and transportation graphs without first-class support.

This thesis presents three feature extensions that address these limitations while preserving backward compatibility with existing MASS C++ programs. A three-tier dispatch architecture replaces integer switch/case with string-named methods, header-defined lambdas, and JIT-compiled lambdas, unified through a single dispatch registry. A phase-pipeline builder (`IterationConfig`) backed by an asynchronous handle-based executor lets the user record

an entire iteration as one dispatch, collapsing the $K \times N$ master round-trip pattern into a single dispatch by allowing workers to advance through compute, communication, and agent-management phases autonomously. A graph stack extends MASS C++ from grid-only topologies to arbitrary graphs, with bidirectional adjacency, locality-aware edge-cut partitioning, edge-constrained agent migration, Pregel-inspired combiners and aggregators, and a pluggable parser interface for user-defined graph formats.

This thesis evaluates the extensions on five benchmarks (Wave2D, SugarScape, PageRank, BFS Wavefront, and Random Walk) covering correctness, performance, and programmability. Compound execution reduces barrier round-trips by orders of magnitude and yields speedups that grow with the number of workers on Places-only workloads. SugarScape scaling exposes an $O(P^2)$ bottleneck (where P is the worker count) in the existing all-to-all agent exchange protocol, which caps compound speedup as P grows and motivates the sparse neighbour-rank exchange that graph-migrating agents adopt. The graph stack also positions MASS C++ as the only system surveyed in Section 3 that supports mobile agents over irregular topologies while remaining a drop-in extension of the existing grid-based runtime. The new lambda dispatch tiers cut user-written lines of code by 20–34% across the evaluated benchmarks against native methods, at per-call overheads under $\sim$15% on compute-heavy stencils and rising to $\sim$25% on operation-dense agent workloads.

Contents

1 Introduction

MASS (Multi-Agent Spatial Simulation) C++ [1] is a parallel computing library for distributed agent-based and spatial simulation on distributed memory clusters. Its programming model is built around two collective abstractions, distributed `Places` containers and mobile `Agents`, coordinated by a master-worker variant of the Bulk Synchronous Parallel (BSP) model [2]; the API and execution model are described in detail in Section 2. As MASS workloads have grown beyond regular spatial grids and compute-bound stencils, three structural features of the original design have begun to constrain what can be built on the library. This thesis identifies those features, presents three extensions that address them, and demonstrates with measurements that the extended library serves as a general-purpose runtime for distributed agent-based and graph-based computation while remaining a drop-in upgrade for existing simulations.

1.1 Problem Statement and Motivation

User-defined Place and Agent classes communicate with the runtime through a single virtual entry point, `callMethod(int functionId, void* arguments)`, dispatched by a hand-maintained switch statement inside each user class. Each collective primitive (such as `callAll` or `exchangeAll`, introduced in Section 2.1) therefore corresponds to an integer constant that the user must define, document, and pass at every call site. The scheme has four shortcomings. First, adding or reordering a method requires renumbering every existing call site, and the compiler cannot detect an incomplete renumbering. Second, the integer constants leak the framework's internal dispatch contract into user code, so user classes cannot be ported between MASS versions without source edits. Third, the runtime cannot validate that a requested method exists before invocation, so a stale integer silently dispatches to the wrong case (or to none at all) and surfaces only as a downstream crash or incorrect result. Fourth, the scheme has no support for inline lambdas, so even a one-line operation requires a header declaration, a source-file body, and a switch-case entry.

The master-worker variant of BSP that MASS C++ implements imposes a barrier at the end of every collective operation, so a simulation running K operations per iteration over N iterations forces the master to issue $K \times N$ collective triggers, each requiring a round-trip between the master and every worker. On communication-intensive workloads such as SugarScape (six operations per iteration with mobile-agent migration) or Wave2D (two operations per iteration with shadow-boundary exchange between adjacent ranks), barrier coordination consumes the dominant share of wall-clock time. Adding workers does not

improve the pattern: the master serialises dispatch and remains the per-iteration bottleneck, with overhead growing linearly with $K \times N$ rather than shrinking with P.

The `Places` abstraction is a multi-dimensional array of simulation entities with implicit lattice connectivity: array indices determine neighbour relationships, and there is no representation of an explicit edge between entities. This suits cellular automata, stencil computations, and structured spatial simulations, but not the broad class of agent-based workloads whose connectivity is irregular: social-network simulations, protein-interaction networks, transportation and supply-chain graphs, and contact-network epidemiology. Encoding any such workload in the grid abstraction forces edge information into user-managed out-of-band lookup tables, leaves the user responsible for restricting agent migration to logical neighbours, and ties partitioning to the grid dimensions rather than to the structure of the graph. The Java implementation of MASS includes a graph-topology container that handles a limited form of undirected graphs with round-robin vertex distribution [1]; no equivalent existed in MASS C++.

1.2 Thesis Objectives

The first objective is to replace integer-based method dispatch with a name-based system that decouples user classes from framework internals. The new system organises three coexisting tiers around one lookup path: a compile-time table of string-named member methods, header-defined lambdas registered once at shared-library load time, and just-in-time-compiled lambdas built at simulation start time. The three tiers share one name space and one call-site syntax, so a user class may declare operations in any combination of tiers without changing how the runtime invokes them.

The second objective is to collapse the $K \times N$ master coordination cost of an iterative simulation into a single master dispatch. A phase-pipeline executor lets the user describe an entire iteration as a sequence of named phases (place compute, place exchange, agent compute, agent manage, and so on), and an asynchronous handle-based dispatch mechanism lets the master invoke the pipeline once at the start of the simulation. Workers then advance through the phases autonomously, observing the original BSP barriers locally between phases but not crossing the master on every barrier, so per-phase BSP ordering is preserved while the per-iteration master round-trip is removed.

The third objective is to lift the grid-only restriction of the existing Places container by adding a graph-based distributed-computing stack on top of the existing Places and Agents APIs. The stack has three layers: a graph topology layer with bidirectional adjacency and

locality-aware edge-cut partitioning; a graph-aware vertex container that distributes graph vertices across workers and supports neighbour-only message exchange; and a graph-aware agent container that constrains agent migration to topology edges. The stack imports message combiners and named distributed aggregators from the vertex-centric framework family as opt-in optimisations while continuing to expose the mobile-agent abstraction that distinguishes MASS from those frameworks, and a pluggable parser interface lets users load graphs in their own formats without modifying library code.

Every extension must preserve backward compatibility with existing MASS C++ programs. Legacy simulations that use the integer-based dispatch path, the original blocking collective operations, or the grid-only Places container must recompile and run unchanged against the extended library, so each new feature appears as an opt-in surface alongside the existing APIs rather than as a breaking replacement.

1.3 Contributions

The first contribution realises the three-tier dispatch path as concrete machinery in MASS C++: a class-level `MASS_DISPATCH_TABLE` macro for the compile-time table, a `MASS_LAMBDA` header macro for the header-lambda tier, and a `JitCompiler`/`LambdaRegistry` pair for the JIT tier. Section 4.1 describes the implementation; Section 5.2 reports the resulting 20–34% reduction in user-written lines of code against native methods across the evaluated benchmarks and per-call overheads that stay under ∼15% on compute-heavy stencils and rise to ∼25% on operation-dense agent workloads.

The second contribution implements compound execution in three layers: a non-blocking handle for individual collective calls, a single-threaded executor that serialises those handles in each process, and a phase-pipeline builder that fuses an entire iteration into one master dispatch. Section 4.2 describes the implementation; Section 5.2 reports the resulting reduction in barrier round-trips and the speedups measured on Wave2D and SugarScape, including the worker-count $O(P^2)$ ceiling on SugarScape that motivates the sparse-neighbour exchange in §4.3.

The third contribution implements the graph stack as three classes in MASS C++ (`GraphTopology`, `GraphPlaces`, and `GraphAgents`), together with an `IGraphParser` extension point for user-defined graph formats and `setCombiner`/`registerAggregator` hooks for the vertex-centric optimisations. Section 4.3 describes the implementation; Section 5 reports correctness on PageRank and BFS Wavefront and the comparison against the graph-processing frameworks surveyed in Section 3. The combination of mobile agents with irregular topology

places MASS C++ in a workload class not directly supported by any of those systems.

The rest of the document is organised as follows. Section 2 describes the MASS C++ programming interface, its master-worker BSP execution model, and the state of graph support in the MASS ecosystem prior to this work. Section 3 surveys related work in task-based parallel runtimes, BSP optimisation, and distributed graph-processing frameworks, and positions the three extensions against those systems. Section 4 presents the design and implementation of the dispatch, compound-operations, and graph extensions, along with the resolved-issues catalogue that emerged during the work. Section 5 evaluates the extensions on five benchmark programs across correctness, performance, programmability, and comparison with industry standards. Section 6 summarises the findings and identifies directions for future work.

2 Background

This section describes the MASS C++ programming interface, the execution model it follows, and the state of graph support in the MASS ecosystem prior to this work. These details establish the constraints and design space within which all extensions operate.

2.1 MASS C++ API

MASS (Multi-Agent Spatial Simulation) C++ [1] is a parallel computing library for agent-based modeling and spatial simulations on distributed memory clusters. It exposes two core abstractions to the user:

- **Places**: A distributed array of simulation entities mapped to a logical grid. Each Place holds local state and communicates with neighbors through boundary or all-to-all exchange.
- **Agents**: Mobile entities that reside on Places and migrate across the grid. Agents carry their own state and interact with the Place they occupy.

Collective operations form the primary API. `callAll` invokes a method on every Place or Agent: each worker walks its local entities and, for each one, dispatches into the user class through a single virtual entry point, `callMethod`, which the user implements to map an operation identifier to a concrete member function. `exchangeAll` transfers state between arbitrary peer pairs across worker boundaries, and `exchangeBoundary` runs the special case of *shadow exchange*, in which adjacent ranks swap one-cell-thick boundary strips of Place state so that local stencil computations can read up-to-date neighbour values from the other side of the partition. `manageAll` handles agent lifecycle operations: spawn, kill, and migrate. A simulation is typically a loop that alternates compute steps (`callAll`) with communication steps (`exchangeAll`, `exchangeBoundary`, `manageAll`).

2.2 Execution Model

Master–worker topology. MASS runs as P processes across the cluster. Rank 0 acts as the master, driving each collective operation by broadcasting a trigger to the workers and waiting for their acknowledgment. Ranks $1, \ldots, P-1$ are workers that execute the requested operation locally and communicate peer-to-peer over TCP sockets. A single shared-state pattern on the master ensures that only one collective operation is in flight at a time. This constraint simplifies the implementation but prevents concurrent collective calls, which shapes

the asynchronous execution design in Section 4.2.

Bulk Synchronous Parallel execution. MASS follows the Bulk Synchronous Parallel (BSP) model [2], in which computation proceeds in a series of supersteps: local computation, global communication, then a barrier. Each MASS collective operation corresponds to one superstep. Within a superstep, workers read input from their local partition and any inbound messages exchanged in the previous step, then produce new state. Cross-worker effects become visible only after the barrier that closes the superstep. This is the weak-consistency guarantee characteristic of BSP-style runtimes: results become consistent across the cluster after each superstep, but not during one. For an iterative simulation with N iterations and K operations per iteration, this produces $K \times N$ barriers, each incurring a round-trip between the master and every worker. When per-operation computation is small relative to coordination cost, the BSP overhead dominates wall-clock time, which Section 4.2 addresses.

Intra-process threading. Each worker process runs a thread pool that parallelises every collective over its local entities. `callAll` and `exchangeAll` partition the local Place or Agent range across threads, each thread executes the user method on its assigned entities, and thread 0 performs any remaining serial coordination (per-rank buffer merge and decode into per-vertex inboxes) before the rank-level barrier. The default split is uniform by entity count, which suits the grid container where every Place does comparable work; Section 4.3 introduces an edge-balanced split for graph workloads where degree distributions are skewed. Network I/O between ranks runs on short-lived pthreads spawned per peer rank, which keeps send and receive paths independent and underpins the deadlock-free neighbour exchange of Section 4.3.

Dynamic class loading. User-defined Place and Agent classes compile to shared libraries (`.so` files). At runtime, MASS wraps the POSIX `dlopen/dlsym` interface to load these libraries and instantiate user classes on each node. This decouples the framework from user code: users can deploy new simulations without recompiling the MASS library itself. MASS assumes a shared filesystem (for example, an NFS-mounted home directory) so that master and workers see identical paths. Every rank sees compiled shared libraries, JIT-generated code, and configuration files immediately, without explicit distribution. This assumption simplifies deployment and enables the JIT compilation scheme described in Section 4.1, where a compiled lambda becomes available to every rank the moment it lands in the shared-filesystem cache.

2.3 Graph Support

MASS C++ began this thesis with no graph-aware runtime support. MASS Java includes a `GraphPlaces` module that supports undirected graphs with round-robin vertex distribution and agent traversal over vertices [1]; its C++ counterpart had no `GraphTopology`, no graph-aware communication, and no agent migration along graph edges, so the entire stack had to be built from scratch.

Modern graph workloads need several capabilities that MASS C++ lacks: bidirectional edge storage for reverse traversal, locality-aware partitioning beyond round-robin, compact binary serialization, and agent migration constrained to graph edges. Section 4.3 describes the resulting design.

3 Related Work

Three lines of prior work bear on the design of this thesis. The first is modern C++ parallel runtimes that expose lambda and task-level concurrency, represented here by HPX. The second is the C++/MPI subset of distributed runtimes that target the per-superstep cost of the Bulk Synchronous Parallel model, where Blogel, Pregel+, and Distributed GraphLab span the main design points. The third is the C++ side of the distributed graph-processing ecosystem, covering Pregel as the vertex-centric programming model, Pregel+ as its closest C++/MPI implementation, PowerGraph as the Gather-Apply-Scatter alternative, and the Boost Graph Library and METIS as the underlying graph and partitioning primitives. Each line aligns with a specific extension in this thesis: HPX with the dispatch and asynchronous-execution work (Sections 4.1 and 4.2), the BSP family with the compound-operation work (Section 4.2), and the graph ecosystem with the graph extensions (Section 4.3).

3.1 Task-Based Parallel Runtimes

HPX [3], [4] is a C++ runtime for parallel and distributed computing that implements the C++ standard library's parallelism and concurrency interfaces. Computation is expressed as asynchronous futures with continuations (`.then()`, `when_all`), and an Active Global Address Space (AGAS) lets remote objects be addressed as if they were local. Lambda-based execution policies attach user code to parallel algorithms, and a work-stealing user-level thread scheduler decides where each task runs.

HPX integrates tightly with modern C++: lambdas, futures, and ranges compose naturally with the rest of the language, so application code is close to standard C++ rather than an embedded DSL (Table 1, lambda support). AGAS hides the local/remote distinction, which avoids the explicit message-passing boilerplate that MPI-style codes carry, in contrast to the explicit master–worker socket plumbing in MASS. Continuation-based futures are declarative: dependencies are encoded through `.then()` chains rather than manual synchronization, which reduces the risk of barrier mistakes in complex pipelines.

Against those strengths, Grubel et al. [5] show that HPX performance is highly sensitive to task granularity, and fine-grained tasks can lose more to scheduling overhead than they gain from parallelism. HPX also has no first-class abstraction for spatial topologies or mobile agents, so agent-based simulations have to be encoded on top of futures by hand (Table 1, target workloads), and it offers no direct analog of the compound-operation pipeline that MASS uses to amortize coordination cost. Its programming model is orthogonal to BSP

rather than complementary: dropping HPX into an existing BSP library such as MASS C++ is not a port but a redesign. This thesis therefore keeps the BSP programming model and borrows only the two HPX ideas that map cleanly onto it, lambda-based behavior and handle-based asynchronous dispatch (Sections 4.1 and 4.2).

Table 1: *MASS C++ vs. HPX.*

Dimension	**MASS C++ (Extended)**	**HPX**
Lambda support	Three-tier: dispatch table, header, JIT.	Native lambdas in execution policies.
Async model	AsyncHandle on shared_future; serialized executor.	hpx::future with continuations.
Compound ops	IterationConfig: $O(1)$ round-trips per N iterations.	None; manual future composition.
Distributed memory	Master–worker, peer-to-peer sockets.	AGAS: transparent remote access.
Target workloads	Agents on grids and graphs.	General HPC (stencils, N-body, adaptive mesh refinement).

3.2 BSP Optimization and Iterative Execution

Valiant's BSP model [2] structures distributed computation as a sequence of supersteps consisting of local computation, global communication, and a barrier. Within the C++/MPI ecosystem to which MASS C++ belongs, three systems most directly target the per-superstep cost. Blogel [6] groups vertices into blocks and runs BSP at block granularity, amortising the barrier cost across many vertices per superstep. Pregel+ [7] keeps the per-vertex Pregel programming model intact but reduces the volume of inter-rank traffic through message combining at the source rank and a mirror-based broadcast scheme for high-degree vertices, alongside several load-balancing techniques for skewed graphs. Distributed GraphLab [8] relaxes the global barrier altogether by running vertex update functions asynchronously as soon as their inputs are ready, governed by configurable consistency models (full, edge, or vertex).

Each design point trades a different cost. Blogel preserves determinism and cuts the number of global synchronisations by orders of magnitude on partitioned graphs, but it relies on an offline blocking step that must be run before execution and ties partitioning to that one-shot choice. Pregel+ keeps the standard Pregel synchronisation pattern and only lowers

the constant factor through message reduction. Distributed GraphLab gives up the strict BSP barrier entirely, so application code must reason about update consistency rather than relying on a clean separation between supersteps. None of the three systems collapses an entire user-level iteration into a single master dispatch while keeping strict BSP semantics, which is the specific gap targeted by the compound-operation design in Section 4.2. The three systems also have no notion of a mobile agent that carries its own state between vertices.

3.3 Distributed Graph Processing Frameworks

The distributed graph-processing ecosystem spans both complete vertex-centric frameworks and the data-structure and partitioning primitives they build on. Pregel [9] introduced the vertex-centric "think like a vertex" model (computation expressed as a per-vertex `compute()` function invoked once per superstep on every active vertex) with per-superstep message passing, combiners, aggregators, and vote-to-halt; this programming model has influenced almost every subsequent graph-processing system. Pregel+ [7] is the most widely cited C++/MPI implementation of that model, layering message-reduction and load-balancing techniques on top of an API that stays close to the original Pregel surface. PowerGraph [10] departs from per-vertex `compute()` in favour of a Gather-Apply-Scatter decomposition and uses vertex-cut partitioning to handle high-degree vertices on power-law graphs. Underneath these frameworks sit the primitives that influence their internals: the Boost Graph Library [11] provides a header-only C++ adjacency-list representation with bidirectional edge queries, and METIS [12] performs multilevel k-way partitioning that minimises edge cuts while balancing partition size. METIS in particular informs the locality-aware partitioning strategy adopted in this thesis, even though the runtime implementation runs online at graph-load time rather than calling METIS as an offline tool.

Collectively, this ecosystem advances several axes that MASS C++ lacks: vertex-centric programming with combiners and aggregators (Pregel, Pregel+), GAS decomposition and vertex-cut partitioning for skewed graphs (PowerGraph), bidirectional in- and out-edge queries (BGL), and high-quality k-way partitioning (METIS). Each system has matured documentation, benchmarks, and tooling, and has demonstrated scaling to large graphs on commodity clusters. The combiner, aggregator, and fault-tolerance dimensions are summarised in Table 2.

For MASS C++, however, the limitations are mostly structural rather than performance-related. Both Pregel+ and PowerGraph are graph-only: there is no spatial-grid container alongside the vertex set, so workloads that mix spatial simulation with graph analytics

(for example, contact networks on a geographic grid) must stitch two frameworks together (Table 2, topology). Computation is also bound to the vertex, with no first-class entity that carries its own state between vertices, so simulations whose abstraction is a moving participant rather than a stationary node have to be encoded on top of vertex messages (Table 2, programming model). PowerGraph's vertex-cut model complicates per-vertex state ownership compared to the edge-cut model used in MASS. BGL is single-node only, so its data structures do not help with distributed execution. METIS is an offline tool: it must be re-run whenever the graph changes, which does not fit workflows where the graph is loaded once and then mutated by agents. None of these systems expose a pluggable parser layer that would let a user register a new graph file format without touching framework internals.

Table 2: *MASS C++ vs. C++ vertex-centric graph frameworks.*

Dimension	**MASS C++ (Graph Extensions)**	**Pregel+ and PowerGraph**
Programming model	Places + Agents (mobile, stateful). Phase pipeline.	Vertex-centric `compute()` with `voteToHalt()` (Pregel+); Gather-Apply-Scatter (PowerGraph).
Communication	exchangeNeighbors; grid exchangeAll fallback.	Vertex-to-vertex messages, next superstep (Pregel+); GAS edge updates (PowerGraph).
Combiners and aggregators	setCombiner; named aggregators.	Built-in combiners and aggregators (Pregel+); GAS reduction (PowerGraph).
Topology	Grids and graphs in one runtime.	Graphs only.
Fault tolerance	None (planned).	Checkpoint-based (Pregel+); checkpoint or GAS replication (PowerGraph).

3.4 Positioning of the Thesis

The limitations above fall into four themes. *(i) BSP mismatch*: HPX is orthogonal to BSP and cannot be adopted without a ground-up redesign, and the dominant open-source graph-processing implementations are JVM-bound and cannot be linked into a C++ runtime

like MASS, leaving Blogel, Pregel+, Distributed GraphLab, and PowerGraph as the principal C++/MPI alternatives. *(ii) Barrier reduction*: Distributed GraphLab relaxes consistency, requiring application code to reason about update consistency models rather than relying on the BSP barrier; Blogel adds an offline blocking stage; and Pregel+'s message-combining and mirroring lower the per-superstep constant factor but still incur the standard Pregel synchronisation pattern. None of these systems collapse a full user-level iteration into a single master dispatch while keeping strict BSP semantics. *(iii) Graph-processing gaps*: no mobile agents over graph topology, no unified grid-plus-graph runtime, offline or block-precomputed partitioning (METIS, Blogel), and closed parser pipelines. *(iv) Single-node tooling*: BGL provides the adjacency design space but does not scale beyond one node.

Table 3 summarises the position across MASS C++ and the closest related systems. The extensions address each theme in turn. Against (i), Section 4.1 imports lambda-based behaviour from HPX into MASS's BSP programming model, and Section 4.2 imports handle-based asynchronous dispatch (Table 3, lambda dispatch and async futures), both without disturbing backward compatibility. Against (ii), the compound-operation design in Section 4.2 collapses the $K \times N$ barriers of a K-operation, N-iteration loop into a single master dispatch while preserving strict BSP semantics inside each iteration (Table 3, compound iterations), giving a different trade-off than the message-reduction approach of Pregel+ or the asynchronous-relaxation approach of Distributed GraphLab. Against (iii), `GraphPlaces` and `GraphAgents` (Section 4.3) put mobile agents on graph topology using the existing Places and Agents APIs, the same runtime hosts both grid and graph workloads (Section 4), partitioning runs online at graph-load time with BFS-aware ordering inspired by METIS-style edge-cut minimisation (Table 11), and `IGraphParser` opens the parser pipeline to user-defined formats (Table 3, agent abstraction, grid and graph support, combiners, and aggregators). Against (iv), the `GraphTopology` data structure takes bidirectional adjacency from BGL and extends it across the cluster so in-edge queries work in distributed execution. Checkpoint-based fault tolerance of the kind Pregel+ provides is the one feature this thesis does not attempt; it is noted briefly as a direction for future work.

Table 3: *Feature matrix across MASS C++ and the closest related systems.*

Feature	**MASS C++**	**HPX**	**Pregel+**	**PowerGraph**
Lambda dispatch	✓	✓	–	–
Async futures	✓	✓	–	–
Compound iterations	✓	–	–	–
Grid support	✓	✓	–	–
Graph support	✓	–	✓	✓
Agent abstraction	✓	–	–	–
Combiners	✓	–	✓	GAS
Aggregators	✓	–	✓	GAS
Vote-to-halt	Via Agents	–	✓	–
Fault tolerance	–	–	✓	✓

4 Design and Implementation

This section presents the three feature extensions developed in this thesis, each addressing one of the structural limitations identified in §1.1. Section 4.1 replaces the integer-based method dispatch with three coexisting tiers (a string-keyed compile-time dispatch table, a header-defined lambda registry, and a JIT-compiled lambda registry) sharing one call-site syntax; Section 4.2 introduces an asynchronous and compound execution model that collapses the $K \times N$ master round-trip cost of an iterative simulation into a single master dispatch; Section 4.3 adds a graph-based distributed-computing stack on top of the existing Places and Agents APIs; and Section 4.4 documents the cluster-level robustness fixes that surfaced during integration.

4.1 Method Dispatch Modernization

The original MASS C++ dispatch contract required every user class to implement `callMethod(int functionId, void* argument)` as a hand-maintained `switch` statement mapping a numeric function identifier to a member function. Adding or reordering a method forced the user to allocate a new integer constant, update the switch statement, and update every call site, with no compile-time check that the three agreed. Figure 1 contrasts that scheme with the three-tier architecture that replaces it: a class-level macro builds a name-keyed dispatch table at compile time, a header macro registers lambdas once at shared-library load time, and a JIT compilation path produces lambdas at simulation start time. The three tiers share one call-site syntax (`callAll("Class::method")`) and one generated `callMethod` override, with the legacy integer entry point preserved alongside.

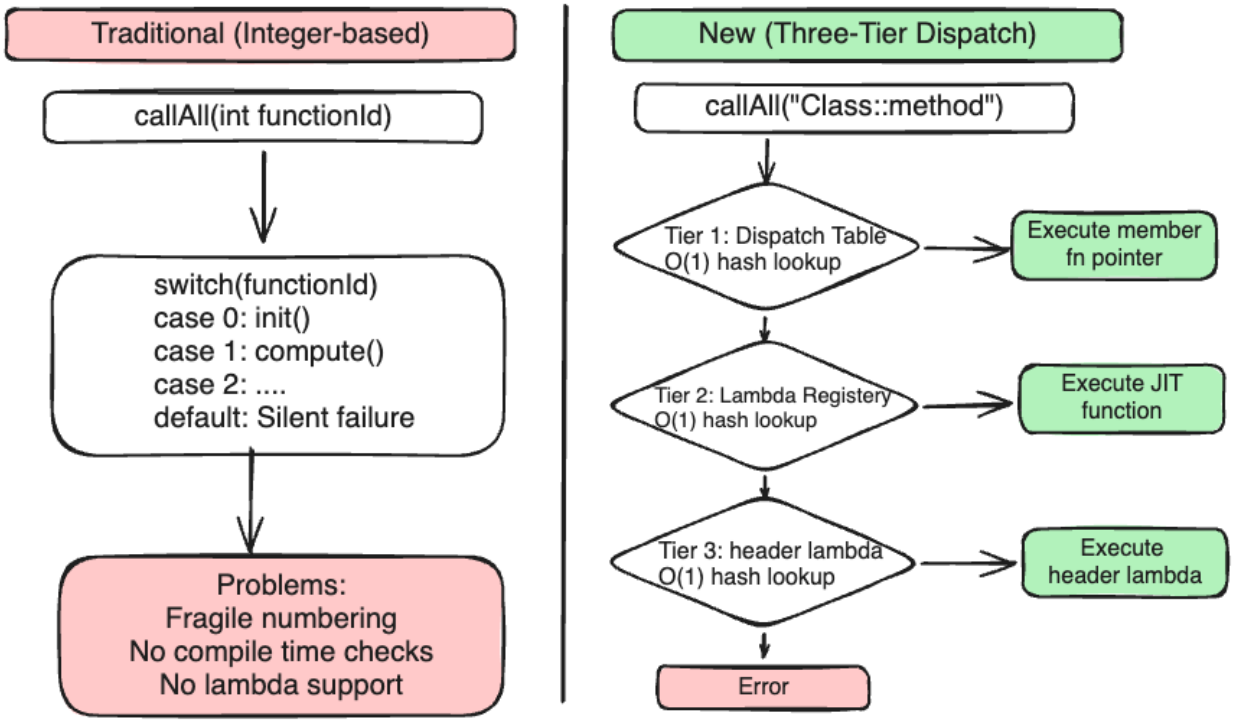

Figure 1: *Integer-based dispatch (left) vs. the three-tier architecture (right).*

String-Based Dispatch.

The first tier replaces the integer-based switch statement with a class-level macro that generates a name-keyed dispatch table at compile time. `MASS_DISPATCH_TABLE(ClassName, ...)` expands inside the user's class declaration into a static hash map keyed on the fully qualified method name (for example, `"Land::init"`) and storing the corresponding member-function pointer. The same expansion generates a `callMethod` override that resolves each call through one hash lookup. A nested helper, `MASS_METHOD(cls, name)`, contributes one entry per method, so the table accepts an arbitrary number of methods, is initialised at first use, and is shared across every instance of the class. A mismatch between a call-site name and a registered method is reported through `MASS_base::log` at the moment of the call rather than returning a null pointer as the integer switch did on fall-through.

Header Lambda Support.

The second tier supports inline lambda definitions inside header files for rapid prototyping. The `MASS_LAMBDA(ClassName, "name", body)` macro registers a `std::function` wrapping the lambda body in a static per-class lambda map owned by the Place or Agent base, keyed on the same `ClassName::name` string format used by the compile-time dispatch table. The macro expands inside a constructor or initialisation scope and pairs each registration with a one-shot

guard, so the first instance of the class registers the lambda and every subsequent instance skips the registration without a map lookup. An alternative form, `MASS_REGISTER_LAMBDAS`, runs the registration once at shared-library load time through a namespace-scoped static initialiser when the user prefers to keep all lambdas of a class in one source file.

Header lambdas are the lightest-weight way to add a new operation to a user class: a new method costs three coordinated edits in the compile-time tier (header declaration, source body, dispatch-table entry) but only one `MASS_LAMBDA` line in the header tier. §5.3 reports the resulting per-tier line-count savings against the native-method baseline and §5.2 reports the runtime overhead, which stays small for compute-heavy stencils and grows on operation-dense agent workloads. The overhead reflects the additional hash lookup the dispatch path performs when the compile-time table does not contain the requested name and the lookup falls through to the header-lambda map; once the lookup completes, the lambda body itself is a normal function-pointer invocation.

JIT Lambda Compilation.

The third tier supports lambdas written and compiled at simulation start time rather than at user-class compile time. Two singletons cooperate: a `JitCompiler` wraps each user-supplied lambda body in a thin C-linkage wrapper, compiles it to a position-independent shared object, and caches the result on the shared filesystem so subsequent simulations reuse the previous build; a `LambdaRegistry` loads the shared object and stores the resulting function pointer under the same `ClassName::name` key used by the other two tiers. The user-facing entries are `registerLambda` for synchronous compilation and `registerLambdaAsync` together with `waitForLambdas` for compilation that overlaps with simulation startup.

The JIT tier is the only path for lambdas whose body depends on parameters unknown at user-class compile time, and the only path that lets the user edit a behaviour and re-run without rebuilding the user-class shared library. The trade-off is a higher dispatch overhead bounded by 5–15% in §5.2 plus a one-time compilation cost that the shared-filesystem cache hides after the first run. Once compiled, each call reduces to a hash lookup, a function-pointer indirection, and the lambda body itself.

Unified Three-Tier Lookup.

The three tiers share a single `callMethod` implementation generated by `MASS_DISPATCH_TABLE`. Lookup checks the compile-time dispatch table first, then queries the `LambdaRegistry` on a miss, and finally falls back to the appropriate per-class lambda map (Place or Agent, selected through a compile-time-dispatched helper). The common case is therefore a single hash lookup on the compile-time table; the two lambda tiers run only when the dispatch table

does not contain the requested name. Resolution is deterministic across ranks because every rank sees the same shared-library cache on the same shared filesystem. A single class may mix all three tiers, and the generated `callMethod` routes each call to the right tier without changing the call-site syntax.

4.2 Asynchronous Execution and Compound Operations

The original collective API on Places and Agents is fully synchronous: every `callAll`, `exchangeAll`, `exchangeBoundary`, and `manageAll` returns to the master only after every worker has acknowledged the barrier that closes the operation. An iterative simulation with N iterations and K operations per iteration therefore charges $K \times N$ master–worker round-trips against wall-clock time, plus the per-round-trip cost of issuing the trigger and serialising the worker responses. On the communication-intensive workloads of §5.2, the per-operation compute is small relative to that coordination cost, so master barriers dominate end-to-end time and adding workers does not improve the pattern. This subsection addresses the bottleneck in two stages, illustrated side by side in Figure 2: a non-blocking handle-based interface that lifts master-side synchronisation off the critical path of a single collective, and a phase-pipeline executor that lifts master-side synchronisation off the critical path of an entire iteration loop.

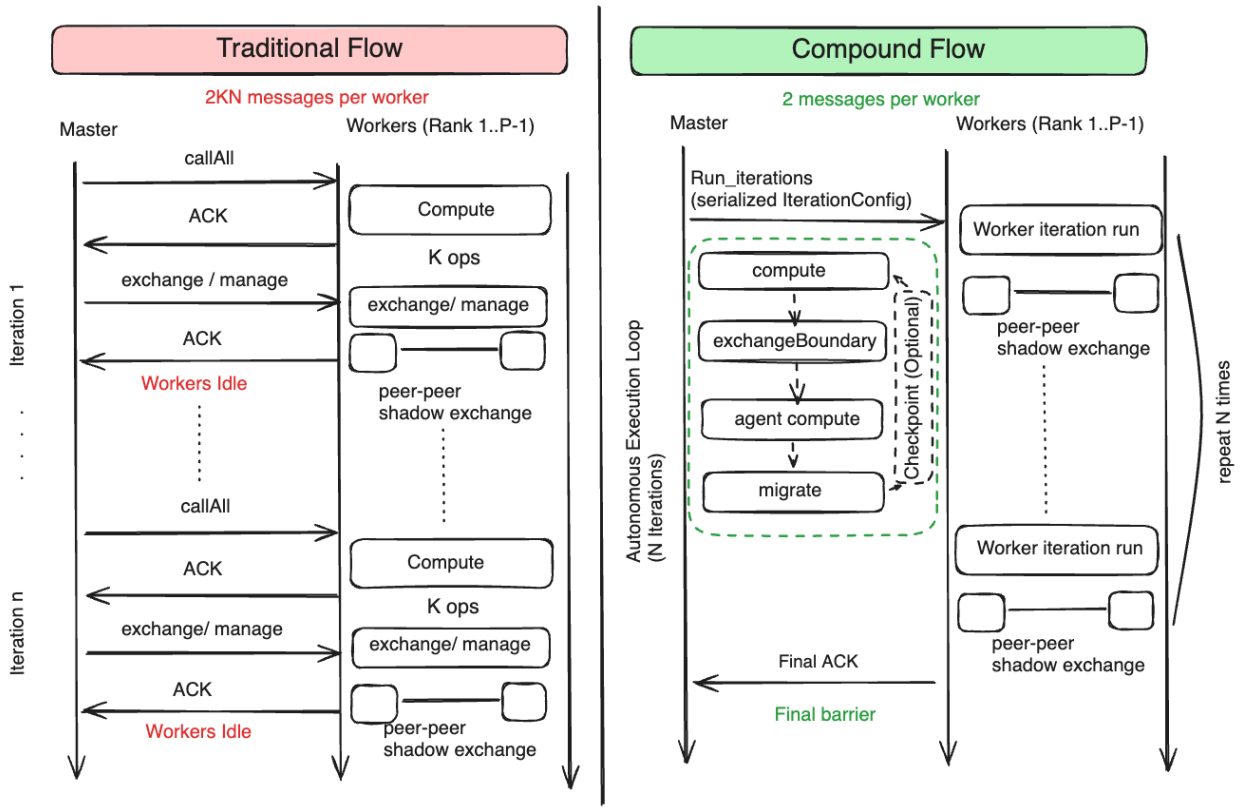

Figure 2: *Traditional $K \times N$ barrier execution vs. autonomous compound execution.*

AsyncHandle and AsyncExecutor.

The first stage of the redesign provides a non-blocking interface for the existing collective operations without changing their wire protocol. The user calls a new `*Async` variant of `callAll`, `exchangeAll`, `exchangeBoundary`, or `manageAll` and receives an `AsyncHandle`, a thin wrapper around a shared future (with a `void` specialisation for collectives that return no data). The handle exposes a small interface (`get`, `wait`, `isReady`, plus static `waitAll` and `waitAny` utilities for coordinating multiple outstanding handles). `std::shared_future` is used in place of `std::future` so that the handle can be copied between caller contexts, queried multiple times, and propagate exceptions to the call site of `get` instead of swallowing them.

Behind the handle, an `AsyncExecutor` singleton owns a single worker thread that drains a FIFO task queue and serialises the underlying synchronous collective. The executor is single-threaded because MASS keeps process-wide globals (the current Places container, the current method name, the current message type) that the worker thread pool reads during dispatch; running two collectives concurrently from the master would race on them. The master thread is still free to do other work between submission and `get`, and that property is

what makes the phase pipeline below safe to layer on top.

IterationConfig Phase Pipeline.

The second stage of the redesign moves the iteration loop itself from the master to the workers. The user describes an entire iteration through a fluent builder that records each per-iteration operation as a typed phase: place compute, place exchange, agent compute, agent management, and graph neighbour exchange. The configured pipeline is dispatched once through `MASS::runIterations(handle, config)`, which serialises the phase vector, sends a single dispatch message to every worker, and runs the same loop locally on the master's rank-0 partition. Workers then run all N iterations autonomously, hitting only the per-phase local barriers that BSP already requires, and the master and workers reconverge through a single barrier when the loop ends; the right-hand sequence of Figure 2 traces the autonomous loop. Compound dispatch therefore reduces the $K \times N$ master round-trips of a synchronous loop to one while keeping per-phase BSP ordering intact: Wave2D's two-phase iteration collapses from 200 master round-trips to one, and SugarScape's six-phase iteration collapses from 600 to one (Table 4). The full builder API is listed in Table 9 of the Appendix, and three benchmark configurations are reproduced in Listing 2 of the Appendix.

Table 4: *Barrier synchronization comparison for $N = 100$ iterations.*

Application	Ops/Iter	Traditional	Compound	Reduction
Wave2D (Places only)	2	200	1	200×
SugarScape (Places+Agents)	6	600	1	600×

Autonomous Agent Iterations.

Compound execution requires worker-side agents to obtain per-iteration arguments without help from the master, because every agent-management phase changes the per-rank population and invalidates the master's slicing offsets into the user-supplied argument array. The runtime addresses this by running a worker-side preparation step before each agent-compute phase: the user defines a method on the Agent class that reads the agent's current Place and writes the per-agent fields needed on the next compute step, so no per-agent argument data crosses the wire. Spawn, migrate, and kill extend the same idea. They are already peer-to-peer operations on the workers, and the symmetric blocking send/receive in the exchange path acts as an implicit all-to-all barrier, so the master's role reduces to triggering the operation; the compound pipeline drops even the trigger by running them from inside the worker loop, sequenced alongside the place-level phases in Figure 2. Agent identifiers stay globally unique

without a coordinator because each rank pre-reserves a disjoint range (rank multiplied by a per-node capacity) at startup.

Checkpoint and Convergence.

Pure run-to-completion dispatch suits fixed-iteration simulations, but two cases break it: long agent-bearing runs accumulate population drift on the master because its per-rank agent count is never refreshed mid-loop, and iterative solvers usually want to stop on a residual rather than at a hard-coded N. A periodic checkpoint configured as `checkpoint(K)` on the `IterationConfig` builder addresses both, occupying the optional checkpoint slot shown in the worker loop of Figure 2: every K iterations, each worker sends a small status payload and the master replies with a continue-or-stop decision. When the user also calls `convergence(method, threshold)`, the payload carries each worker's local maximum delta and the master compares the global maximum against the threshold; `convergenceCheckInterval(M)` restricts the test to every Mth checkpoint when the metric is expensive. A final barrier after the loop guarantees clean termination whether the run exhausts its iteration budget or stops early on convergence.

4.3 Graph-Based Distributed Computing

MASS C++ prior to this work supported only the regular-grid Places container of §2.1. The grid abstraction suits cellular automata, stencil computations, and structured spatial simulations, where array indices implicitly encode neighbour relationships, but it does not fit the agent-based workloads of §1.1 whose connectivity is irregular. Encoding any such workload in the grid abstraction forces edge data into user-managed lookup tables, constrains agent migration to grid neighbours rather than logical-topology neighbours, and ties partitioning to the grid dimensions. The graph stack introduced in this section gives those workloads first-class support without breaking the grid path.

Graph data follows a single, deterministic flow from a file on the shared filesystem to ready-to-compute Place objects on every rank. The six stages are:

1. **Raw input.** A graph file (a SNAP-style edge list[1], a HIPPIE protein-interaction TSV[2], or any other format with a matching parser) is read by rank 0 from the shared filesystem.

2. **Parser selection.** An `IGraphParser` factory selects an implementation based on the

[1]Stanford Network Analysis Project, a widely used distribution of large public graph datasets.

[2]Human Integrated Protein-Protein Interaction rEference, a curated TSV of human protein-interaction data.

file extension or a user-supplied parser handle and streams the file's vertices and edges into the loader.

3. **Topology construction.** The streamed records populate `GraphTopology`, an integer-native bidirectional adjacency keyed on global index and paired with an in-edges map for incoming-edge queries; original string identifiers from the input are preserved as label fields for debugging only.

4. **Partitioning.** The topology assigns each vertex a global integer index and then maps each vertex to one rank under an edge-cut scheme. The choice of partitioner (modulo, BFS-aware block, or a user-supplied function) is paired with the chosen index ordering so that locality-aware orderings produce locality-aware partitions.

5. **Distribution.** Rank 0 serialises each rank's slice as a compact binary `IndexedAdjacency` and ships it through the existing `ExchangeHelper` socket layer. Cross-partition edges carry the remote endpoint's global index as a reference rather than as replicated state, in keeping with the edge-cut model.

6. **Ready state.** Each rank deserialises its slice, instantiates the user's Place subclass for every local vertex, and registers the container with the framework's graph-container registry. The collective primitives `callAll`, `exchangeNeighbors`, and `manageAll` can then run over the graph using the same call-site syntax as the grid container.

For large graphs that would not fit in master memory after stage 3, the framework also provides a two-pass loading path: stage 1 scans the file once for integer ID statistics, and stage 4 invokes the parser-specific partition builder on a second scan to produce each rank's `IndexedAdjacency` directly. Peak master memory drops from $O(V + E)$ to $O(V + E/P)$ at the cost of $P + 1$ file reads.

The stack also exposes a small surface of new collective and per-vertex operations. The remaining paragraphs of §4.3 discuss the rationale behind each one, but for orientation:

1. `GraphPlaces::exchangeNeighbors(method)`, a neighbour-only collective that triggers a per-vertex compute step followed by a delivery step constrained to the graph topology.

2. `GraphPlaces::setCombiner(fn)`, a binary reducer applied at the source rank so that messages destined for the same remote vertex merge before transmission.

3. `registerAggregator(name, fn)` together with `aggregate(name, value)` and `getAggregatedValue(name)`, named distributed reductions whose result is visible on

the next iteration.

4. `GraphVertex::getOutEdges()` and `GraphVertex::getInEdges()`, adjacency queries the user calls inside a vertex method for scatter and gather passes (Table 10 lists the full accessor set).

5. `GraphAgent::migrate({dest})`, agent migration constrained to neighbour vertex indices read from the current vertex.

6. `IterationConfig::graphExchangeNeighbors(method)`, the same neighbour exchange exposed as a phase of the compound iteration pipeline introduced in §4.2.

Graph Topology Representation.

`GraphTopology` owns the input topology, the global-index assignment, and the partition routing for a graph workload. It stores two adjacency views in parallel: an out-edge view attached to each vertex and an in-edge view keyed on the target's global index. Maintaining both views adds a modest constant per edge but reduces in-neighbour queries from $O(V)$ to O(out-degree), which several algorithms require (connected components, PageRank's reverse iteration, clustering coefficients). A directedness flag controls the semantics: undirected mode auto-inserts the reverse edge but counts each unique edge once in the statistics, so degree, density, and partition-quality metrics match the standard graph-theoretic definitions. Identifiers are integer-native: the parser maps each input string to a sequential integer used as both vertex ID and global index, with the original string preserved as a label for debugging and JSON export. Per-edge memory is roughly 16 bytes and the routing hot path reduces to integer arithmetic. Helpers for average degree, maximum degree, and density feed the partition-quality diagnostics used in §5.2.

Index Ordering and Partitioning.

Global indices and partition assignment are coupled. The topology supports three index orderings selected through an `IndexOrder` enum. *Lexicographic* sorts vertex identifiers as strings before assigning integers; it is deterministic and reproducible but produces no locality, because identifiers in real graphs are not correlated with structural neighbourhood. *BFS* walks the graph from the highest-degree vertex and assigns indices in visit order, producing broad index bands that suit high-fan-out neighbourhoods. *DFS* walks from the same vertex but recurses into each subtree before backtracking, producing tall subtrees that suit locality along long paths. Both BFS and DFS yield locality because adjacent indices then tend to correspond to graph-adjacent vertices.

The partition strategy is paired with the index ordering through a `PartitionScheme` enum. Modulo partitioning ($v \bmod P$) pairs naturally with lexicographic ordering, where any contiguous slice of indices is as good as any other. Block partitioning ($\lfloor v/\lceil V/P \rceil \rfloor$) pairs naturally with BFS or DFS, where a contiguous slice keeps most edges local. A user-supplied callback that maps each vertex to a rank covers cases where neither default fits, including direct consumption of a METIS partitioning vector. All three options resolve to the same `IndexedAdjacency` structure on the worker side, so the rest of the runtime is partition-strategy-agnostic; Table 11 summarises the recommended pairings. Figure 3 contrasts the two defaults on a small example: modulo partitioning leaves the graph's community structure unaligned with rank boundaries, while block partitioning paired with BFS ordering co-locates graph-close vertices on the same rank and cuts the number of cross-rank edges. In either case, each rank holds its full out-edge lists, edges whose target lies on another rank carry the global index of that target rather than a replicated copy of the remote vertex's state (the edge-cut model), and cross-partition messages are sent on demand through `exchangeNeighbors`.

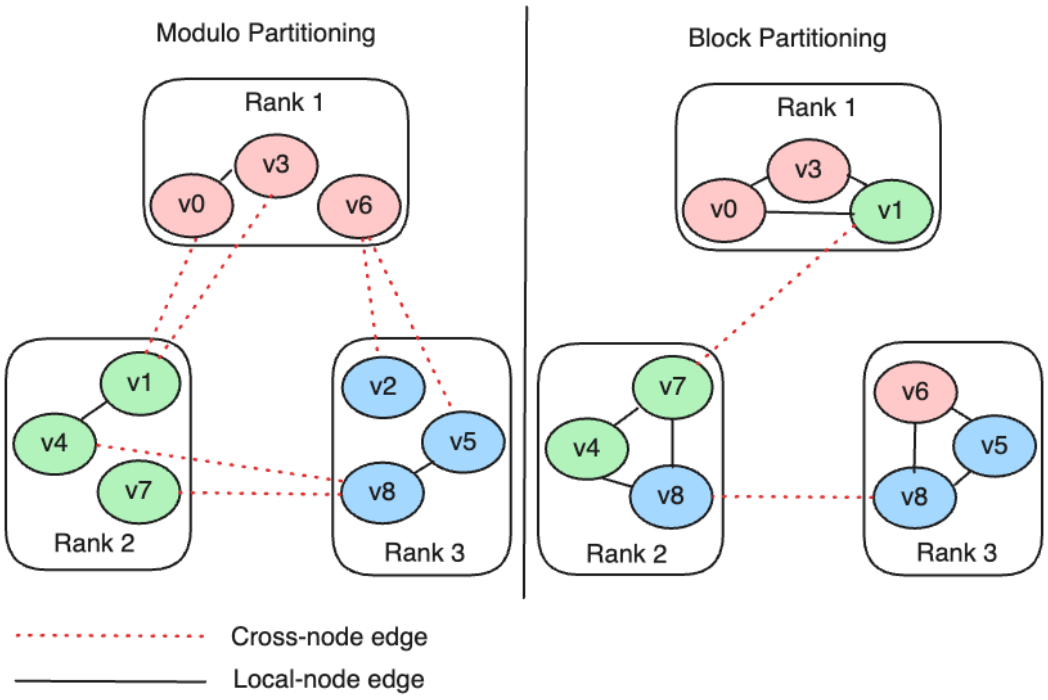

Figure 3: *Modulo versus block partitioning of a nine-vertex graph; colours mark each vertex's original modulo rank.*

Binary Serialization.

Partitions are shipped between the master and the workers as compact binary payloads. The wire format opens with a one-byte flag (directed or undirected, plus the chosen index order),

a four-byte vertex count, and a four-byte edge count, followed by vertices in sorted index order (a four-byte global index per vertex, optionally followed by its original string label) and then edges in sorted source-then-target order (two four-byte endpoint indices and an eight-byte weight per edge). Deterministic ordering is necessary for cross-rank reproducibility: serialisation sorts by integer index at write time and deserialisation rebuilds in the same order at read time. The integer-native representation cuts the wire payload by roughly 4× over a string-keyed equivalent at typical SNAP edge-list densities, and the same binary format underlies the two-pass loading path: the only difference is which side of the wire produces the bytes.

GraphPlaces: Vertex Distribution and Communication.

`GraphPlaces` bridges the topology layer with the existing MASS Place infrastructure. It extends `Places_base` so that the worker thread pool dispatches into user code on graph vertices without any change to the thread loop; the grid-only fields of `Places_base` (shadow size, boundary width, the multi-dimensional extent array) are set to one-dimensional defaults and never touched at runtime. The container offers two initialisation paths: the master can build the topology from a file through the parser factory, partition it, and ship each rank's slice to the workers through a new initialisation message, or it can build the topology programmatically and feed the same partition path. Workers run the same construction logic on the deserialised partition, so the master and worker code paths converge at a single graph-initialisation routine that loads the user's shared library, instantiates one Place object per local vertex, and computes the neighbour-rank set by scanning the local out-edges once. Listing 4 in the Appendix shows the file-based initialisation path.

The user-visible API change is shown in Listing 1: a traditional loop alternates `exchangeNeighbors` with a local `callAll`, while the compound version expresses the same two operations as phases of a single `IterationConfig` dispatched in one round-trip. Both formulations use the same neighbour-only collective underneath, which is the key departure from grid `exchangeAll`: `exchangeNeighbors` consults the neighbour-rank set computed at initialisation and only communicates with ranks that actually hold a neighbour of a local vertex, instead of opening a connection from every rank to every other rank on every iteration. Figure 4 summarises the threading model: worker threads run scatter compute in parallel to produce each vertex's `outMessage`, thread 0 merges the per-thread scratch into per-rank send buffers, one send and one recv pthread per neighbour rank exchange the merged payloads concurrently on independent sockets, and thread 0 decodes the inbound payloads into each vertex's `inMessages`. Sends and receives run on independent sockets, which removes the cyclic-wait deadlock that a naive send-then-receive ordering would suffer

on a graph cycle such as $A \to B \to C \to A$.

Graph workloads also need a different thread-level split. Grid `exchangeAll` partitions worker threads uniformly by Place count because every cell does the same amount of work; graph degree distributions are heavy-tailed, so a vertex-uniform split hands the few hub vertices to one thread and that thread dominates the iteration. The implementation balances by edge count instead, precomputing a prefix sum over local out-degrees at initialisation and caching the resulting per-thread vertex ranges so the scatter compute and buffer-build phases see the same balanced spans. Buffer build runs on lock-free per-thread scratch containers and is merged sequentially on thread 0, re-applying the combiner where two threads target the same destination. Receive threads run one pthread per neighbour rank into per-rank slots, leaving only the final decode into `inMessages` as a sequential step on thread 0 so ordering stays reproducible across runs. Local deliveries use a flat per-thread byte buffer of (target, size, body) triples in place of an earlier per-edge nested vector, dropping per-iteration heap allocations from O(local edges) to O(active threads).

Listing 1: *Traditional loop replaced by compound iteration; same semantics, one master round-trip for all iterations.*

```
// Traditional: per-iteration master round-trip
for (int i = 0; i < 50; i++) {
    gp->exchangeNeighbors("PageRankVertex::scatter");
    gp->callAll("PageRankVertex::gather");
}
```

Traditional loop replaced by compound iteration.

```
// Compound: one master dispatch covers all 50 iterations
IterationConfig config;
config.iterations(50)
      .graphExchangeNeighbors("PageRankVertex::scatter")
      .placeCompute("PageRankVertex::gather");
MASS::runIterations(placesHandle, config);
```

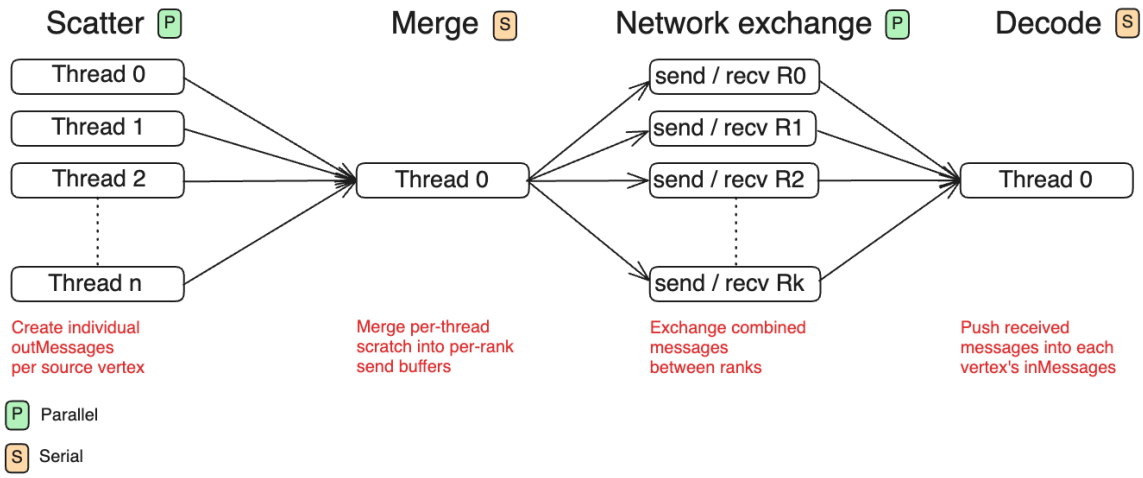

Figure 4: *Threading model of* `exchangeNeighbors` *on a single rank.*

Pregel-Inspired Features: Combiners and Aggregators.

Two Pregel-style optimisations were folded into the graph stack because each addresses a class of workload that the baseline neighbour exchange handles poorly. *Combiners* address the message-volume problem on hub vertices by reducing many same-destination messages at the source rank into a single payload before transmission. A SUM combiner on a power-law PageRank workload collapses the thousands of in-edge messages a hub would otherwise receive into one merged payload per source rank, with the merge running inside the per-thread buffer build. *Aggregators* address the convergence-detection problem by exposing named global reductions: each vertex contributes a value during its compute step, the framework performs a lightweight all-reduce across ranks at the end of the iteration, and the next iteration reads back a single global value. Aggregators slot directly into the convergence machinery of §4.2, so a termination condition such as "global delta below threshold" or "no remaining active vertices" is expressed as one or two aggregators rather than as ad-hoc collective traffic. Listing 3 in the Appendix shows a complete PageRank vertex class using both hooks.

Active-vertex bitsets and runtime topology mutation are not exposed: the agent model already subsumes the former (a living agent is an active vertex, death is vote-to-halt, migration is frontier expansion), and the workloads targeted here do not need the latter.

GraphAgents: Edge-Based Migration.

Agent migration on the grid runs as an all-to-all collective: every rank opens an agent-migration exchange with every other rank on every `manageAll`. The graph stack restricts agent movement to the topology and uses that restriction to eliminate the all-to-all. `GraphAgent::migrate({dest})` succeeds only if the destination vertex is one of the current

vertex's out-edge endpoints, and the migration helper obtains the local rank's neighbour set from the registered `GraphPlaces` container and opens migration sockets only to those ranks. For grid containers, which do not register as graph containers, the original all-to-all path runs unchanged. Table 5 contrasts the two patterns: a sparse graph partitioned with BFS or DFS ordering plus block partitioning typically yields a neighbour-rank set whose size is a small constant times P rather than $P - 1$, so the per-iteration connection count grows linearly in P rather than quadratically.

`GraphAgents` (instantiation example in Listing 5 of the Appendix) reuses the existing migration machinery wherever possible. It extends `Agents` through a protected constructor that accepts a Places handle directly, so `callAll`, `manageAll`, `doAll`, async variants, JIT lambda support, and population tracking are all inherited rather than duplicated, and an upcast to the base `Agents` type succeeds for both grid and graph agents so the compound iteration loop needs no special-case branches. The worker-side migration path reuses the same per-rank exchange helper as the grid path, with the per-pthread join loop tracking which threads were actually created because sparse exchange leaves gaps in the rank space. Sparse migration currently requires undirected graphs, because only their symmetric neighbour relations let the symmetric send-receive terminate cleanly on both sides; directed graphs require a pre-communication step to materialise the reverse neighbour set and are reserved as future work.

Table 5: *Agent exchange patterns: grid (all-to-all) vs. graph (sparse).*

Property	**Grid manageAll**	**Graph manageAll**	**Impact**
Connections per iteration	$P \times (P-1)$	$\leq E_{\text{cross}}$	Sparse graphs touch far fewer ranks.
Message routing	All-to-all	Neighbor-only	Avoids empty messages to non-neighbor ranks.
Scalability bottleneck	$O(P^2)$	O(cut edges)	Graph exchange avoids quadratic scaling.

Data Ingestion and Parsers.

Real graph data lives in too many formats to bake any one into the framework, so the loader is pluggable. `IGraphParser` is an abstract interface that concrete parsers implement and register with a factory keyed on file extension. Four parsers ship with the stack, covering SNAP edge lists, adjacency lists, CSV edge tables, and the HIPPIE protein-interaction TSV; Table 12 in the Appendix lists their formats and Listing 6 in the Appendix shows how a

user registers a new format. Every parser produces an integer-native `GraphTopology`: each unique input identifier is mapped to a sequential integer at first sight, with the original string preserved as a label.

For graphs too large to fit in master memory after stage 3 of the data flow, opt-in parsers implement a two-pass path. The first pass streams the file once for integer ID statistics, and the second pass streams it once per rank to emit each rank's partition directly. Peak master memory drops from $O(V + E)$ to $O(V + E/P)$ at the cost of $P + 1$ file reads on the shared filesystem.

Vertex Properties

Edge weights alone are insufficient for many target workloads: road and city graphs carry latitude, longitude, and capacity per intersection; protein-interaction networks carry expression levels and functional annotations; web graphs carry categorical metadata and dense embeddings. The vertex-properties extension layers structured per-vertex state on top of the existing partitioning and distribution machinery without disturbing the edge-cut model, and supports two ownership modes for who reads the property file. In *master-push* mode (master reads and distributes), the master parses the property file once, slices the per-vertex blob to match each partition, and ships the slice alongside the partition's adjacency on the existing graph-initialisation message; this suits sets up to a few hundred megabytes. In *worker-pull* mode (each worker reads its own slice), the master forwards only the file path and the property format, and each worker streams the file from the shared filesystem retaining only the rows whose vertex identifier maps to a local global index; this suits sets too large to materialise on the master.

Loader selection mirrors the parser factory: a vertex-property-loader interface with two entry points (one for master-push, one for worker-pull), with a CSV loader keyed on a vertex-identifier column as the first concrete implementation. The graph wire format reserves an optional fixed-size byte blob per vertex; when present, the construction loop passes each vertex's blob to the user's Place constructor on instantiation, and when absent the construction loop routes through the unchanged single-argument path so grid and graph workloads without per-vertex properties incur no additional cost. The loader provides the runtime foundation for the property-mutation direction in §6.3.

4.4 Resolved Issues

Integration of the compound and graph extensions exposed three robustness issues, all resolved before evaluation.

- **ExchangeHelper handshake reliability.** The hostname-based connection handshake hung intermittently at three-node configurations and produced sporadic `failed to read connecting rank` errors at all cluster sizes. The fix replaces the hostname check with a rank-based handshake protocol, adds `readFull`/`writeFull` helpers for partial-read safety, and hardens the accept loop with per-socket receive timeouts.

- `cleanInMessages` **leak and missing autonomous-loop cleanup.** A pre-existing memory leak in `Place::cleanInMessages` dominated heap pressure under compound iteration, and the autonomous compound loop never invoked user-side cleanup between phases. The fix corrects the leak and inserts an automatic `cleanInMessages()` call before every `exchangeAll` phase.

- **Wave2D Gauss-Seidel snapshot bug.** Fusing Wave2D's read and write into a single `callAll` let same-node neighbours observe already-updated state, producing a Gauss-Seidel update within each thread stripe rather than the intended Jacobi update across the grid. The fix is snapshot double-buffering plus a restructured pipeline that separates the read and the write into distinct phases so that the implicit barriers between phases enforce Jacobi ordering.

5 Evaluation

This section evaluates the three extensions on five benchmarks. §5.1 fixes the measurement environment and the per-benchmark correctness criterion; §5.2 reports performance across dispatch tiers, compound vs. traditional execution, and strong scaling; §5.3 quantifies developer-effort impact; §5.4 positions MASS C++ against the closest C++ analogues surveyed in Section 3; and §5.5 records the limitations and threats to validity that bound the conclusions.

5.1 Experimental Setup and Verification

Benchmarks ran on the UWB *Hermes* cluster (Rocky Linux 9.7; Intel Xeon Gold 5315Y, 6 virtual per node). Worker processes used 2–24 nodes (plus a master on the submission host), six POSIX threads per worker, and a shared NFS home directory (`cssnfs6.uwb.edu`). MASS C++ and the benchmark drivers were built with `g++ 11.5.0` (`-std=c++20`).

The five benchmarks (Table 6) span the four topology–coordination combinations of interest. Wave2D runs over 500×500, 1000×1000, 1500×1500, and 2000×2000 grids for 200 iterations of a Jacobi-style wave update. SugarScape runs over the same grid sizes for 200 iterations with 1000 mobile agents per run. PageRank runs on the web-Google (875,713 vertices, 5.1M edges) and soc-Pokec (1,632,803 vertices, 30M edges) graphs for 100 iterations. BFS Wavefront runs on web-Google from a fixed source vertex, capped at 100 outer iterations but terminating earlier when the visited set is stable. Random Walk runs 1000 agents on web-Google for 100 iterations, with a matching grid variant (571×571) for the all-to-all versus sparse-exchange comparison. Each configuration is executed at 2, 4, 8, 12, 16, 20, and 24 worker ranks with six threads per rank.

Each benchmark carries a verification criterion that every reported configuration satisfies before timing numbers enter the medians of §5.2. Wave2D outputs from the compound variant match the traditional variant to within 10^{-10} per cell (maximum absolute difference). SugarScape preserves a global sugar-plus-wealth conservation invariant, with the final value within one percent of the initial sugar mass. PageRank converges to the reference ranks within 10^{-6} per vertex. BFS Wavefront's visited set and discovery depths match an offline single-rank BFS from the same source. Random Walk preserves the total agent population across all migrations. A failure of any criterion drops the affected configuration from the median and triggers a re-run.

The performance numbers in §5.2 are collected through the v2 `IterationConfig` phase

pipeline, which replaces an earlier template-based compound API. The v2 pipeline expresses six simulations that the v1 design could not fully accommodate: a SocialNetwork iteration whose per-iteration argument changes between calls; a BrainGrid iteration with four exchange cycles per outer iteration; a VDT iteration whose termination depends on a collective return value; a MatSim iteration with eight exchanges and a directional sub-loop; a Tuberculosis iteration with two agent types and explicit spawn-and-init phases; and a BailInBailOut iteration with three agent types and agent-level return values. These six simulations are not measured in this thesis, but they motivate the generalised pipeline; the five benchmarks below are the subset on which the pipeline could be measured against a matching traditional baseline.

Table 6: *Benchmark programs.*

Benchmark	**Topology**	**Thesis feature evaluated**	**Verification criterion**
Wave2D	Grid (Places)	Three-tier dispatch (Section 4.1); compound operations on Places (Section 4.2).	Compound output near-identical to baseline ($\Delta_{\max} < 10^{-10}$ per cell).
SugarScape	Grid (Places + Agents)	Dispatch tiers (Section 4.1); compound operations with agent migration; $O(P^2)$ exchange limit (Section 4.2).	Sugar conservation: $\lvert\text{final sugar} + \text{final wealth} - \text{initial sugar}\rvert / \text{initial sugar} < 1$ across the run.
PageRank	Graph (Places)	GraphPlaces with combiners and compound iteration (Section 4.3).	Per-vertex rank converges to reference values within 10^{-6}.
BFS Wavefront	Graph (Places + Agents)	GraphAgents migration along edges; spawn/migrate/halt lifecycle (Section 4.3).	Visited set and discovery depths match offline BFS from the same source.
Random Walk	Graph (Agents)	Sparse graph manageAll vs. grid all-to-all migration (Section 4.3).	Total agent population conserved across all migrations.

5.2 Performance Evaluation

Each benchmark in Table 6 is reported in three slices: dispatch overhead (method vs. header lambda vs. JIT), compound vs. traditional across workload sizes, and strong scaling at the largest workload. All simulation-time numbers are the median of three repetitions; per-configuration run-to-run variance stays under ∼3% unless noted, and the per-phase recorder

(an instrumentation hook that snapshots Place and Agent state at each compound phase) is disabled for every reported measurement (see §5.5).

Wave2D.

Dispatch overhead on a compute-heavy stencil stays bounded (Fig. 5): header lambda runs 3–6% over the native-method baseline and JIT runs 14–15%, with the ordering (method < header lambda < JIT) consistent at both 1000×1000 and 2000×2000. The JIT tier pays the extra cost of a per-call `LambdaRegistry` lookup that the other two tiers resolve at compile time or shared-library load time.

Compound beats traditional at every node count on the median across grid sizes (Fig. 6), and the gap widens from ∼3% at two nodes to ∼13–17% at 24 nodes. The widening tracks the 400 master round-trips saved per 200-iteration run: as per-rank compute shrinks, that constant saving becomes a larger share of wall-clock. Both variants share the same plateau shape across node counts, so above eight nodes the limiting cost is the per-iteration boundary exchange between adjacent ranks rather than master coordination; compound's saving rides on top of the same exchange floor.

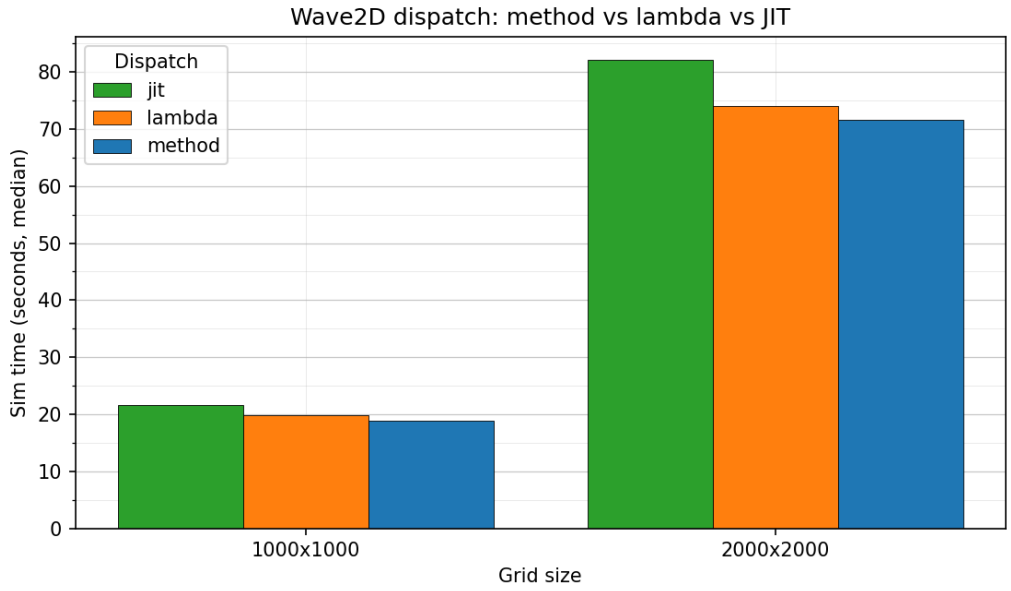

Figure 5: *Wave2D: method vs. header lambda vs. JIT.*

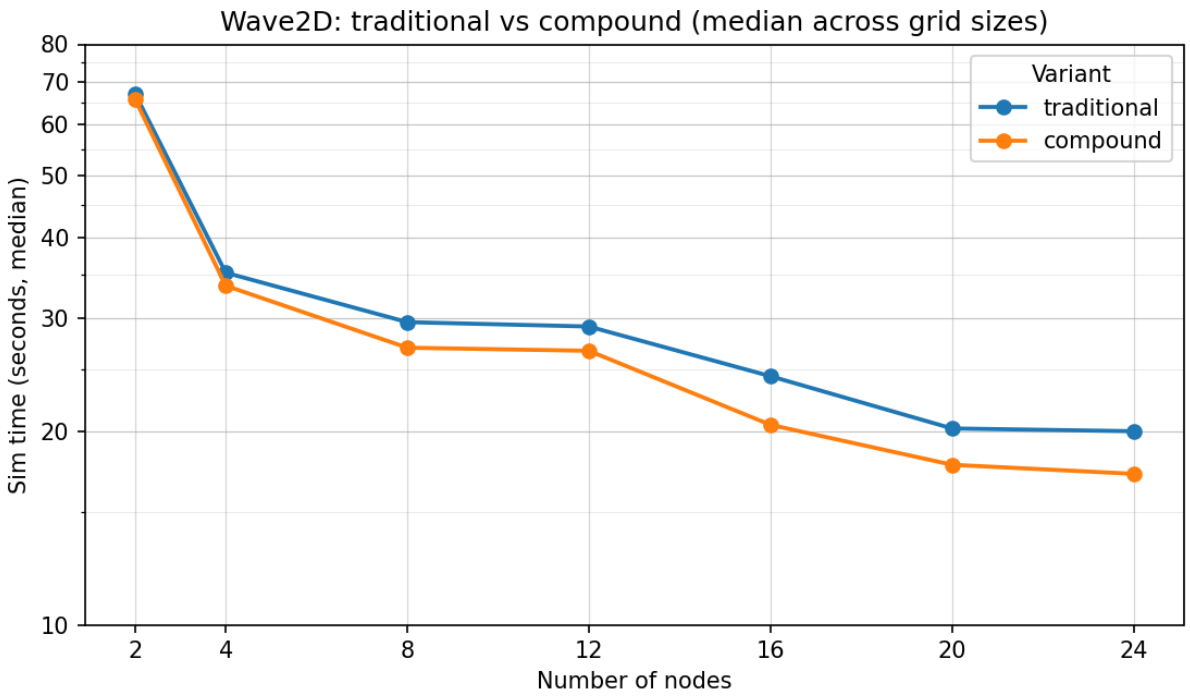

Figure 6: *Wave2D: traditional vs. compound, median across grid sizes.*

SugarScape.

SugarScape's per-iteration loop has six collective operations and mobile-agent migration, so the per-call dispatch cost is more visible than on Wave2D (Fig. 7): header lambda 0–14% and JIT 16–25% across the two grid sizes. The six dispatches per iteration amplify the JIT tier's per-call `LambdaRegistry` lookup relative to the two-dispatch Wave2D loop.

The node-count curve (Fig. 8) is U-shaped, with median wall-clock falling from two to eight nodes and then climbing back as P grows, ending higher at 24 nodes than at four. The inflection at eight nodes is the $O(P^2)$ all-to-all agent exchange limit identified in §1.1: each rank exchanges a migration buffer with every other rank per agent-management phase, so per-iteration migration traffic grows quadratically while per-rank compute shrinks linearly. Compound stays 30–50% below traditional at every P, but cannot remove the inflection, since the per-iteration master overhead it eliminates is a constant whereas the all-to-all cost is structural. The sparse neighbour-rank exchange in `GraphAgents` (evaluated under Random Walk below) is the corresponding structural fix on the graph side.

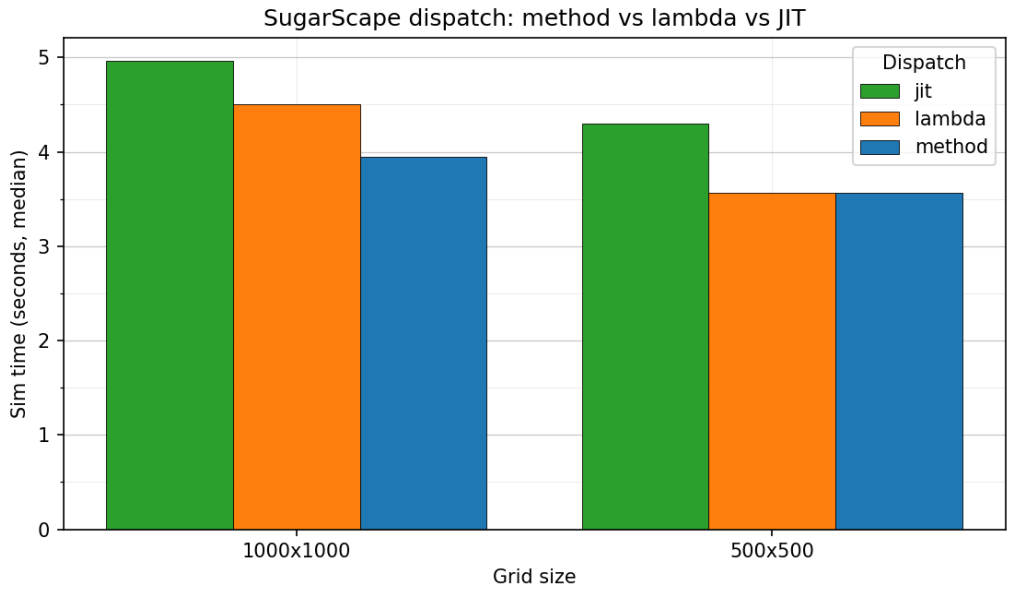

Figure 7: *SugarScape: method vs. header lambda vs. JIT.*

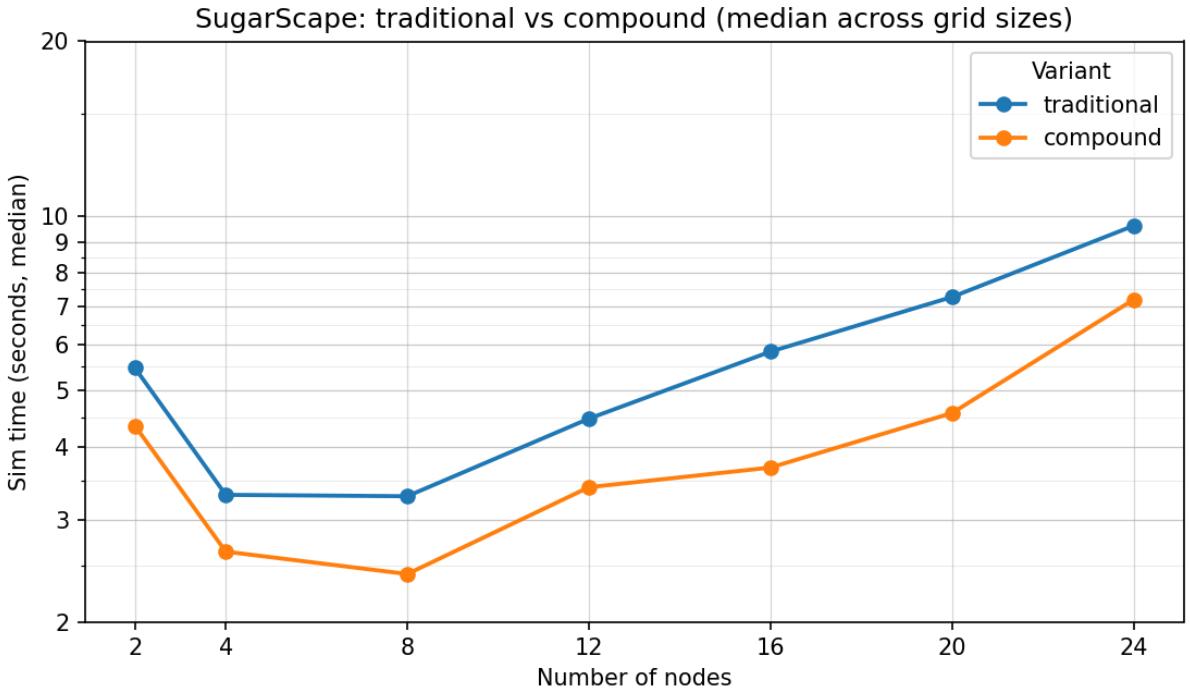

Figure 8: *SugarScape: traditional vs. compound, median across grid sizes.*

PageRank.

PageRank on web-Google and soc-Pokec (Fig. 9) shows compound 10–25% faster than traditional at every node count, with the smallest margin on soc-Pokec (∼10–13%) and the largest on web-Google (∼15–25%). The 12-to-16-node web-Google curve plateaus briefly before resuming its descent: PageRank's per-iteration traffic is proportional to the global edge-cut, which grows monotonically with P for any fixed partitioning, so at intermediate cluster sizes the added cross-rank combiner volume offsets the per-rank work reduction. The other benchmarks do not show this plateau because their per-iteration traffic is bounded by an active subset (frontier in BFS, migrating agents in Random Walk, boundary cells in Wave2D) rather than by the entire cut. Partition strategy matters too (Fig. 10). Three combinations are compared: `lex_modulo` (lexicographic ordering with modulo partitioning, the deterministic baseline with no locality), `bfs_block` (BFS-from-highest-degree ordering with block partitioning), and `dfs_block` (DFS-from-highest-degree ordering with block partitioning); all three are introduced in §4.3. On web-Google, `dfs_block` (44% edge-cut) finishes PageRank in ∼62 s versus ∼105 s for `lex_modulo` (90% cut), a 41% reduction. The same locality margin shrinks on soc-Pokec, where the higher average degree (∼18 vs. ∼6) keeps every scheme above 68% edge-cut and leaves the three schemes within a few percent of each other; `dfs_block` is marginally fastest (∼479 s) over `bfs_block` (∼500 s) despite a lower edge-cut on DFS. On dense graphs the partitioner runs out of leverage; on sparse graphs it dominates the runtime budget.

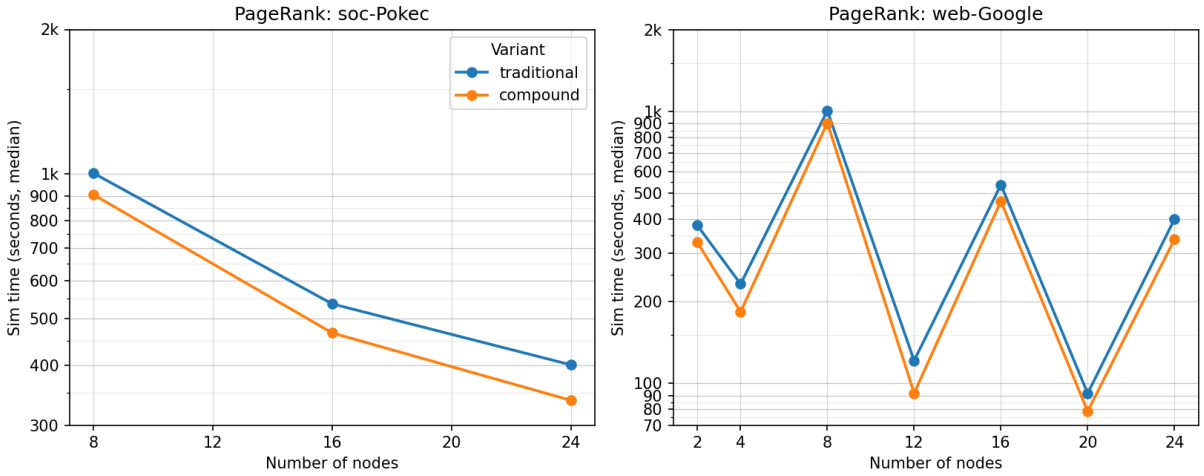

Figure 9: *PageRank: traditional vs. compound on web-Google and soc-Pokec.*

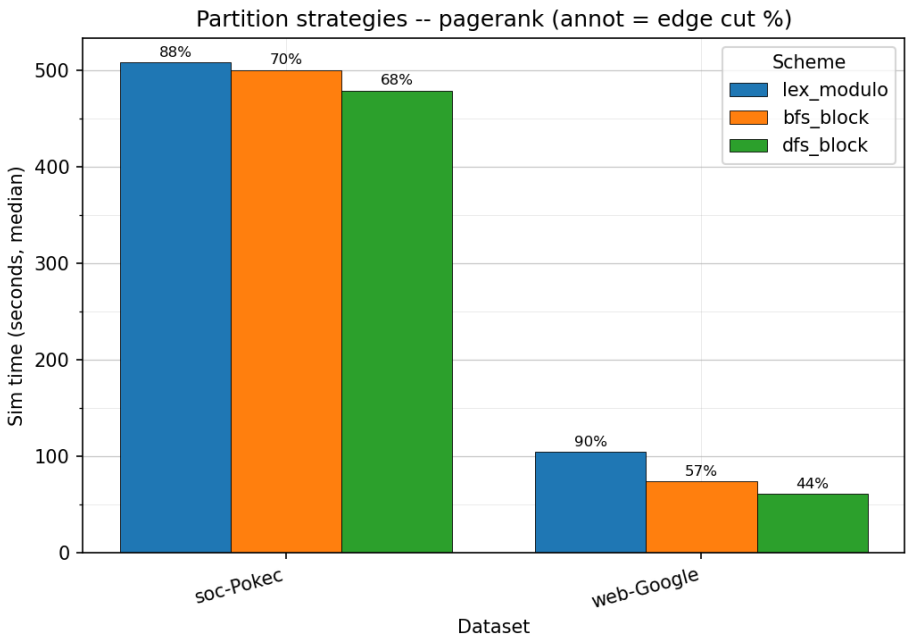

Figure 10: *PageRank partition strategies (eight nodes):* `lex_modulo`*,* `bfs_block`*, and* `dfs_block`*, with edge-cut percentage annotated above each bar.*

BFS Wavefront.

BFS Wavefront strong scaling on web-Google (Fig. 11) descends from ∼142 s at two nodes to ∼17 s at 20 nodes before plateauing, as the per-rank frontier shrinks below useful size past ∼20 nodes. Unlike PageRank, BFS shows no intermediate-P bump because its per-iteration traffic is bounded by frontier size (the set of active agents per iteration) rather than by the global edge cut. Visited counts match the reference offline BFS on every repetition, confirming correctness as defined in §5.1.

Partition strategy reverses the PageRank ordering (Fig. 12). On both datasets the highest-edge-cut scheme (`lex_modulo`) finishes BFS fastest: ∼33 s on web-Google (25% below `bfs_block` at 57% cut) and ∼190 s on soc-Pokec (17% below `bfs_block` at 70% cut). Because BFS traffic is bounded by frontier size rather than the global cut, a locality-aware partition pays for clustered work (uneven per-rank frontier sizes, load imbalance) without an offsetting saving on traffic, so the ordering flips against PageRank where the same locality reduced 41% of edge-cut traffic on web-Google.

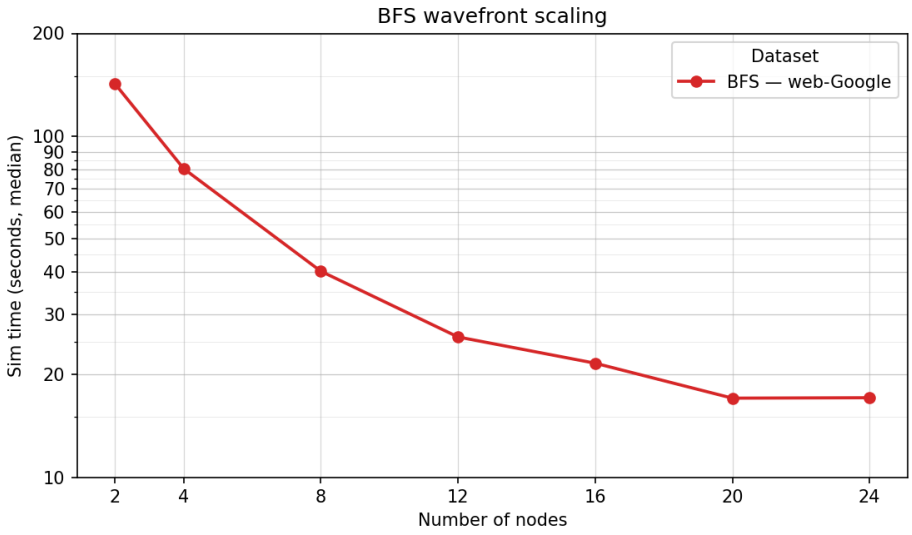

Figure 11: *BFS Wavefront strong scaling on web-Google.*

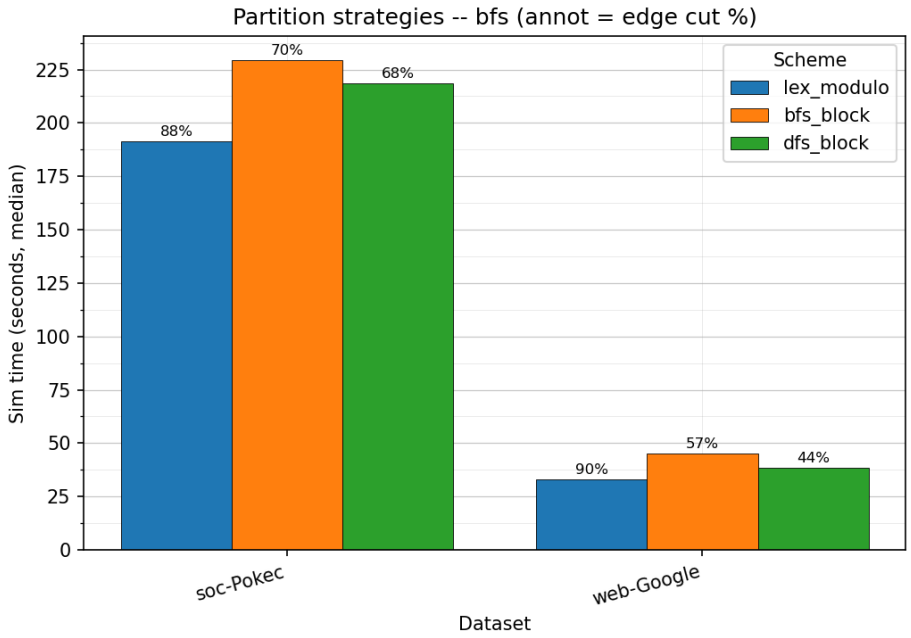

Figure 12: *BFS partition strategies (eight nodes):* `lex_modulo`*,* `bfs_block`*, and* `dfs_block`*, with edge-cut percentage annotated above each bar.*

Random Walk.

The random-walk benchmark (Fig. 13) compares grid `manageAll`, which uses the $P \times (P-1)$ all-to-all migration protocol, with the sparse graph `manageAll` built on `GraphTopology` adjacency; the compound pipeline used for the graph-side runs is reproduced in Listing 7 of

the Appendix. Both rise with P as inter-rank migration traffic grows, but graph `manageAll` stays 1.3–1.9× below grid `manageAll` throughout, with the widest gap at 24 nodes (∼1.3 s vs. ∼1.7 s median per agent phase). This is the direct payoff of the `GraphAgents` sparse neighbour-rank exchange and the structural answer to the $O(P^2)$ ceiling visible on SugarScape: the same agent-migration workload that grows quadratically on a grid topology grows only with the partition-graph degree on a graph topology. The remaining gap between the two curves reflects per-message cost, which bounds the graph variant's growth as well but at a substantially lower coefficient.

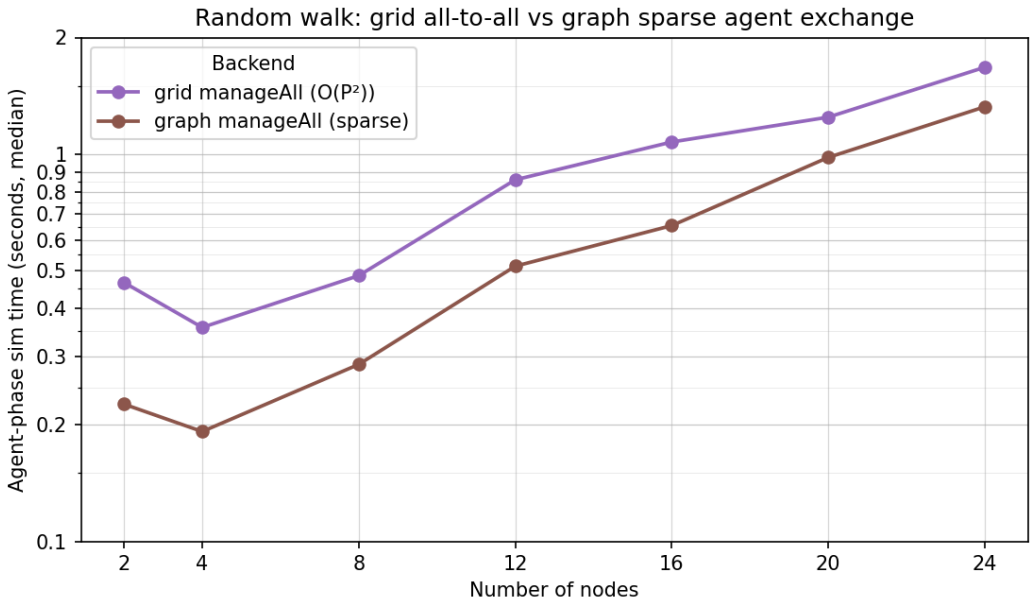

Figure 13: *Random walk: grid* `manageAll` *(all-to-all) vs. graph* `manageAll` *(sparse).*

Summary across extensions.

The three extensions were measured on different benchmarks and report different quantities (LoC for dispatch, compound-vs-traditional runtime for coordination, edge-cut and migration traffic for the graph stack), so a single averaged number would not be meaningful. Table 7 lists the headline results from the paragraphs above side by side for reference.

Table 7: *Headline results for each extension across the benchmark suite.*

Extension	**Benchmark focus**	**Reported result**
Three-tier dispatch	Developer effort (LoC)	LoC −15 to −34%; dispatch boilerplate −53 to −63% (Wave2D, SugarScape).
Compound iteration	Coordination cost	Compound vs. traditional: Wave2D 3–17%, SugarScape 30–50%, PageRank 10–25% faster.
Graph partition locality	Traffic-bound runtime	web-Google PageRank ∼41% faster (`dfs_block` at 44% cut vs. `lex_modulo` at 90%).
Sparse agent migration	Migration scaling	Graph `manageAll` 1.3–1.9× below grid all-to-all (Random Walk).

5.3 Programmability

Programmability is reported only for Wave2D and SugarScape, the two benchmarks implemented in all three new dispatch tiers; the graph-stack benchmarks of §5.2 exist only against the compound API. Lines of code (LoC) are counted as non-blank, non-comment lines in each tier's user-written class files (`.h` and `.cpp`), plus the JIT tier's lambda registration block in `main.cpp` (`setProjectHeaders` through `waitForLambdas`). Shared main-driver scaffolding (argument parsing, `MASS::init`, iteration loop, result prints) is excluded as common across tiers. A separate *dispatch boilerplate* count tallies only the pure dispatch scaffolding (signatures, dispatch-table or registration entries, integer constants and `switch` cases for native methods, per-class registration glue), excluding handler bodies, helpers, class skeleton, and includes. Reductions are reported against the native-method baseline.

Wave2D's five dispatched handlers decline from 160 LoC for native methods to 123 for header lambda (−23%) and 105 for JIT (−34%). Each body is short relative to its signature and registration scaffolding, so removing more of the scaffolding (header lambda eliminates the in-class signature; JIT additionally moves the body out of the class) yields a larger reduction.

SugarScape distributes nine handlers across `SugarPlace` (4) and `AntAgent` (5). LoC declines from 227 to 181 for header lambda (−20%) and 192 for JIT (−15%). The two lambda tiers are closer here because the per-handler bodies are long enough to dilute the scaffolding savings; JIT also lags header lambda slightly because each out-of-class body needs explicit class qualifiers for static constants and member calls, partially offsetting the saving of moving

bodies out of the class.

Across both benchmarks, lambda tiers cut LoC by 15–34% and dispatch boilerplate by 53–63% against native methods (Wave2D: 40 → 15/19; SugarScape: 73 → 29/33), so even SugarScape, where total LoC barely moves, sees more than half of its per-handler routing code removed. Beyond line counts, the lambda tiers add inline authoring (no signature/body split), framework-version portability (string names instead of integer identifiers), and recompile-free iteration (JIT only), giving the developer three calibrated points on one spectrum rather than a binary choice.

Table 8: *Lines of code and dispatch boilerplate per tier; reductions vs. native methods in parentheses.*

Benchmark	**Lines of code**			**Dispatch boilerplate**		
	native	**header λ**	**JIT λ**	**native**	**header λ**	**JIT λ**
Wave2D	160	123 (−23%)	105 (−34%)	40	15 (−63%)	19 (−53%)
SugarScape	227	181 (−20%)	192 (−15%)	73	29 (−60%)	33 (−55%)

5.4 Comparison with Industry Standards

HPX is the closest C++ analogue to MASS C++ on the surveyed axes (task-based runtime, MPI transport, no agent model). The comparisons below use three matched HPX implementations developed by Pay as part of a concurrent UWB MS thesis [13], used here with permission: Heat3D for the stencil regime, three motif queries (triangle, star3, square) for the graph-search regime, and SugarScape for the single-node agent regime. All numbers are simulation-time medians on the same cluster as §5.1.

Figure 14 reports Heat3D on a 768^3 grid. MASS C++ is faster than HPX at every rank count, with a 24% gap at rank=1 for 100 timesteps and 43% at rank=1 for 512 timesteps, reflecting HPX's per-call runtime overhead before parallelism amortises it. Both implementations scale well to about 8 ranks; communication then dominates and the curves flatten. HPX failed to launch at 32 ranks; MASS C++ reached 32 before hitting the same ceiling at 48.

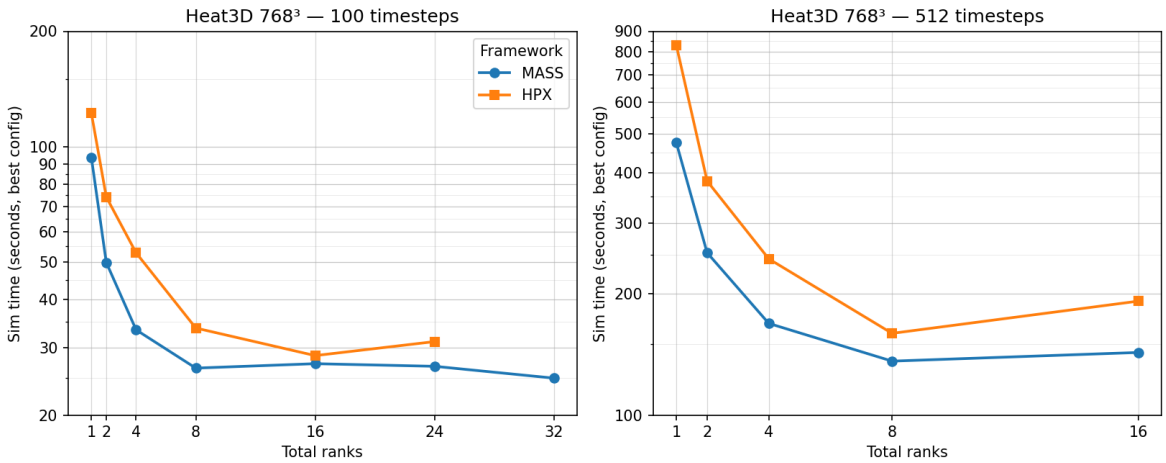

Figure 14: *Heat3D (768^3): MASS C++ vs. HPX simulation time at 100 and 512 timesteps.*

Figure 15 reports three motif queries on the same graph at 1 to 64 ranks. Triangle and star3 track within ∼5% across the full range, with MASS C++ slightly ahead at the high end. On square, HPX is marginally faster at low-to-mid rank counts, but the gap closes at 48 to 64 ranks. Both show near-linear speedup to about 16 ranks and diminishing returns thereafter as per-query coordination overtakes the search work.

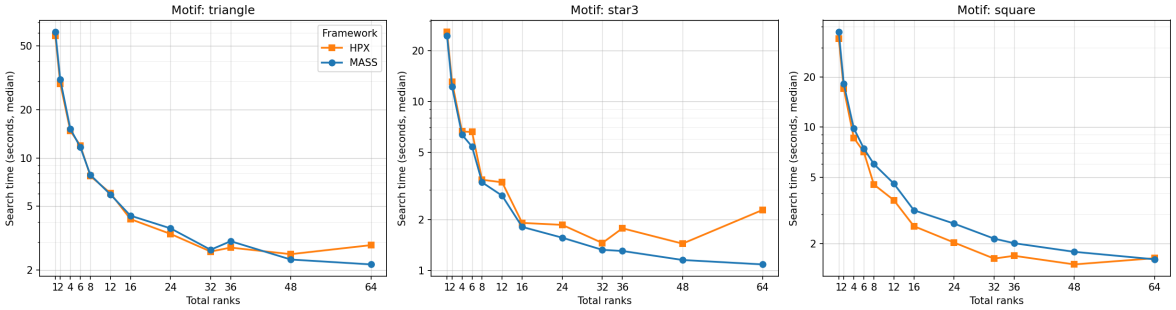

Figure 15: *Motif search (triangle, star3, square): MASS C++ vs. HPX median search time.*

Figure 16 reports a single-node comparison (5,000 × 5,000 grid, 1 M agents) at processes per node from 1 to 8. MASS C++ is faster at ppn=1 (∼9%); HPX wins at ppn=2 (∼9%); MASS C++ is then 26 to 30% faster from ppn=4 onward. HPX flatlines between 380 and 420 k ms across ppn=4 to 8, indicating a scalability wall for its agent representation under intra-node process scaling on this workload.

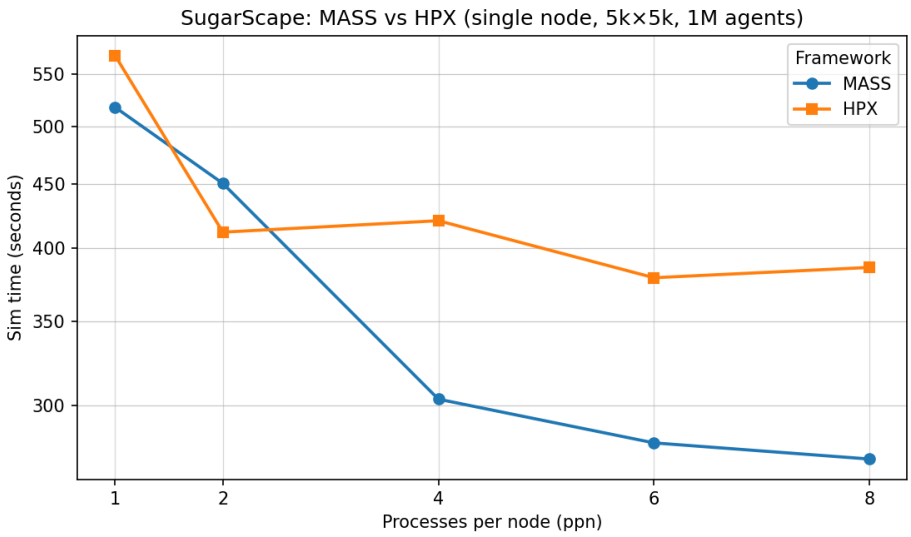

Figure 16: *SugarScape (*5,000 × 5,000*, 1 M agents, single node): MASS C++ vs. HPX by processes per node.*

Across the three workloads, MASS C++ wins on stencil and intra-node agent runs and is close to parity on motif search. The single-rank Heat3D gap (24 to 43%) is the clearest evidence that HPX's per-call runtime overhead is the dominant cost at low parallelism. The motif results suggest HPX's task-based scheduling is well-matched to irregular graph search; the SugarScape result suggests it does not scale comparably on agent populations that exchange state every iteration. PageRank, BFS Wavefront and Random Walk are not compared here because no matched HPX implementation is available.

5.5 Limitations and Threats to Validity

The current `AsyncExecutor` serialises every `AsyncHandle` onto a single thread per process. This sidesteps the synchronisation cost that a concurrent executor would otherwise incur on shared internal state (the `ExchangeHelper` message tables, the `LambdaRegistry`, the per-rank logging buffer), but it also forfeits any overlap between successive collective operations within an iteration. The compound speedups in §5.2 are therefore lower bounds against a hypothetical concurrent executor.

JIT-tier numbers include the one-time cost of compiling each registered lambda the first time it is called in a run, after which the compiled object resides in the shared-filesystem cache. The medians of three repetitions amortise but do not eliminate this cost: a run on a cold cache (after a manual flush or a fresh checkout) is tens of seconds slower at start-up than the

steady-state runs. All §5.2 numbers are reported as simulation time (excluding setup), so the cache state affects only the totalTimer column of the underlying CSV.

The per-phase recorder used during development adds ∼18% overhead (Fig. 17: Wave2D, 1000×1000, eight nodes, ∼8.9 s with recording off versus ∼10.5 s with recording on); it is disabled by default, and all simulation-time numbers in §5.2 are collected with the recorder off.

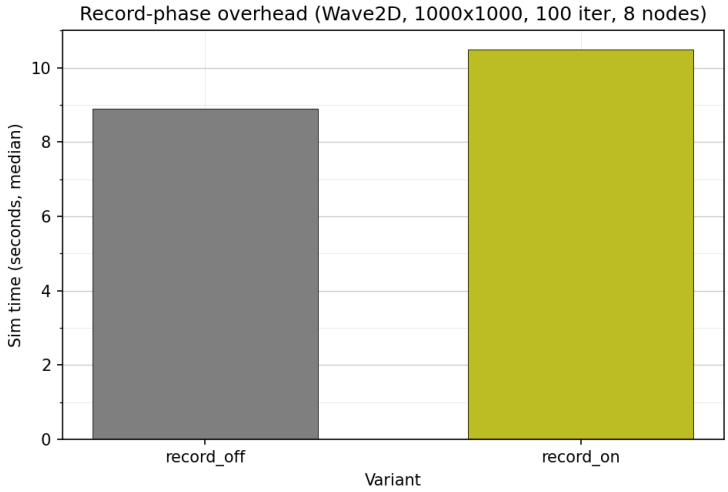

Figure 17: *Per-phase recorder overhead.*

`IterationConfig` requires the user to declare the entire phase pipeline before any iteration starts, with no in-pipeline branching. Workloads that change their per-iteration structure based on runtime state (adding a new exchange phase when a convergence metric drops, or skipping the agent-management phase on iterations with no pending migrations) cannot be expressed through the compound path and must fall back to the traditional dispatch loop. The five benchmarks in §5.2 all have static per-iteration structure and therefore do not exercise this limitation.

The grid-only `manageAll` protocol exchanges a migration buffer between every pair of ranks per agent-management phase, so per-iteration agent traffic grows as $P \times (P-1)$. SugarScape makes this concrete (§5.2): the wall-clock curve turns upward at eight nodes and never returns to the eight-node minimum. The structural fix on the graph side is the sparse neighbour-rank exchange in `GraphAgents` (Random Walk paragraph of §5.2); on the grid side, no equivalent fix is in this thesis, and the same scaling ceiling applies to any grid simulation whose iteration includes mobile agents.

The dispatch and graph extensions assume that the master and every worker see identical

paths on a shared filesystem: the master writes compiled lambdas and graph partition files there, and every worker reads them from the same location. Deployments without a shared mount (per-node scratch storage with no NFS, or containerised workers without a bind-mount) would need an explicit pre-run file distribution step, which the framework does not provide. The assumption is documented as an operational prerequisite rather than a hard limitation.

§5.3 counts only lines of code and dispatch boilerplate. Those counts are reproducible but say nothing about correctness, maintainability, or how hard it is to pick among the three dispatch tiers, a custom partitioner, or a combiner. More API surface also means more ways to misconfigure a run; the compile-time dispatch table, block partitioning, and the no-op combiner are the defaults, and `IterationConfig` fixes phase ordering at configuration time rather than at each call site. A controlled user study would be needed to claim anything beyond reduced boilerplate; the LoC numbers in §5.3 should be read as that narrower result only.

6 Conclusion

This thesis presented three extensions that move MASS C++ from a fixed-topology, integer-dispatched simulation framework toward a general-purpose distributed runtime. The dispatch overhaul replaces fragile integer call identifiers with a three-tier name-based system; the compound execution model collapses the $K \times N$ master barriers of an iterative simulation into a single dispatch; and the graph stack lifts the grid-only restriction with bidirectional adjacency, locality-aware partitioning, and edge-constrained agent migration. §5.2 and §5.5 measured the gains and bounded the conclusions of each extension on five benchmarks.

6.1 Summary

The dispatch extension routes every call through a name-keyed table generated at compile time, a header-defined lambda registry, and a JIT-compiled lambda registry, in that priority order. §5.2 shows the resulting per-call overhead under ∼15% on Wave2D and rising to ∼25% on SugarScape's operation-dense iterations. §5.3 reports the corresponding programmability gain on the two benchmarks implemented in all three tiers: total user-written LoC falls 15–34% (Wave2D 160 → 105–123; SugarScape 227 → 181–192), and dispatch boilerplate falls 53–63% (Wave2D 40 → 15–19; SugarScape 73 → 29–33), with the larger saving on routing code because handler bodies are short relative to it.

The compound extension exposes `AsyncHandle`, `AsyncExecutor`, and `IterationConfig`, which together let the master dispatch an entire iteration as a single asynchronous call. §5.2 reports compound beating traditional by 3–17% on Wave2D (gap widening with cluster size), 30–50% on SugarScape, and 10–25% on PageRank. The compound gain does not affect per-iteration boundary exchange, all-to-all agent migration, or combiner traffic: it removes only the constant per-iteration master coordination overhead.

The graph extension introduces `GraphTopology` with bidirectional adjacency, `GraphPlaces` for vertex distribution with neighbour-only message exchange, and `GraphAgents` for edge-constrained migration. Combiners, named distributed aggregators, and a pluggable parser interface complete the stack. §5.2 shows that partition choice is workload-dependent: on traffic-bound PageRank, `dfs_block` finishes web-Google in ∼62 s versus ∼105 s for `lex_modulo` (41% faster at 44% vs. 90% edge-cut), while on frontier-bound BFS Wavefront the ordering reverses and `lex_modulo` finishes ∼25% below `bfs_block` on web-Google and ∼17% below on soc-Pokec because lower edge-cut does not offset clustered frontier work. Sparse graph `manageAll` stays 1.3–1.9× below the grid all-to-all across every cluster size, the structural

answer to the $O(P^2)$ ceiling surfaced by SugarScape.

Against HPX as an external baseline (§5.4), MASS C++ leads on stencil (Heat3D: 24–43% faster at rank 1, faster at every rank count tested) and intra-node agent runs (SugarScape: 26–30% faster from ppn=4 onward while HPX flatlines), with motif search within ∼5% across triangle, star3, and square queries.

Taken together, the extensions place MASS C++ in a workload class not directly served by any system surveyed in Section 3: mobile agents over irregular topologies, with grid and graph simulations sharing one runtime, one collective API, and one backward-compatible dispatch path. The original integer dispatch, blocking collectives, and grid-only `Places` all continue to function for legacy code, so adoption is incremental rather than wholesale.

6.2 Limitations

Three functional gaps in the current implementation are out of scope for this thesis but are worth recording explicitly.

Fault tolerance.

The runtime provides no mechanism for ranks lost to preemption, network failure, or out-of-memory events. Long-running simulations on shared clusters tolerate such failures only by manual restart from the beginning.

Runtime topology mutation.

`GraphTopology` is immutable after load, and the combiner and aggregator hooks of §4.3 do not expose vertex- or edge-mutation entry points. Workloads whose edge set evolves with the simulation (contact-network epidemiology, for example) cannot be expressed through the current API.

Single graph per run.

The graph stack supports one `GraphTopology` per process. Workloads that operate on multiple graphs simultaneously (cross-graph agent migration in a layered transportation model, or hybrid grid-plus-graph simulations) cannot be expressed without manual stitching outside the runtime.

6.3 Future Work

Five extensions follow directly from the limitations of §5.5 and §6.2.

Phase overlap.

A two-buffer scheme that posts the next exchange while the current compute phase runs would extend compound's gain past the $K \times N$-to-1 master saving. The `AsyncHandle` infrastructure of §4.2 already provides the synchronisation primitive, but `IterationConfig` needs a way to declare which phases may overlap.

Checkpoint-based fault tolerance.

Pregel+ handles rank loss with periodic graph-state checkpoints and rank-replacement on failure. A similar mechanism on top of the `IterationConfig` phase boundary would slot into the MASS C++ pipeline, which already provides the consistent global snapshot point that a checkpointer would need.

Runtime topology mutation.

A mutation layer that allows running computation to add or remove vertices and edges between supersteps (as Pregel allows) would broaden the workload class MASS C++ can express, and is a sensible follow-on once the static-graph performance baseline of §5.2 is settled.

Grid-side sparse migration.

Two grid-side fixes are plausible for the $O(P^2)$ grid-`manageAll` ceiling identified on SugarScape: routing agents through a sparse logical overlay rather than the full all-to-all, and pooling cross-rank connections so that small per-iteration migration buffers do not each pay full TCP setup cost. Both would interact with the `ExchangeHelper` handshake protocol described in §4.4.

Multi-graph workloads.

Concurrent `GraphTopology` instances per process, a routing layer that maps an agent to its current topology, and an extension to `IGraphParser` for loading multiple graph files in one pipeline would support layered transportation models and hybrid grid-plus-graph workloads. None of the three is conceptually difficult, but each touches a different layer of the runtime.

References

[1] M. Fukuda, A. Kamil, S. Gupta, C. Tessier, and C. Hanks, "MASS: A parallelization library for multi-agent spatial simulation," in *Proceedings of the 2013 IEEE International Conference on Granular Computing (GrC)*, IEEE, 2013, pp. 283–288. DOI: 10.1109/GrC.2013.6740425

[2] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990. DOI: 10.1145/79173.79181

[3] H. Kaiser et al., "HPX: The C++ standard library for parallelism and concurrency," *Journal of Open Source Software*, vol. 5, no. 44, p. 2352, 2020. DOI: 10.21105/joss.02352

[4] T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, and H. Kaiser, "HPX—an open source C++ standard library for parallelism and concurrency," in *Proceedings of the International Workshop on OpenCL*, ACM, 2017, pp. 1–12. DOI: 10.1145/3078155.3078156

[5] P. Grubel, H. Kaiser, J. Cook, and A. M. Serio, "The performance implication of task size for applications on the HPX runtime system," *IEEE Cluster Computing*, 2015. DOI: 10.1109/CLUSTER.2015.140

[6] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014. DOI: 10.14778/2733085.2733103

[7] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web (WWW)*, ACM, 2015, pp. 1307–1317. DOI: 10.1145/2736277.2741096

[8] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012. DOI: 10.14778/2212351.2212354

[9] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ACM, 2010, pp. 135–146. DOI: 10.1145/1807167.1807184

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX Association, 2012, pp. 17–30.

[11] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[12] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. DOI: 10.1137/S1064827595287997

[13] A. B. Pay, "A benchmark of C++ cluster-computing libraries: MASS C++, HPX, and PM2," MS thesis, in preparation, M.S. thesis, University of Washington Bothell, 2026.

A API Reference and Code Examples

Table 9: *`IterationConfig` phase pipeline builders.*

Builder	Description
`iterations(n)`	Number of outer iterations to execute autonomously on workers.
`checkpoint(K)`	Master sync every K iterations (population, return values).
`convergence(method, threshold)`	Early stop when a global delta falls below `threshold`.
`placeCompute(method)`	`callAll` on the host Places container.
`placeExchangeBoundary()`	Boundary exchange between adjacent grid ranks.
`placeExchangeAll(dest, method)`	Full `exchangeAll` to another Places handle.
`agentCompute(handle, method)`	`callAll` on an Agents collection.
`agentManage(handle)`	`manageAll` (spawn, kill, migrate) on an Agents collection.
`graphExchangeNeighbors(method)`	Topology-routed neighbour exchange on `GraphPlaces`.
`recordPlaces(method, retSize)`	Append per-place snapshots to rank-local files.
`recordAgents(handle, method, retSize)`	Append per-agent snapshots to rank-local files.

Listing 2: *Compound iteration pipelines for Wave2D, SugarScape, and PageRank.*

```
#include "IterationConfig.h"

// Wave2D: Places-only stencil (2 phases/iteration)
mass::IterationConfig wave2d;
wave2d.iterations(200)
      .placeCompute("Wave2DCompound::computeWaveAndLoad")
      .placeExchangeBoundary();
MASS::runIterations(PLACES_HANDLE, wave2d);

// SugarScape: Places + Agents (6 phases/iteration)
mass::IterationConfig sugar;
sugar.iterations(200)
      .agentCompute(AGENTS_HANDLE, "AntAgentCompound::eat")
      .agentCompute(AGENTS_HANDLE, "AntAgentCompound::move")
```

```
        .agentManage(AGENTS_HANDLE)
        .agentCompute(AGENTS_HANDLE, "AntAgentCompound::aging")
        .agentManage(AGENTS_HANDLE)
        .placeCompute("SugarPlaceCompound::regrow");
MASS::runIterations(PLACES_HANDLE, sugar);

// PageRank: graph neighbour exchange + local gather
mass::IterationConfig pagerank;
pagerank.iterations(100)
          .graphExchangeNeighbors("PageRankVertex::scatter")
          .placeCompute("PageRankVertex::gather");
MASS::runIterations(PLACES_HANDLE, pagerank);
```

Table 10: *GraphVertex API: accessors inherited by user-defined vertex classes.*

Accessor	**Description**
`getGlobalIndex()`	Global vertex index across all ranks.
`getTotalVertexCount()`	Total vertices in the distributed graph.
`getNeighborCount()`	Number of adjacent vertices (out-edges).
`getNeighborGlobalIndex(i)`	Global index of the i-th neighbor.
`getEdgeWeight(i)`	Weight of the i-th edge.
`isDirected()`	Whether the graph is directed.
`isLocalNeighbor(i)`	Whether the i-th neighbor resides on the same rank.
`getOwnerRank(idx)`	Rank that owns the given global index.
`getEdges()`	Full edge list as `vector<IndexedEdge>`.

Table 11: *Index ordering strategies and their partitioning effects.*

Order	**Best Paired With**	**Effect**
Lexicographic	Modulo	Original vertex order from file; no locality optimization.
BFS	Block	Breadth-first from highest-degree vertex; keeps neighbors on consecutive indices.
DFS	Block	Depth-first traversal; similar locality benefits, different clustering shape.

Table 12: *Built-in graph parser formats.*

Format ID	Extensions	Description
`edgelist`	`.edges`, `.txt`	One edge per line; supports SNAP-style integer IDs, string IDs, optional weights. Two-pass loading for large graphs.
`adjlist`	`.adj`, `.adjlist`	One vertex per line with its neighbor list; optional weights.
`csv`	`.csv`	Comma-separated source, target, optional weight; configurable column indices.
`hippie`	`.txt`	HIPPIE protein-protein interaction format; tab-separated with confidence scores.

Listing 3: *GraphVertex subclass with dispatch table and scatter/gather methods.*

```
class PageRankVertex : public GraphVertex {
public:
    PageRankVertex(void* argument) : GraphVertex(argument) {}

    MASS_DISPATCH_TABLE(PageRankVertex,
        MASS_METHOD(PageRankVertex, init),
        MASS_METHOD(PageRankVertex, scatter),
        MASS_METHOD(PageRankVertex, gather)
    )

    void* scatter(void* argument) {
        contribution_ = rank_ / getNeighborCount();
        outMessage = &contribution_;
        outMessage_size = sizeof(double);
        return nullptr;
    }

    void* gather(void* argument) {
        double sum = 0.0;
        for (int i = 0; i < inMessage_size; i++)
            sum += *(double*)inMessages[i];
        rank_ = 0.15 / getTotalVertexCount() + 0.85 * sum;
        return nullptr;
    }

private:
```

```
    double rank_ = 0.0;
    double contribution_ = 0.0;
};
```

Listing 4: *GraphPlaces initialization from a graph file.*

```
GraphPlacesConfig gpConfig;
gpConfig.handle = placesHandle;
gpConfig.className = "PageRankVertex";
gpConfig.graphFile = "data/web-Google.txt";
gpConfig.format = "edgelist";
gpConfig.directed = true;
gpConfig.indexOrder = IndexOrder::BFS;
gpConfig.partitionScheme = PartitionScheme::BLOCK;

GraphPlaces* gp = new GraphPlaces(gpConfig);
```

Listing 5: *GraphAgents construction, binding agents to a GraphPlaces instance.*

```
#include "graph/GraphAgents.h"

GraphAgents* agents = new GraphAgents(
    AGENTS_HANDLE,    // unique Agents handle
    "WalkerAgent",    // shared library class name
    nullptr, 0,       // constructor argument and size
    *gp,              // reference to GraphPlaces
    1000              // initial agent population
);
```

Listing 6: *Custom graph parser registration via macro.*

```
#include "graph/parsers/GraphParserFactory.h"

REGISTER_GRAPH_PARSER("myformat", MyFormatParser)
// MyFormatParser must extend IGraphParser and implement parse()
```

Listing 7: *Random walk compound pipeline with recording.*

```
IterationConfig config;
config.iterations(10)
      .agentCompute(AGENTS_HANDLE, "WalkerAgent::walk")
```

```
4         .agentManage(AGENTS_HANDLE)
5         .recordAgents(AGENTS_HANDLE, "WalkerAgent::exportData",
6                       sizeof(ExportRecord))
7         .recordDir("exports/recording");
8 MASS::runIterations(placesHandle, config);
```