A Benchmark of C++ Cluster-Computing Libraries: MASS C++, HPX, and PM2

Ahmed Bera Pay

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

Master of Science

University of Washington

2026

Reading Committee:

Munehiro Fukuda

David Socha

Brent Lagesse

Program Authorized to Offer Degree:

Computer Science and Software Engineering

University of Washington

Abstract

A Benchmark of C++ Cluster-Computing Libraries:
MASS C++, HPX, and PM2

Ahmed Bera Pay

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

The selection of a distributed C++ runtime for high-performance cluster applications entails significant performance and programmability trade-offs, yet existing evaluations of higher-level runtimes remain siloed within each runtime's own application domain, precluding workload-diverse, controlled cross-runtime comparison. This thesis presents a benchmark suite of five computationally distinct patterns—a 3D stencil kernel (Heat3D), dense matrix multiplication (DGEMM), graph motif search, a phased financial graph simulation (Bail-In/Bail-Out, hereafter BIBO), and an agent-based model (SugarScape)—implemented equivalently across MASS C++, HPX, and PM2 (via MadMPI) from runtime-neutral algorithmic specifications. Runtime behavior is characterized along two dimensions: performance, measured through strong-scaling experiments across computing-node counts and instances-per-node configurations visualized as 3D surfaces, and programmability, assessed through objective code metrics (cyclomatic complexity, lines of code, and function-size distribution) applied consistently across all fifteen benchmark-runtime combinations. Key findings include MASS C++'s dominance on structured spatial workloads (SugarScape: 16.2× at 96 parallel processes (ranks); superior programmability on structured workloads), PM2's superiority on dense matrix multiplication (17× SUMMA speedup (ratio of single-process to parallel time) from explicit communication overlap), and HPX's leading programmability on irregular workloads.

This work constitutes the first algorithmically equivalent cross-runtime characterization of MASS C++, HPX, and PM2, yielding empirically grounded, workload-specific guidance for distributed C++ runtime selection.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

High-performance distributed computing traditionally relies on the Message Passing Interface (MPI), which requires programmers to manage all inter-node data transfer explicitly through point-to-point sends, receives, and collective operations. While expressive, this low-level interface demands substantial expertise: buffer management, non-blocking communication overlap, and collective synchronization must all be handled by the programmer, making programs difficult to write, debug, and maintain as problem complexity grows.

Higher-level distributed C++ runtimes have emerged to address this gap, encapsulating communication, scheduling, and synchronization behind richer abstractions. However, runtime adoption in production settings is hindered by the absence of principled, side-by-side performance and programmability comparisons across diverse workloads. Practitioners selecting a runtime for a new application must rely on single-application evaluations or design-time assumptions rather than empirical evidence spanning multiple workload classes.

This thesis compares three such runtimes—MASS C++ [2], HPX [3], and PM2 [**?**]—each representing a distinct execution paradigm. MASS C++ provides a synchronous place-agent model designed for structured spatial simulation; HPX implements an asynchronous many-task runtime aligned with the C++ standard; and PM2 supports distributed multithreading with user-level thread migration for dynamically irregular workloads. These three runtimes occupy different points in the design space with dimensions of abstraction, scheduling granularity, and communication model, making their joint evaluation informative for a wide range of application developers.

The selection of these three runtimes is motivated by three complementary criteria. First, *paradigm uniqueness*: MASS C++, HPX, and PM2 represent three fundamentally different

execution models—bulk-synchronous place-agent, asynchronous many-task, and distributed multithreading respectively—none of which is reducible to the others. Choosing runtimes with overlapping paradigms (e.g., two asynchronous task runtimes) would narrow the evaluation without broadening its coverage of the runtime design space. Second, *prior deployment context*: each runtime has been applied to at least one of the benchmark domains in this thesis—MASS C++ to agent-based spatial simulation [4], HPX to scientific compute kernels [3], and PM2/MadMPI to communication-intensive workloads [**?**]—making the comparison grounded in demonstrated capability rather than speculation. Third, *availability and support*: all three runtimes were available on the evaluation cluster (Hermes, described in Section 5.1) with maintainer support, enabling reliable installation, reproducibility, and debugging. The three selected runtimes collectively provide a coverage of the C++ distributed runtime design space accessible under the resource constraints of this thesis.

The comparison is grounded in a benchmark suite of five computational patterns—Heat3D (stencil/CFD) [5], DGEMM (dense matrix multiplication) [6], Graph Motif Search [7], Bail-In/Bail-Out (phased financial simulation) [8], and SugarScape (agent-based model) [9]—each implemented uniformly across all three runtimes from runtime-neutral algorithmic specifications. This design ensures that observed performance and programmability differences reflect runtime behavior rather than incidental algorithmic variation between implementations.

The central research questions driving this work are: (1) how do MASS C++, HPX, and PM2 perform and scale across workloads that stress different aspects of distributed execution; (2) what programmability trade-offs does each runtime impose, characterized through quantitative code metrics and structural analysis of implementation patterns; and (3) for which workload classes does each runtime offer the best balance of performance and programmability?

The remainder of this thesis is organized as follows. Chapter 2 provides background on distributed-memory computing and the three runtimes. Chapter 3 surveys related benchmark suites and prior runtime evaluations, and identifies three dimensions on which this work differentiates itself from prior art. Chapter 4 describes the benchmark designs and per-runtime

implementations. Chapter 5 defines the experimental methodology and evaluation metrics. Chapter 6 presents results. Chapter 7 synthesizes cross-runtime findings. Chapter 8 concludes with recommendations and future directions.

Notation and Terminology

Throughout this thesis, `fixed-width (monospace) font` denotes runtime API identifiers, function names, and command-line flags exactly as they appear in source code (e.g., `callAll`, `hpx::async`, `MPI_Isend`). All other technical terms are set in roman type.

The terms and abbreviations in Table 1 are used consistently throughout this thesis. Each term is defined briefly here and elaborated further where it is first used in context.

Table 1: Key terminology and notation.

Term / Symbol	**Definition**	**Runtime-specific meaning**
Node (computing node)	A physical machine in the Hermes cluster.	Same across all runtimes. *Never* a graph vertex.
Graph vertex	A vertex in a graph (Motif Search only).	Always called "vertex", never "node".
`ipn`	Instances per node: parallel execution units within one computing node.	MASS: `nThreads` (POSIX threads, "pthreads") per process. HPX: localities per node (each a separate OS process with its own HPX thread pool). PM2: MPI ranks per node.
Total ranks	nodes × `ipn`.	MASS: threads; HPX: localities; PM2: MPI ranks.
Speedup (Sp)	$T_{1,1}/T_{n,p}$: wall-clock time at **one** node / **one** instance per node, divided by time at n nodes and p instances per node. Both subscripts index the same runtime's own measurements.	Computed independently per runtime from its own single-process baseline.
`best-ipn` envelope	For each node count, the minimum wall-clock time across *all* tested `ipn` values at that node count.	Shown as solid markers in scaling figures; represents the fastest observed configuration at each node count regardless of `ipn`.
Ideal speedup	Linear scaling: k nodes give exactly $k\times$ speedup.	Shown as dotted line in scaling figures; assumes zero communication cost.

Chapter 2

BACKGROUND

2.1 Distributed-Memory Parallel Computing

In distributed-memory systems each compute node maintains its own private address space, and all data sharing between nodes requires explicit message passing, placing the full burden of communication management on the developer [10]. This architecture is the foundation of modern HPC clusters, where nodes are connected by high-speed interconnects such as InfiniBand or Ethernet, and the programmer must reason explicitly about which data lives on which node and when it must be transferred.

MPI provides point-to-point and collective communication primitives that are expressive but verbose, while OpenMP extends shared-memory thread-level parallelism within a single node but does not address inter-node communication [10]. The MPI+OpenMP hybrid parallelization model is dominant in HPC, but it demands significant low-level expertise: buffer management, non-blocking communication overlap, and collective synchronization must all be handled manually, making programs difficult to write, debug, and maintain as problem complexity grows.

The programmability cost of MPI has motivated a generation of higher-level distributed C++ runtimes that retain performance while reducing boilerplate (code mandated by the runtime API rather than by the algorithm itself, such as buffer allocation, type registration, and communicator setup), synchronization complexity, and error-prone buffer management. MASS C++, HPX, and PM2 represent three distinct approaches to this goal, each with a different abstraction strategy, synchronization model, and target workload class.

2.2 MASS C++: Synchronous Place-Agent Model

MASS C++ exposes two core abstractions: Places, stateful objects arranged in a distributed n-dimensional grid, and Agents, mobile objects that reside on Places and can migrate between them [2]. Together these abstractions support structured spatial simulations without exposing raw inter-node communication to the programmer. The runtime handles data distribution, inter-node transfers, and process management transparently.

Computation in MASS C++ proceeds through `callAll` operations that invoke the same function on every Place or Agent in parallel, followed by an implicit global synchronization barrier before the next phase begins. This bulk-synchronous execution model ensures that all nodes advance in lockstep, simplifying reasoning about program correctness at the cost of scheduling flexibility.

Data is passed between the programmer and the runtime through a `void*` interface using plain-old-data structs, and SSH-based process management initializes the distributed runtime. While this interface is straightforward in concept, it requires manual casting and careful struct layout, contributing to high boilerplate ratios relative to runtimes with automatic serialization.

2.3 HPX: Asynchronous Many-Task Runtime

HPX implements the ParalleX execution model, targeting fine-grained asynchronous task execution at scale through `hpx::future<T>`, lightweight threads, and plain actions registered via `HPX_PLAIN_ACTION` and dispatched with `hpx::async` [3]. Actions are C++ functions registered at program startup; calling them remotely on another locality (process) is syntactically similar to calling them locally, with serialization handled automatically through C++ template machinery.

HPX exposes a global address space across localities, performance counters for profiling, and a work-stealing scheduler—a dynamic load-balancing strategy in which idle threads "steal" tasks from the queues of busy threads, reducing idle time without programmer intervention.

Computation and communication can overlap naturally through future chaining: a task can be submitted as soon as its inputs are available, without waiting for unrelated work to complete.

Unlike MPI's collective synchronization, HPX coordinates through future resolution and action completion, enabling fine-grained load balancing and eliminating the need for explicit global barriers in most communication patterns. This flexibility comes at the cost of increased reasoning complexity: the programmer must track which futures must be resolved before a given computation is safe to proceed.

2.4 PM2: Distributed Multithreading and NewMadeleine Communication

PM2 is a distributed runtime built around two capabilities: user-level thread migration for dynamic load rebalancing, and high-performance asynchronous communication through its native protocol library, NewMadeleine (nmad) [**?**]. NewMadeleine is a multi-rail, adaptive communication engine that progresses posted non-blocking operations in the background, reducing latency for workloads that issue many concurrent messages.

PM2 as used in this thesis. All PM2 benchmarks are written against MadMPI—the standard MPI interface layer built on top of NewMadeleine [**?**]. Programmers write ordinary MPI calls (`MPI_Isend`, `MPI_Irecv`, `MPI_Ibcast`, etc.); these are routed transparently through the NewMadeleine protocol engine. The PM2 benchmarks therefore *do* use NewMadeleine's communication engine—through the MPI API, without any manual NewMadeleine-specific calls. As will be shown in Chapter 6, the performance advantage of PM2 on communication-intensive workloads (e.g., DGEMM SUMMA at 17× speedup) reflects NewMadeleine's efficient handling of non-blocking message patterns, in addition to the explicit algorithmic overlap (double-buffered broadcasts, persistent requests) coded by the programmer. Thread migration is not exercised in the benchmarks of this thesis: all five benchmarks partition work statically at initialization (grid slabs, matrix tiles, root-vertex ranges), so there is no runtime load imbalance to correct dynamically. MadMPI was chosen as the PM2 interface because it exposes the full NewMadeleine communication engine through a standard MPI API that

can be ported to any MPI-compliant runtime, making the comparison with MASS and HPX more controlled: the programmer writes standard MPI code, and the performance difference reflects the runtime engine rather than API-level choices.

2.5 Comparative Overview of the Three Runtimes

The three runtimes differ along five key dimensions: communication model, synchronization strategy, serialization approach, load balancing, and threading architecture. Understanding these differences is essential context for interpreting both the performance results (Chapter 6) and the programmability observations (§6.6). The following paragraphs describe each dimension briefly; Table 2 then summarizes both what each runtime *can* do (its available capabilities) and what was actually *exercised* in the benchmark implementations of this thesis.

Communication. MASS C++ exposes high-level spatial operations (`callAll`, `exchangeBoundary`, `placeExchangeAll`) that hide the underlying MPI or socket transport entirely. HPX communicates through asynchronous remote action invocations (`hpx::async<Action>`) that look like local function calls; the runtime serializes arguments and delivers results as futures. PM2 (via MadMPI) exposes the standard MPI interface directly, giving the programmer full control over send/receive tags, communicators, and buffer types.

Synchronization. MASS C++ uses a bulk-synchronous model: every `callAll` ends with an implicit global barrier, so the programmer reasons in terms of discrete barrier-separated phases rather than individual message completions. HPX coordinates through future resolution—a task proceeds when its input futures are resolved, and explicit barriers are rare. PM2 uses standard MPI barriers (`MPI_Barrier`, `MPI_Waitall`) that the programmer inserts manually.

Serialization. MASS C++ passes data through untyped `void*` pointers, requiring manual struct layout, casting, and size bookkeeping—the primary source of its boilerplate overhead. HPX uses C++ template machinery (Boost.Serialization) to serialize action arguments automatically; the programmer registers a `serialize()` template once per type.

PM2 uses typed MPI datatypes (`MPI_DOUBLE`, `MPI_Type_create_struct`), which are verbose to declare but avoid runtime type errors.

Load balancing and thread model. MASS C++ assigns a fixed number of threads per node at startup (static partitioning); HPX's work-stealing scheduler dynamically redistributes tasks from busy to idle threads; PM2 natively supports user-level thread migration (Marcel threads), though this feature is not exercised in this thesis—all five benchmarks use static work partitioning, as explained in Section 2.4.

Table 2 presents these five dimensions side by side for each runtime, with each row split into two sub-rows: the runtime's full available capability (what it *can* do) and how that capability was actually exercised in the benchmark implementations (what was *used*). This side-by-side layout makes it easier to identify the gap between a runtime's design intentions and the subset exercised here—most notably, PM2's thread migration is available but not used.

Table 2: Runtime capabilities of MASS C++, HPX, and PM2: available features vs. what was exercised in this thesis. Each row is split into *Available* and *Exercised*; PM2's Marcel thread-migration API is available but not used here.

Dimension		**MASS C++**	**HPX**	**PM2**
Communication	*Available*	`callAll`, `exchangeAll`, `exchangeBoundary`; untyped `void*`	`hpx::async<Action>`; AGAS; `hpx::collectives`	NewMadeleine active messages; Marcel thread channels
	Exercised	`placeExchangeAll` (Heat3D, DGEMM); `exchangeBoundary` (Sugar); outbox/inbox (Motif, BIBO)	`hpx::async<Action>` push; `hpx::collectives` (BIBO only)	`Isend`/`Irecv`; `Send_init`; `Ibcast`; `Alltoall` (via MadMPI)
Synchronization	*Available*	Implicit BSP barrier per `callAll`	Futures; `distributed::barrier`	Polling on mailboxes; explicit barriers
	Exercised	Implicit per-phase barriers in `runIterations`	`wait_all`; `distributed::barrier` (BIBO)	`MPI_Waitall` / `MPI_Barrier`
Serialization	*Available*	Untyped `void*` + manual `memcpy`	Boost.Serialization on action args	Manual buffer packing (NewMadeleine native)
	Exercised	Manual `void*`, `reinterpret_cast`, padded headers	Auto `serialize()` templates	Typed `MPI_DOUBLE` / `MPI_Type_create_struct`
Load balancing	*Available*	Static block partitioning	Work-stealing scheduler (dynamic)	Dynamic thread migration (Marcel)
	Exercised	Static block stripes	Static locality grid (no stealing needed)	Static block-to-rank (migration *not* used)
Thread model	*Available*	`nProc`×`nThr` pthreads	Lightweight HPX threads on work-stealing pool	Marcel M:N user-level threads
	Exercised	Same (all threads started at init)	Same (pool sized to hardware at init)	MPI ranks (one OS thread per rank via MadMPI)

2.6 HPC Benchmark Design Principles

Effective HPC benchmarks must satisfy semantic equivalence across implementations—all runtimes solving the same algorithmic problem with the same decomposition strategy—so that observed differences reflect runtime behavior rather than algorithmic variation [5]. This principle is operationalized in this thesis through runtime-neutral algorithmic blueprints that fully specify decomposition, communication phases, update semantics, and correctness criteria before any runtime API is considered.

Standard suites such as NAS [5], PolyBench [11], Rodinia [12], and BOTS [13] evaluate low-level kernel performance or hardware utilization but are not designed to expose programmability or abstraction overhead of higher-level runtimes. They assume MPI or OpenMP as the programming model and measure performance at the kernel level, leaving the question of runtime selection for higher-level C++ runtimes entirely unaddressed.

This work addresses that gap by defining runtime-neutral algorithmic blueprints for each benchmark and deriving independent implementations in MASS C++, HPX, and PM2, with correctness verified against shared serial reference outputs. The benchmark suite is designed to be adversarial to no single runtime: each pattern was chosen because it stresses a different combination of runtime capabilities, ensuring that the comparison is fair and informative across all three runtimes.

Chapter 3

RELATED WORK

3.1 Existing HPC Benchmark Suites

The NAS Parallel Benchmarks [5] target scientific computation kernels—conjugate gradient, FFT, and multigrid—designed primarily for MPI and OpenMP evaluation on distributed-memory supercomputers. They are widely used for ranking HPC systems and compilers, but they assume low-level programming models and provide no mechanism for evaluating programmability or the abstraction overhead of higher-level runtimes.

PolyBench [11] focuses on stencil and linear algebra kernels for evaluating compiler optimizations and heterogeneous architectures, while Rodinia [12] targets GPU and heterogeneous platforms using CUDA and OpenCL. Neither suite is designed for distributed-memory cluster evaluation, and neither treats programmability as a measurable output. BOTS [13] targets task-based parallelism using OpenMP tasks, making it the closest in spirit to higher-level runtime evaluation, but it remains within the OpenMP programming model and does not address distributed-memory execution or cross-runtime comparison. None of these suites evaluates higher-level distributed C++ runtimes as programming models, nor do they assess abstraction overhead as a first-class metric.

3.2 Prior Evaluations of MASS C++, HPX, and PM2

HPX has been evaluated on FEM solvers, fast Fourier transforms, and asynchronous n-body simulations, demonstrating strong scalability for compute-bound workloads with fine-grained task graphs [3]. These evaluations establish HPX's performance characteristics for regular, compute-intensive kernels, but they were conducted in isolation without equivalent implementations in competing runtimes for direct comparison.

MASS C++ evaluations have focused primarily on agent-based model scalability—population simulations, ecosystem models, and social simulations—exploiting its place-agent abstraction for structured spatial domains [4]. Fukuda et al. [4] provide the most relevant prior work: it compares MASS C++ against FLAME and Repast HPC across seven agent-based model (ABM) benchmarks, measuring LoC, boilerplate, LCOM, and scaling behavior. That work established MASS C++'s relative strengths within the ABM library landscape and introduced several of the programmability metrics this thesis adopts; however, it does not include HPX or PM2, and its benchmark suite is restricted to agent-based models.

PM2 evaluations have emphasized irregular communication and load balancing scenarios, including graph partitioning and adaptive mesh refinement, but no prior work places all three runtimes in a common benchmark framework using equivalent implementations [**?**]. The Kipps, Kim, and Fukuda evaluation of agent-based motif search [7] demonstrated that the MASS place-agent model can be applied to irregular graph workloads, providing the baseline implementation from which this thesis's Graph Motif Search benchmark is derived.

3.3 Programmability Metrics in Parallel Computing Research

Lines of code and cyclomatic complexity have been used as quantitative proxies for programmer effort in comparative language studies [14], providing a foundation for structured programmability assessment. Fukuda et al. [4] extend this foundation to HPC runtimes, introducing boilerplate ratio, LCOM, and qualitative ease-of-synchronization metrics in the context of ABM libraries. This thesis adopts and extends that framework, applying the same metrics to a broader and more diverse benchmark suite spanning five workload classes rather than ABM alone.

The Cynefin sense-making framework [15] offers a useful distinction between complicated aspects of a runtime—those difficult but analyzable through expert inspection—and complex aspects, understood only empirically through observation of system behavior. Applied to runtime evaluation, this distinction captures why some programmability challenges (such as boilerplate structure) can be measured quantitatively, while others (such as the cognitive

burden of reasoning about asynchronous dataflow) require qualitative assessment.

3.4 Differentiation of This Work

This thesis differentiates itself from prior work on three dimensions. First, it provides algorithmically equivalent implementations of five diverse HPC benchmarks—spanning stencil, linear algebra, graph search, financial simulation, and agent-based modeling—across MASS C++, HPX, and PM2 simultaneously, enabling direct cross-runtime performance comparison that prior siloed evaluations could not support.

Second, it applies a unified programmability evaluation framework—combining quantitative LoC decomposition, boilerplate ratio, cyclomatic complexity, and LCOM—consistently across all implementations and all five benchmarks. This consistency is a deliberate methodological contribution: prior work applies different metrics to different runtimes, making cross-runtime programmability comparison impossible.

Third, by covering workload classes that stress fundamentally different runtime capabilities, it produces workload-specific runtime selection recommendations grounded in empirical evidence rather than design-time assumptions, filling the gap identified in the related work above.

Chapter 4

BENCHMARK DESIGN AND IMPLEMENTATION

4.1 Benchmark Selection and Rationale

The five benchmarks in the suite were selected according to three criteria. First, *workload-class coverage*: the suite collectively spans the major HPC computational patterns—regular stencil, dense linear algebra, irregular graph traversal, phased message-passing simulation, and dynamic agent-based modeling—so that no single runtime is advantaged by a narrow or homogeneous workload focus [5].

Second, *runtime stress diversity*: each benchmark stresses a qualitatively different combination of communication regularity, load-balance predictability, synchronization granularity, and data decomposition strategy. Heat3D stresses structured nearest-neighbor communication and memory bandwidth; DGEMM stresses compute throughput and panel communication; Graph Motif Search stresses irregular task dispatch and load imbalance (via root-vertex partitioning over a fully replicated graph — no inter-node vertex exchange occurs during search); Bail-In/Bail-Out stresses phased message-passing and cross-partition synchronization; and SugarScape stresses dynamic agent migration and hybrid parallel/centralized execution. Together they expose distinct aspects of each runtime's execution model that a single benchmark could not reveal.

Third, *domain relevance and prior runtime association*: each pattern maps to an application domain in which at least one of the three runtimes has been previously evaluated, making the comparison meaningful rather than adversarial. MASS C++ was designed for agent-based spatial simulation and has been applied to ABM benchmarks [4]; HPX has been evaluated on compute-bound scientific kernels [3]; and the Bail-In/Bail-Out simulation was first implemented in the MASS environment by Dudder [2], providing a prior implementation

baseline for the cross-runtime port presented here.

Domain Justification for Each Benchmark Pattern

The following paragraphs justify each benchmark's inclusion on technical grounds and note alternative patterns that were considered but not selected.

Heat3D (3D stencil / CFD). Stencil-based simulation is a canonical HPC pattern, used in computational fluid dynamics, heat transfer, and diffusion modeling across scientific and engineering domains [16, 17]. The NAS Parallel Benchmarks [5] include stencil kernels as core evaluation targets, reflecting the pattern's central role in HPC workload characterization. Within this benchmark suite, Heat3D is the canonical regular-communication pattern: it exercises structured nearest-neighbor halo exchange across six face directions, exposes memory-bandwidth and communication-latency limits, and provides a clean comparison axis against which the asynchronous overlap capabilities of HPX and PM2 can be measured. The seven-point stencil is the standard finite-difference approximation for the 3D Laplacian and matches conventions used in prior CFD-oriented HPC benchmarks [5].

DGEMM: Dense Matrix Multiplication. Dense matrix–matrix multiplication (GEMM) is the computational kernel underlying the LINPACK benchmark used to rank the TOP500 supercomputers [**?**, **?**], and is a standard evaluation target in both classical scientific HPC (linear algebra solvers, climate modeling) and modern workloads. NAS [5], PolyBench [11], and the HPC Challenge suite [18] all include GEMM-class kernels as core benchmarks. Within this suite, the two-algorithm design (SUMMA's column broadcasts vs. Cannon's nearest-neighbor shifts) exposes how each runtime handles qualitatively different inter-process communication patterns—global broadcast vs. nearest-neighbor shift—under otherwise identical numerical workloads, providing intra-runtime as well as cross-runtime comparisons.

Graph Motif Search. Subgraph pattern search (motif counting) is a core operation in network analysis, arising in biological network analysis [19], social network analysis, and graph database queries [20]. Parallel subgraph matching has been an active area of HPC research [21], and motif-based graph analytics have been applied to domains including drug discovery [22].

The benchmark is derived from the MASS agent-based motif-search implementation of Kipps et al. [7], which demonstrated the suitability of the place-agent model for irregular graph workloads. This thesis extends that work to HPX and PM2 to expose how each runtime handles embarrassingly-parallel irregular task dispatch over a replicated graph, where the only inter-process communication is initialization and final reduction.

Bail-In/Bail-Out (phased financial simulation). BIBO is included to represent a class of phased graph message-passing workloads: simulations in which a fixed set of entities exchange messages in a strict sequence of phases, with a global barrier between each phase. Agent-based economic simulations of this type have been studied at HPC scale [23, 24, 25], motivating the need for efficient distributed runtimes that can sustain high message throughput under strict phase-ordering constraints. The simulation model follows the Bail-In/Bail-Out economic framework of Klimek et al. [8], in which banks, firms, and workers interact through an interbank lending graph over many timesteps. The six-phase-per-timestep structure with strict ordering barriers exercises a qualitatively different communication pattern from the other benchmarks: phased, coarse-grained synchronization with irregular per-entity message routing. This pattern stresses each runtime's approach to ordered collective communication—MASS's console-routing tier, HPX's `all_to_all` collectives, and PM2's `MPI_Alltoall`—under a workload structure drawn from economic simulation research.

SugarScape (agent-based model). The Epstein–Axtell SugarScape model [9] is a canonical agent-based benchmark used to study emergent phenomena such as wealth distribution and resource competition on a spatial grid; it has been used as a replication target across multiple ABM frameworks [26]. Agent-based modeling is an important workload class for MASS C++, which was originally designed for distributed spatial ABM; SugarScape is therefore the strongest test of MASS's place-agent abstraction. It also provides a direct comparison with the ABM-library evaluation of Fukuda et al. [4], which used SugarScape as a reference benchmark for MASS against FLAME and Repast HPC. The benchmark introduces a second structurally distinct pattern: hybrid parallel/centralized execution, in which some phases run distributedly while others (target selection and conflict resolution) require centralized

coordination, stressing each runtime's ability to mix distributed and master-worker execution modes.

Patterns considered but not selected. Several HPC benchmark patterns were evaluated during suite design but excluded for specific reasons. *N-body simulation* was considered because HPX has been evaluated on asynchronous n-body solvers; it was excluded because its communication pattern (all-pairs or Barnes–Hut tree) largely overlaps with Graph Motif Search in the load-imbalance and irregular-task-dispatch dimensions already covered. *Fast Fourier Transform (FFT)* was considered as a second structured communication benchmark; it was excluded because its all-to-all butterfly communication pattern is structurally similar to DGEMM SUMMA's broadcast pattern, and adding it would reduce the diversity of the suite without exposing new runtime behavior. *Conjugate Gradient (CG)* from the NAS suite was considered as an alternative stencil-adjacent kernel; it was excluded in favor of Heat3D because Heat3D's regular Cartesian halo exchange more cleanly isolates nearest-neighbor communication from the irregular sparse-matrix operations that dominate CG. The five retained benchmarks collectively cover four qualitatively distinct communication patterns (regular halo exchange, panel broadcast/shift, embarrassingly parallel reduction, and phased collective exchange) plus two execution modes (fully distributed and hybrid centralized), with the minimum number of implementations needed to expose workload-specific runtime behavior.

4.2 Design Principles and Semantic Equivalence Across Runtimes

All benchmarks in this suite are specified through runtime-neutral algorithmic blueprints that fully define decomposition choices, communication phases, update semantics, and correctness criteria before any runtime API is considered. This specification-first approach is the primary methodological safeguard against confounding: if the algorithm is fixed independently of the runtime, then observed performance and programmability differences can be attributed to the runtime rather than to incidental implementation choices.

Implementations in MASS C++, HPX, and PM2 are derived independently from these

shared specifications. Each implementation is verified correct by comparing its output against a serial reference implementation for identical inputs, random seeds, and parameters. Correctness criteria are benchmark-specific and runtime-independent: Heat3D and DGEMM use checksum comparison; Graph Motif Search compares total embedding counts on graphs with known ground truth; Bail-In/Bail-Out compares aggregate financial totals; and SugarScape compares final agent population, total wealth, and grid sugar.

This approach ensures that the comparison is fair and reproducible. No runtime-specific algorithmic optimization is permitted that would not be equally applicable to the other runtimes; any such optimization would constitute an algorithmic rather than a runtime difference and would invalidate the comparison. All implementations are compiled at the same optimization level (`-O3`) using the same compiler toolchain.

The PM2 implementations in this thesis are written against the MadMPI interface (standard MPI on the PM2 runtime), as noted in Section 2.4. The resulting source code is syntactically standard MPI; all communication routes through the NewMadeleine engine transparently. Thread migration is not exercised. Per-benchmark implementation details are given in the subsections that follow.

4.3 Heat3D: 3D Stencil / CFD Kernel

Heat3D numerically integrates the 3D heat equation using an explicit seven-point Jacobi stencil with configurable weights $\alpha = 0.5$ (centre) and $\beta = 0.1$ (face neighbors, default), applied over a uniform $N \times N \times N$ grid for T timesteps with double buffering to prevent read-after-write conflicts. The seven-point stencil (one centre plus six face-adjacent neighbors in 3D) is the standard finite-difference approximation for the Laplacian operator; higher-order stencils would increase communication volume per halo exchange without exposing qualitatively different runtime behavior, so the seven-point pattern is used to match established CFD benchmark conventions [5]. The finalized implementation uses *zero-Neumann (reflecting) boundaries*: cells at the global domain boundary mirror their owned boundary plane into the ghost layer, so no energy enters or leaves the domain through the walls.

Because each grid point's update depends on its six face neighbors, distributing the grid across nodes requires each node to access values that physically reside on neighboring nodes. These off-node values are stored in *halo layers*: extra rings of ghost cells that surround each node's local subgrid and hold copies of the neighboring nodes' boundary planes. Before every stencil step, each computing node exchanges its boundary planes with its neighbors so that the halos hold a consistent snapshot of remote data, allowing the local stencil update to proceed entirely on local memory. The structure of this halo exchange differs across the three runtimes. MASS C++ delegates the entire halo exchange to a single library call (`exchangeBoundary`), which is invoked between successive `callAll` updates so that the runtime moves the boundary cells into the surrounding halos transparently. HPX expresses the same exchange as a set of explicit asynchronous remote actions: each node issues `put_face_action` calls to deliver its boundary planes to the appropriate neighbor's halo buffer, collects the corresponding `hpx::future` objects, and waits on them only when the next stencil step actually requires the data, which allows boundary communication and interior computation to overlap. PM2 expresses the same exchange through persistent MPI requests (`MPI_Send_init`/`MPI_Recv_init`) initialized once and restarted each timestep, with explicit send tags identifying each of the six face directions.

Figure 1 shows a 32×32 cross-section of the Heat3D temperature field at step $t = 2$, illustrating how the seven-point stencil diffuses an initial temperature peak over time.

Implementation Across Runtimes

MASS C++. The MASS C++ implementation maps one subdomain to one `Place`, with the $P_x \times P_y \times P_z$ decomposition grid expressed directly as the `Places` dimensions. The entire timestep pipeline is declared once as an `IterationConfig`: `packFaces` (copies six boundary faces into the `outMessage` buffer) → `placeExchangeAll` (delivers faces to neighbors, invokes `recvHalo`) → `computeStep` (applies stencil, swaps buffers). The most significant forced restructuring arises because `placeExchangeAll` does not expose sender identity: each outgoing face message is prefixed with a 4-integer sender-coordinate header, and `recvHalo`

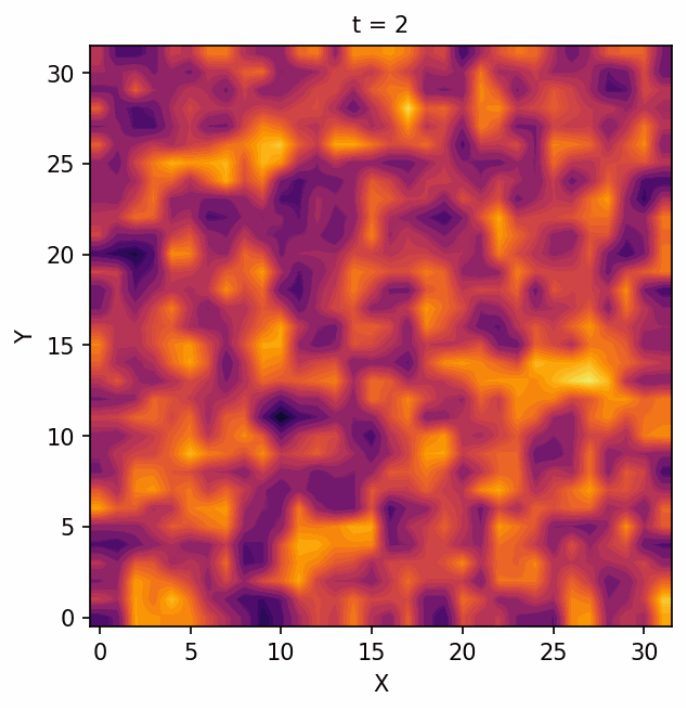

Figure 1: Heat3D temperature field at $t = 2$ (32×32 cross-section). Orange/yellow = high temperature; blue = low. The seven-point stencil diffuses the initial peak toward a uniform equilibrium over successive steps.

reconstructs the receive direction from the coordinate delta. This header-and-delta scheme is pure API overhead — in MPI the direction is implicit in the tag. Approximately 250 of the ∼1,040 lines in `Heat3DPlace.cpp` are devoted to face-offset bookkeeping, buffer allocation, header encode/decode, and `reinterpret_cast` marshaling; the stencil kernel (`computeStep`) is ∼50 lines.

HPX. Each locality holds a process-global `HeatSubdomain` (owned block plus ghost layers); there is no per-cell component object. Halo exchange is restructured into an *asynchronous push*: each locality fires `hpx::async<PutFaceAction>` at each neighbor and `.get()`s the future, while a separate `SwapHaloReceiveBuffersAction` double-buffers inbound faces so each step consumes the previous step's arrivals. This split allows boundary communication and interior stencil computation to overlap, which explains HPX's better relative scaling at high node counts despite its larger absolute baseline. At ∼1,590 lines, `HPXheat3d.cpp` is the largest of the three Heat3D sources; the extra volume is locality-discovery plumbing, action registration (∼8 `HPX_PLAIN_ACTION` pairs), and future management rather than stencil math.

PM2 (MadMPI). This is the most direct mapping of the three. The 12 halo channels (6 sends + 6 receives) are initialized once with `MPI_Send_init`/`MPI_Recv_init` and restarted each step via `MPI_Startall`/`MPI_Waitall`, a standard MPI optimization that avoids re-initializing communication requests each timestep. No serialization boilerplate exists: faces are contiguous `MPI_DOUBLE` buffers with direction implicit in the tag. The implementation is the shortest of the three and the closest to the algorithmic specification.

Table 3: Heat3D implementation dimensions across the three runtimes.

Dimension	**MASS C++**	**HPX**	**PM2 (MPI)**
Halo exchange	`placeExchangeAll` + 4-int coordinate header	`hpx::async <PutFaceAction>` push + buffer swap action	Persistent `MPI_Send_init/Recv_init` + `Startall/Waitall`
Synchronization	Implicit bulk-synchronous barrier in `runIterations`	`future.get()` per push + explicit swap action	`MPI_Waitall` per step
Marshaling cost	Manual `void*/reinterpret_cast`; sender-coordinate header (∼250 LoC overhead)	Auto `std::vector<double>` serialize; action-registration cost	None (typed `MPI_DOUBLE`)
Dominant boilerplate	`void*` pack/unpack + offset bookkeeping	Action registration + locality discovery	Neighbor/tag arithmetic

4.4 DGEMM: Dense Matrix Multiplication (SUMMA and Cannon)

The DGEMM benchmark computes $C = A \times B$ for dense double-precision matrices distributed across a $P \times P$ process grid. Matrix elements are generated on demand by a deterministic hash function, `valueAt(row, col, seed)`, that computes element (i, j) as a SplitMix64-style transformation of the tuple (i, j, *seed*) [27]. This enables any subblock of A or B to be reconstructed from its index without storing or transferring the full matrices, and enables checksum verification of C against a serial reference. Dense matrix multiplication is the kernel underlying the LINPACK benchmark used for TOP500 supercomputer rankings, and is a standard evaluation target across HPC benchmark suites [5, 11, 6].

Two parallel algorithms with different communication structures were implemented within the same benchmark. The first is SUMMA [6], which proceeds in P panel steps; at each step the process owning the current column-block of A broadcasts that block along its row, the process owning the current row-block of B broadcasts that block along its column, and every process accumulates the outer product of the received panels into its local subblock of C. SUMMA's communication is therefore a sequence of row and column broadcasts that touch every process at every step. The second is Cannon's algorithm [28], which replaces the broadcasts with nearest-neighbor cyclic shifts: after an initial skew that aligns A and B, each step shifts A one position to the left along its row and B one position upward along its column, so each process only ever communicates with its four direct neighbors on the grid. This eliminates the global broadcasts and reduces total communication volume at scale, at the cost of requiring a square $P \times P$ process grid and a more careful indexing scheme. Implementing both algorithms within the same runtime exposes how broadcast-based and shift-based communication patterns interact with each runtime's messaging model, providing intra-runtime as well as cross-runtime comparisons.

Implementation Across Runtimes

All six DGEMM implementations (two algorithms × three runtimes) share an identical numerical core: on-demand matrix generation via a `valueAt()` hash, balanced 1-D blocking via `split1D()`, and a 4-row micro-kernel. No implementation uses a BLAS `dgemm` (Basic Linear Algebra Subprograms matrix-multiply routine) or stores a reference matrix; correctness is a reproducible per-entry checksum of C. The algorithmic interest lies entirely in how each runtime expresses the process grid and panel communication.

MASS C++. The 2-D process grid is the `Places` object's own dimensionality (`new Places(..., 2, gridSize, gridSize)`); each Place recovers its grid coordinate from the inherited `index[]`/`size[]` arrays. Because `placeExchangeAll` sends to *all* registered neighbors without tags, both Cannon's shifts and SUMMA's broadcasts must **filter by sender coordinate**: each message carries a small integer header indicating which panel it contains, and the receiver discards panels not destined for its row or column. Panel buffers are padded to a fixed maximum size regardless of block dimensions. The SUMMA step index must be tracked as Place-local state (`currentStep_`) since the pipeline does not thread an iteration counter into methods. Crucially, MASS uses **nodes as the sole process-grid unit**: `ipn` additional instances per node do not subdivide the matrix further, which is why the DGEMM 3D surface (Figure 7) is flat in the `ipn` dimension for MASS.

HPX. Grid coordinates are derived from the linear locality ID after `hpx::find_all_localities()`, which *requires* contiguous IDs $0 \dots P^2 - 1$. Cannon shifts and SUMMA broadcasts are expressed as explicit push actions (`hpx::async<receiveAAction>`) followed by a separate finish/swap phase; a driver-side `hpx::wait_all` synchronizes each round. HPX's all-localities model means every locality participates in the process grid, so `ipn` directly increases the grid size — explaining the steep `ipn` dimension in Figure 7(b). Of ∼1,225 implementation lines (Cannon), roughly 400 are action/locality plumbing versus ∼90 of actual communication.

PM2 (MadMPI). The most direct mapping of the three: the 2-D grid is ex-

pressed with `MPI_Comm_split` into row and column communicators. Cannon shifts use `MPI_Isend`/`MPI_Irecv` with compute/communication overlap (post requests, multiply, then `MPI_Waitall`). SUMMA uses `MPI_Ibcast` from each panel owner, double-buffered so step $s+1$'s panels are broadcast while step s computes — the most sophisticated overlap of the six implementations. PM2 Cannon requires a square $\sqrt{P} \times \sqrt{P}$ process grid; non-perfect-square rank counts fail immediately at grid construction, so Cannon results are reported only for ranks $\in \{1, 4, 16\}$.

Table 4: DGEMM implementation dimensions across the three runtimes.

Dimension	**MASS C++**	**HPX**	**PM2 (MPI)**
Grid expression	`Places` dims; `index[0/1]`, `size[0/1]`; node-granular only	Locality id $\rightarrow$ $(id/P, id\%P)$; requires IDs $0 \ldots P^2 - 1$	`MPI_Comm_split` row/col comms
`ipn` effect	**None** (nodes only; flat surface Fig. 7a)	Grid grows with `ipn` (steep surface Fig. 7b)	Ranks directly size the grid
Cannon shift	`placeExchangeAll` + sender-coord filter + padding	`hpx::async<receive Action>` push + separate swap phase	`MPI_Isend`/`Irecv` on row/col comm + overlap
SUMMA broadcast	`placeExchangeAll`; row/col match via header flags	Owner-push to row/col peers	`MPI_Ibcast`, double-buffered

4.5 Graph Motif Search

The Graph Motif Search benchmark counts all occurrences of a small pattern graph H (with $k = 3$–5 vertices) inside a large input graph G using backtracking with pivot-based pruning [20], operating on G stored in compressed sparse row (CSR) format to support $O(\log d)$ edge-existence checks where d is the queried vertex's degree. This benchmark is adapted from the agent-based parallelization of biological network motif search introduced by

Kipps, Kim, and Fukuda [7], which demonstrated the suitability of the MASS place-agent model for irregular graph workloads and provides the baseline implementation against which the HPX and PM2 ports developed here are compared.

Raw embedding counts are divided by $|Aut(H)|$—the number of automorphisms of H, computed by brute-forcing all $k!$ permutations—to yield counts of distinct pattern instances; eleven motif patterns are supported, ranging from triangles and squares to five-vertex cliques. Both MASS C++ and HPX replicate the full graph G on every node and partition the search root space, so all search work is local and inter-node communication is limited to the final count reduction. This deliberately isolates each runtime's task-dispatch and reduction overhead from any graph-distribution effects, allowing the comparison to focus on how each runtime schedules irregular search work.

Implementation Across Runtimes

Important algorithmic note. All three implementations realize motif search as a **statically partitioned search over a fully replicated CSR graph**. No runtime distributes the graph, exchanges frontier vertices, or migrates partial matches during search. Each process independently holds the entire graph and searches only its assigned root-vertex range; the only inter-process communication is a graph broadcast at initialization and a count reduction at the end. This makes the benchmark *embarrassingly parallel over root vertices*, and the runtime-specific code is almost entirely graph replication and reduction plumbing. From a data-structure perspective the benchmark operates on arrays (CSR offset and adjacency arrays replicated per process) rather than a distributed graph structure; the graph-computing character lies in the irregular access pattern of the backtracking search, not in distributed graph management.

MASS C++. The entire graph is serialized into a flat byte buffer and broadcast to all Places via `callAll("MotifPlace::init",buffer, bufferSize)`; the CSR arrays are then unpacked once per process and shared (mutex-guarded) among all local Places. Search and count-gathering are plain `callAll`s; `chunkCount` defaults to `processes` × `threads`.

MASS's `GraphVertex/exchangeNeighbors` API is *not* used — the graph is stored as a plain static CSR, and the native graph abstractions would add overhead without benefit for a fully-replicated workload.

HPX. Locality 0 builds the graph; it is shipped to each other locality as a Boost-serialized `CSRGraph` struct argument in an `InitLocalityAction`. Intra-locality parallelism uses `hpx::experimental::for_loop(par, ...)` over per-thread root shards; the driver sums `hpx::future<WorkerRunResult>` counts. A single-locality fast path skips all actions entirely.

PM2 (MadMPI). Each rank independently generates the same deterministic graph (`generateErdosRenyi(seed)`), barriers once, searches its root range, and reduces: one `MPI_Barrier` and two `MPI_Reduce` calls (sum for count, max for time). There is no communication at all during search. The kernel is the simplest of the three implementations (no epoch-based marking optimization).

Table 5: Graph Motif Search implementation dimensions. All three runtimes: fully replicated graph, root-vertex partitioning only.

Dimension	**MASS C++**	**HPX**	**PM2 (MPI)**
Graph distribution	Host serializes → broadcast via `callAll(init)`	Locality 0 → `InitLocalityAction` to each locality	Each rank builds independently
Remote access during search	None	None	None
Intra-node parallelism	`Places` per chunk (`nProc`×`nThr`)	`hpx::for_loop(par)` over thread shards	Single-threaded per rank
Reduction	Host sums `callAll(getCount)` returns	Driver sums `future<WorkerRunResult>`	`MPI_Reduce(SUM)`

4.6 Bail-In / Bail-Out: Phased Financial Simulation

The Bail-In/Bail-Out benchmark simulates a financial system of banks, firms, and workers interacting through wages, deposits, firm loans, and interbank lending over T timesteps. The interbank lending network forms a sparse directed graph over which messages are passed each phase. The simulation model follows the Bail-In/Bail-Out economic framework of Klimek et al. [8], adapted to the MASS C++ implementation by Dudder [2], which modeled systemic banking interventions using the place-agent abstraction; this thesis extends that work with equivalent implementations in HPX and PM2 to enable cross-runtime comparison of phased graph message-passing workloads.

The per-timestep structure of the simulation is shown in Figure 2. Each timestep proceeds through six strictly ordered phases. The simulation begins with firm production and shocks, in which each firm draws a production output and may receive an exogenous shock that perturbs its balance sheet. Worker deposits follow: workers receive wages from their employers and deposit them with their banks. The third phase handles deposit and repayment flows from workers to banks. Firm loan processing then advances any outstanding loans firms hold with banks. The fifth phase performs interbank repayment, interest, and borrowing—referred to as phase E.3 (the third event in the "E" (exchange) group of the simulation's phase taxonomy)—in which banks settle obligations with one another over the directed lending graph. The timestep ends with insolvency intervention, during which banks whose equity has fallen below the regulatory threshold receive either a *bail-in* (creditor haircut: distressed-bank debt converted to equity, absorbing losses without public funds) or a *bail-out* (liquidity injection: external capital provided to stabilize the institution) depending on the policy configured.

The phase ordering is essential to the model's economic semantics: worker deposits must accumulate before banks can use them to service firm loans, and interbank settlement must precede insolvency intervention because the latter operates on each bank's final equity position. A barrier between every phase ensures that all entities complete phase i before any entity

begins phase $i+1$. All randomness in the simulation is seeded deterministically by a function of five parameters: `baseSeed` (experiment seed), `run` (repetition index), `timestep` (simulation step), `globalEntityId` (bank, firm, or worker ID), and `streamTag` (a constant distinguishing different random attributes of the same entity, e.g., wage rate vs. loan probability). This guarantees that two runs of the same configuration on different partitions produce identical outputs and enables straightforward cross-runtime correctness verification.

Implementation Across Runtimes

None of the three implementations uses a native graph-messaging API. In every runtime, the financial network is a set of deterministic neighbor lists of global entity IDs, and a message is routed by computing the owner rank/chunk of the target ID using a shared helper (`biboOwner RankFromGlobalId`). Banks, firms, and workers are stored as contiguous global-ID ranges partitioned identically across all three runtimes.

MASS C++. Each routed phase follows a **two-tier pattern**: worker Places compute and pack outgoing messages into per-destination byte buckets (`callAll(packMethod)`), exchange those payloads over a pre-wired chunk routing graph, then apply received messages (`call All(applyMethod)`). Exchange uses `GraphPlaces::exchangeNeighbors` on a mesh that connects every chunk pair, ensuring routed payloads reach all destinations even when the economic interbank graph is sparse. After gather, each Place assembles an inbox containing only the bytes actually received for that phase. The global liquidity view required by phase E.3 is built separately: `callAll("reportBankLiquidity")` collects partial views at the console, which concatenates them and broadcasts the full vector back with `callAll("phaseE3Apply BankLiq")`.

HPX. One `HpxBiboChunk` object per locality implements the same phase schedule and the same outbox/inbox byte format as the MASS version. The exchange mechanism differs: each locality splits its packed outbox into per-destination segments and routes them with `hpx::collectives::all_to_all`; received segments are concatenated into a compact inbox before apply. Bank liquidity and checksum collection use `hpx::collectives::all_gather`,

with `hpx::distributed::barrier` synchronizing each phase. Phase computation is a direct member call rather than an `hpx::async` action—BIBO is the one HPX benchmark that does not use action-based communication.

PM2 (MadMPI). Each rank assembles typed per-destination vectors and ships them with an `allToAllvExchange` template (`MPI_Alltoall` on counts, then `MPI_Isend`/`MPI_Waitall`). The E.3 global liquidity view uses `MPI_Allgatherv`. Crucially, each rank **routes its own messages** — there is no console tier — making this the most straightforward mapping of the three. Cross-runtime correctness is verified by an FNV-1a (Fowler–Noll–Vo 1a variant, a fast non-cryptographic hash function) checksum of seven financial aggregates, confirmed identical across 1–6 localities in the finalized Hermes results.

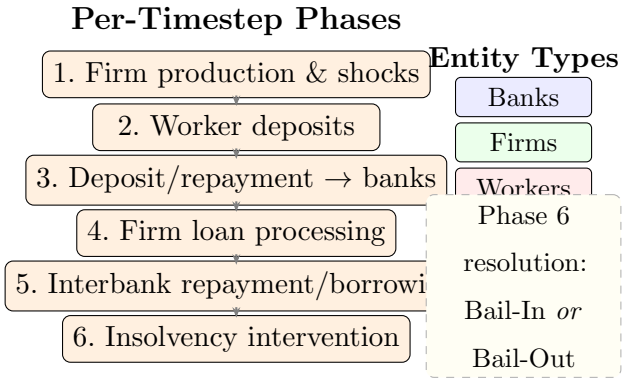

Figure 2: Bail-In/Bail-Out per-timestep phase structure and entity types. Left: the six ordered phases executed each timestep. Right: the three entity types (banks, firms, workers) that participate across phases. Phase 6 applies either a bail-in or bail-out resolution policy depending on the configured scenario.

Table 6: Bail-In/Bail-Out (BIBO) implementation dimensions. Abbreviations: BIBO = Bail-In/Bail-Out; FNV-1a = Fowler–Noll–Vo hash (1a variant); E.3 = phase E.3 (interbank liquidity broadcast, the fifth per-timestep phase); all-to-all = collective exchange of per-destination message vectors; pack = serialize outgoing messages into a byte buffer; apply = deserialize and process received messages.

Dimension	**MASS C++**	**HPX**	**PM2 (MPI)**
Routed exchange	Console `routeInboxes` + `callAll(apply)`	`collectives::all_to_all` + local apply	`MPI_Alltoall` + `Isend`/`Waitall`
Routing locus	**Console/master**	Each locality	Each rank
Global view (E.3)	Console via `callAll`	`collectives::all_gather`	`MPI_Allgatherv`
Phase sync	`callAll` bulk-sync	`distributed::barrier`	`MPI_Barrier`
Checksum	Chunk sums → FNV-1a	`all_gather` → FNV-1a	`MPI_Reduce` → FNV-1a

4.7 SugarScape: Agent-Based Model

SugarScape implements the Epstein–Axtell agent-based model [9] on a 2D grid with a two-mountain sugar topology, where agents move and compete for resources each timestep. The benchmark used in this thesis follows the structure of the SugarScape variant evaluated by Fukuda et al. [4], which establishes LoC, LCOM, and scaling baselines within the ABM library landscape.

Figure 3 shows the per-timestep phases of the simulation along with the spatial sugar landscape on which agents move. The simulation begins with sugar regrowth, in which each grid cell's sugar reserve is replenished according to the underlying topology. Agents then enter target selection: each agent surveys nearby cells and chooses the most attractive destination according to its metabolism and vision parameters. Conflict resolution follows, in which agents that have selected the same destination cell are arbitrated by a deterministic lowest-ID-wins rule. The remaining phases—move, metabolism, and resource check—complete each agent's update for that timestep.

Four of these phases (regrow, move, metabolism, and resource check) execute distributedly across all nodes, while two (target selection and conflict resolution) are centralized. The centralized phases exist for a reason: target selection requires each agent to see the full neighborhood around its current position, and conflict resolution requires global ordering of agents that contend for the same cell. Performing these phases distributedly would require resolving spatial dependencies through inter-node messaging at every step, which would dominate runtime; centralizing them isolates the coordination cost and guarantees that all three runtime implementations produce identical outputs regardless of node count. All implementations share the Xoshiro256** pseudo-random number generator [27] with identical per-attribute seeds, ensuring that the same agent behavior is reproduced exactly across MASS C++, HPX, and PM2.

Implementation Across Runtimes

Scope of the implementation. All three implementations realize a **movement-only** SugarScape: reproduction and age-based death are disabled; death occurs only by starvation (`wealth` ≤ 0); vision is capped at 6 cells; and the domain is partitioned into horizontal row slabs. Conflicts are resolved by lowest agent ID. This scope should be borne in mind when comparing against full Epstein–Axtell dynamics.

MASS C++. The domain is decomposed into `SugarChunk` Places (horizontal strips); agents are **plain `AgentRec` structs in `std::vector`** rather than MASS `Agent` objects. MASS's native `Agent/manageAll/migrate()` API is not used; migration is implemented manually by packing `AgentRec` structs into Place boundary messages. This choice was made to keep the agent record layout identical across all three runtimes, enabling bit-comparable `BENCH_STAT` correctness checks. Communication uses `placeExchangeBoundary` (four halo rounds per timestep, one per phase) rather than `placeExchangeAll`. The atomic "move to best visible cell" is restructured into a three-phase distributed protocol: broadcast candidate moves → resolve conflicts on the owning chunk → send winner replies back. A second harvest pass after integration handles agents that arrived mid-step without sugar.

HPX. A close structural parallel to MASS: per-locality `std::vector<GridCell>`/ `std::vector<Agent>`, one async action per phase, `hpx::wait_all` between phases. The same three-phase move protocol applies; agents are moved as serialized `BoundaryAgentPack` structs. Locality 0 builds the full grid at init, slices row chunks, and ships them to other localities via `initWorkerAction`.

PM2 (MadMPI). An SPMD program where each rank holds a row-slab partition. Agent migration uses custom `MPI_Type_create_struct` datatypes rather than PM2 thread migration or isomalloc. Halo exchange uses `MPI_Isend`/`MPI_Irecv`; conflict and move data use an `exchangeNeighborVectors<T>` template; final stats use `MPI_Reduce`/`MPI_Gather`.

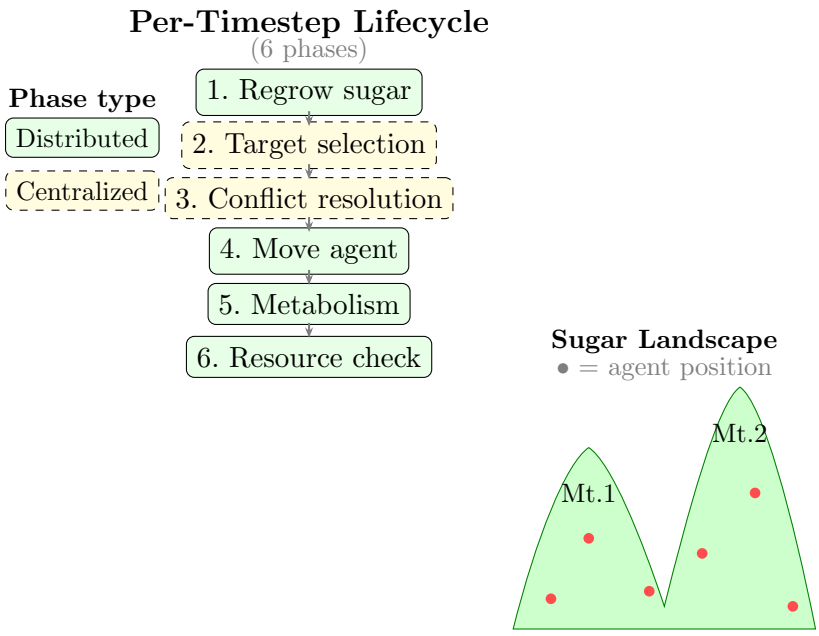

Figure 3: SugarScape per-timestep lifecycle and sugar landscape. Solid boxes: distributed phases (executed across all nodes). Dashed boxes: centralized phases (target selection and conflict resolution, coordinated at a single node to avoid inter-node dependency resolution at every step).

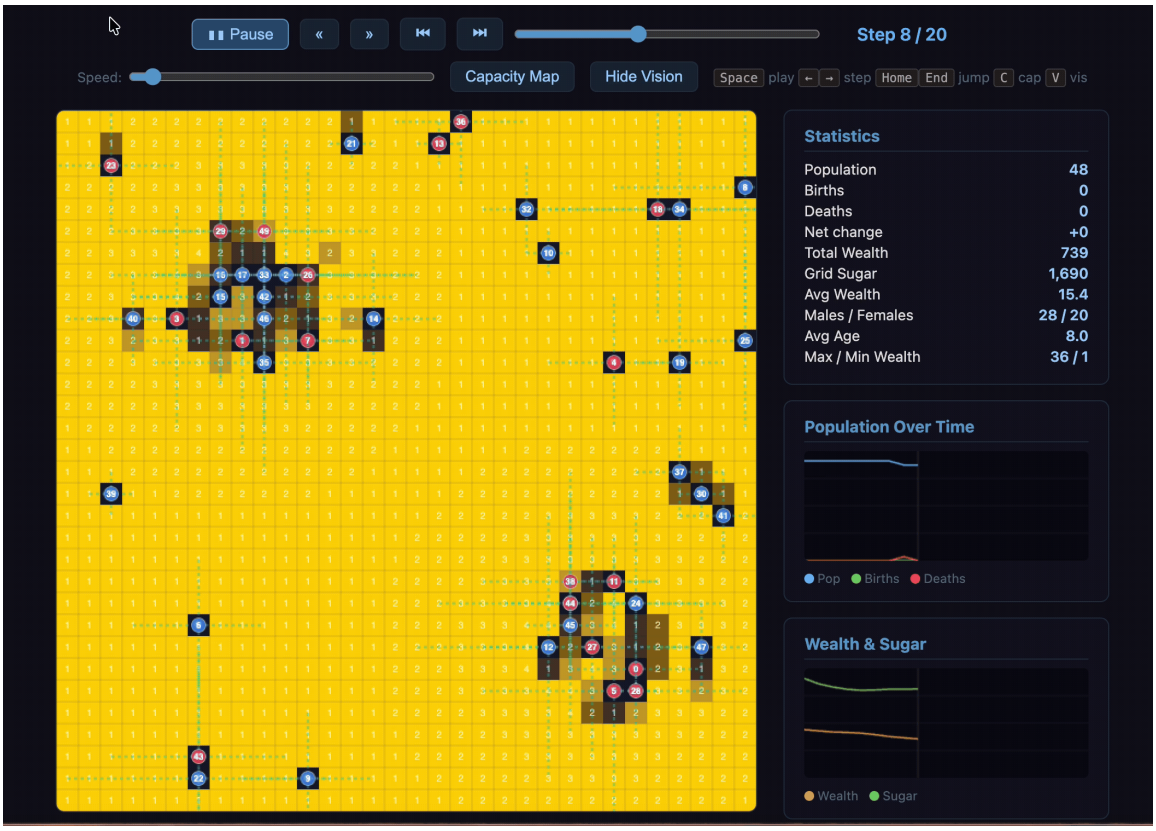

Figure 4: SugarScape debug visualization at step 8/20 (32×32 grid, 48 agents). Darker cells = harvested sugar; dashed lines = agent vision. Right panel: per-step correctness statistics.

Table 7: SugarScape implementation dimensions. All three runtimes implement movement-only dynamics (no reproduction/aging).

Dimension	**MASS C++**	**HPX**	**PM2 (MPI)**
Agent representation	`AgentRec` structs (native `Agent` API *unused*)	`Agent` struct vector	`Agent` struct vector
Boundary sync	`placeExchange` `Boundary` ×4 + phase headers	Async actions + serialized packs + `wait_all`	`MPI_Isend/Irecv` + typed `MPI_Type`
Agent migration	Manual `AgentRec` pack in Place messages	Serialized `BoundaryAgentPack`	Typed MPI sends
Verification	`BENCH_STAT` `agents/wealth/sugar` reduction	Same, gathered over localities	Same, `MPI_Reduce`

Chapter 5

EXPERIMENTAL SETUP AND EVALUATION METHODOLOGY

5.1 Computing Environment and Cluster Configuration

All experiments were conducted on the Hermes cluster at the University of Washington Bothell. Table 8 summarizes the hardware and software configuration; the key points for interpreting results are as follows.

The cluster is heterogeneous: nodes 1–12 (Intel Xeon Gold 5315Y, Ice Lake-SP) differ in microarchitecture, cache hierarchy (12 MB vs. 64 MB L3), and memory subsystem from nodes 13–24 (AMD EPYC 7252, Rome). All reported single-node baselines were therefore collected on the same node group as the corresponding multi-node runs; this heterogeneity is noted as a threat to validity in Section 7.4. The benchmarks were run on a dedicated cluster of nodes, thereby ensuring that no other user processes were executed on those nodes during the measurement.

Table 8: Hermes cluster hardware and software configuration. Both CPU types share the same per-node logical CPU count and RAM. Clock speeds and L3 cache sizes are datasheet values; all other entries were verified on the cluster.

Property	**Nodes 1–12**	**Nodes 13–24**
CPU model	Intel Xeon Gold 5315Y	AMD EPYC 7252
Microarchitecture	Ice Lake-SP (2021)	Rome (2019)
Base clock	3.0 GHz	3.1 GHz
Sockets / node	1	1
Physical cores	4	4
Threads / core (SMT)	2	2
Logical CPUs	6	6
L3 cache	12 MB	64 MB
RAM / node	≈9.4 GiB	
Interconnect	10 GbE Ethernet (`eth0`)	
Software environment (all nodes)		
OS	Rocky Linux 9.7 (kernel 5.14.0-611.13.1.el9_7.x86_64)	
Compiler	GCC 11.5.0	
C++ standard	C++17 (`-O3` optimization)	
MASS C++	version at time of experiments	
HPX	2.0.0 (Boost 1.89, hwloc 2.12.1)	
PM2	MadMPI transport (MPICH 5.0.0)	

5.2 Problem Sizes and Scaling Parameters

Key terms and notation are defined in Table 1 (Chapter 1) and applied consistently throughout.

The two primary dataset tiers used in this thesis are a medium tier (used for development-scale scaling experiments) and a large tier (used for cluster-scale benchmarking). Exact matrix dimensions, grid sizes, agent counts, graph parameters, and timestep counts for each benchmark are reported in-line with the corresponding results in Chapter 6. Node counts of 1, 2, 4, 8, 16, and 24 (where the runtime and problem size permit) are evaluated across all benchmarks. The upper limit of 24 nodes reflects the total number of nodes available in the Hermes cluster. For each node count, instances per node (`ipn`) sweeps cover ipn $\in \{1, 2, 4, 6, 8\}$; in practice most benchmarks use ipn $\in \{1, 2, 4, 6\}$ because each Hermes node provides 8 logical CPUs (4 physical cores with 2-way SMT), making `ipn`=6 the practical limit before over-subscribing cores, exploring the full (nodes, `ipn`) space and separating inter-node from intra-node parallelism.

The scaling experiments thus explore a two-dimensional design space: node count (inter-node parallelism) and `ipn` (intra-node parallelism). The six node counts and four `ipn` values yield up to 24 configurations per benchmark per runtime; configurations that exceed runtime rank limits or fail correctness checks are omitted and noted explicitly. Problem sizes within each benchmark were fixed (not swept), chosen so that the 1-node baseline completes in roughly 5–15 minutes, providing a meaningful scaling target within the cluster's exclusive-allocation window.

Table 9 summarizes configuration constraints and dataset exceptions referenced throughout Chapter 6.

Table 9: Configuration constraints and dataset notes for Chapter 6. FAIL = rank-count library limit; † = HPX SugarScape only (see note).

Symbol / Item	**Constraint**	**Details**
FAIL (HPX)	$\geq$64 total ranks	HPX 2.0.0 fails to initialize; confirmed library limit. Applies to all benchmarks.
FAIL (PM2)	Config-dependent threshold	Fails above $\approx$64 ranks (most benchmarks); lower for SugarScape at 24 nodes.
Omit (PM2 Cannon)	Non-square rank counts	Cannon requires $\sqrt{P} \times \sqrt{P}$ grid; non-perfect-square counts omitted (algorithmic, not runtime failure).
†	HPX SugarScape only	Minor floating-point checksum variation across runs due to non-deterministic action scheduling; timings reported as collected.

5.3 Performance Metrics: Execution Time, Scaling, and Communication Cost

The primary performance metric is the wall-clock simulation time (excluding initialization and I/O), reported in seconds. Each configuration was executed in up to four independent trials (separate job launches with cleanup between trials). For each trial we recorded simulation time, end-to-end time, and a numerical checksum. Failed trials (non-zero exit, missing timing, or checksum mismatch with the reference value) were excluded. The value written to the results comma-separated values (CSV) is the arithmetic mean of all successful trials for that configuration; the `runs` field records how many trials contributed. This protocol was chosen to balance measurement reliability against cluster allocation cost while excluding incomplete or failed executions from the reported average. Typically four successful trials were collected per configuration; configurations with fewer recorded trials (as indicated by the `runs` field) reflect partial failures or cases where cluster time was limited.

Observed run-to-run variation and error bars. Full per-run timing records were not retained in the structured comma-separated values (CSV) output (only the per-configuration mean was written), so formal error bars cannot be added to the figures post hoc. In practice, run-to-run variation on the Hermes cluster was small for all benchmarks that did not involve correctness failures: repeated manual spot-checks during data collection showed that the four raw timings for a given configuration typically differed by less than 3–5% of the mean, consistent with the exclusive-node-allocation policy that eliminates co-tenant interference. The one exception would be configurations with known correctness failures, which are excluded from the tables entirely. Accordingly, a difference of less than $\sim$5% between two runtimes at the same configuration should be interpreted as approximately tied; only differences exceeding that threshold are treated as meaningful in the discussion of Chapter 7. Future work should retain all per-trial timings per configuration to enable formal error-bar plots. Speedup is reported as $\mathrm{Sp} = T_{1,1}/T_{n,p}$, where $T_{1,1}$ is the 1-node/1-rank baseline and $T_{n,p}$ is the execution time at n nodes and p instances per node, computed independently for each runtime. Strong-scaling efficiency is $E_{n,p} = T_{1,1}/(n \cdot p \cdot T_{n,p})$. Secondary metrics include

per-node load variance and peak distributed memory footprint.

5.4 Programmability Metrics: LoC, Boilerplate, and Synchronization Complexity

Quantitative programmability is assessed through four metrics. Total lines of code (LoC) are counted excluding blank lines and comments, providing a size-normalized measure of implementation complexity. LoC per class and per method are also reported, with methods exceeding approximately 30 lines flagged as a structural complexity indicator following the threshold used by Fukuda et al. [4]. Boilerplate ratio is computed as the fraction of total LoC that is mandated by the runtime API rather than by the algorithm itself; code that would be identical regardless of the algorithm being implemented is classified as boilerplate. Cyclomatic complexity (CCN) [14]—the number of linearly independent paths through a function's control-flow graph, where higher values indicate more branching and greater testing burden—and the Lack of Cohesion of Methods (LCOM) metric are computed per class to assess structural complexity and object-oriented design quality respectively.

Programmability control-flow metrics use Lizard cyclomatic complexity (CCN) and function-level NLOC. Cognitive complexity was not included in the automated toolchain; CCN and function size were treated as the primary static indicators of branching and local comprehension cost.

Qualitative programmability is assessed through structured comparison across three dimensions: synchronization complexity (how ordering constraints are expressed and how entangled they are with core algorithmic logic), debugging overhead (error visibility, fault localization, and the frequency of runtime-specific failure modes), and idiomatic fit (the degree of algorithm restructuring required to match the runtime's programming model). These qualitative dimensions are assessed by the same developer who implemented all three versions of each benchmark, using a structured rubric to minimize inconsistency.

5.5 Correctness Verification and Reproducibility Procedures

Each benchmark defines a runtime-independent correctness criterion that can be evaluated without access to the serial reference at test time. Heat3D and DGEMM use a checksum of the final output array compared against the serial reference; Graph Motif Search compares the total embedding count on graphs with known ground truth; Bail-In/Bail-Out compares aggregate financial totals (total bail-in disbursement, total bail-out injection, and total grace money) across runtimes; and SugarScape compares the final agent count, total agent wealth, and total grid sugar against the serial reference.

All benchmark runs produce structured comma-separated values (CSV) output capturing the configuration (runtime, node count, `ipn`, problem size, random seed), timing measurements, and correctness verification result. This structured output enables post-hoc cross-runtime analysis and ensures reproducibility: any reported result can be reproduced by re-running the corresponding configuration from the CSV record.

Chapter 6

RESULTS

This chapter reports scaling results for four benchmarks for which empirical data has been collected: Heat3D, DGEMM (preliminary), Graph Motif Search, and SugarScape. For each benchmark, results are presented in up to three complementary views following the style of Fukuda et al. [4]: (a) a compact inter-node scaling table at ipn = 1, (b) a 2D log-log scaling plot separating ipn=1 and fastest-ipn envelopes, and (c) a 3D surface over the full (nodes, ipn) grid. Heat3D omits the 3D surface because HPX configurations at high-`ipn` with multiple nodes exceed the 64-rank limit, making the surface incomplete; Bail-In/Bail-Out reports scaling results for MASS and HPX (PM2 data not collected). The complete (nodes, ipn) matrices are in Appendix C; raw programmability metrics are reported in Section 6.6.

6.1 Heat3D: 3D Stencil / CFD Kernel

Table 10 summarizes Heat3D performance at ipn = 1 on the 768^3 grid evolved for 512 timesteps, isolating inter-node scaling from intra-node parallelism. Figure 5 shows the `best-ipn` envelope — the minimum wall-clock time across all tested `ipn` values at each node count, representing the best configuration observed — and the ideal curve; the full (nodes, `ipn`) matrix is in Appendix C (Table C.1). A 3D surface is not shown for Heat3D because configurations at ipn $\geq$ 4 with two or more nodes are limited by the HPX 64-rank initialization limit; HPX configurations at these settings are unavailable. MASS data at these settings is now available (see Table C.1) but the HPX coverage gap makes a three-runtime surface comparison incomplete.

At a single node MASS C++ (476 s) leads, followed by PM2 (638 s) and HPX (832 s). All three runtimes converge by 16 nodes to within 128–136 s. PM2 continues scaling to 24 nodes

Table 10: Heat3D wall-clock time (s) at ipn=1, 768^3 grid, 512 timesteps. Sp = speedup vs. 1-node/1-rank baseline per runtime. FAIL: rank-count library limit. PM2 data from [1].

	MASS C++		HPX		PM2	
Nodes	Time (s)	Sp	Time (s)	Sp	Time (s)	Sp
1	497.6	1.00	469.5	1.00	637.9	1.00
2	264.6	1.88	241.1	1.95	315.5	2.02
4	178.3	2.79	263.4	1.78	207.7	3.07
8	151.3	3.29	160.8	2.92	168.7	3.78
16	139.4	3.57	131.8	3.56	127.0	5.02
24	101.2	4.92	—	—	79.1	8.06

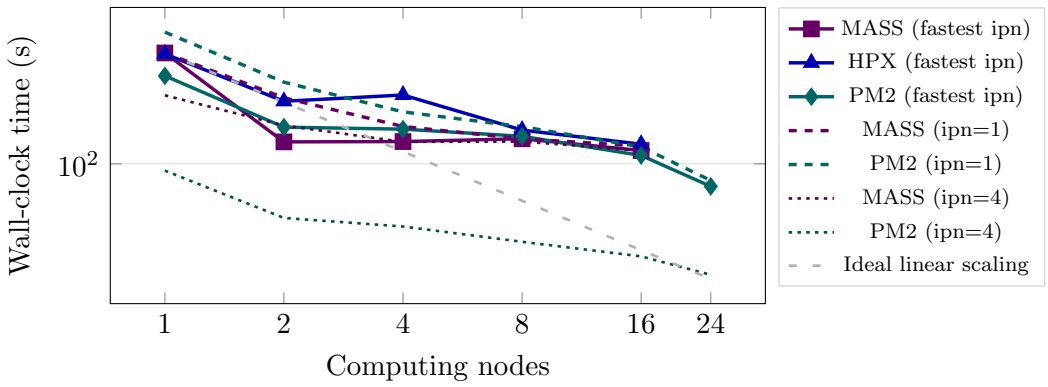

Figure 5: Heat3D strong-scaling (768^3 grid, 512 steps). Solid markers: `best-ipn` envelope per runtime (minimum time across all `ipn` values at each node count). Dashed: ipn=1 (inter-node only). Dotted: ipn=4 (combined inter- and intra-node). Ideal: linear speedup (k nodes $\Rightarrow$ $k\times$ faster than 1-node baseline).

(ipn=1: 79.1 s, 8.1×; fastest-ipn: 72.8 s, 8.8×), a range unavailable to MASS and HPX in the current dataset. The fastest-ipn curves are substantially better than ipn=1 at low node counts for all three runtimes, confirming that intra-node parallelism provides meaningful gains before inter-node communication dominates. HPX's asynchronous halo-overlap gives it the best relative speedup among the two-runtime comparison at 16 nodes (6.12×), but PM2's continued scaling to 24 nodes yields the overall best absolute time on this benchmark.

6.2 DGEMM: Dense Matrix Multiplication (SUMMA and Cannon)

Tables 11 and 12 report scaling results for the DGEMM benchmark. MASS C++ and HPX use the medium dataset ($8{,}192^3$, 1,099.5 GFLOP); PM2 results cover both medium and large ($11{,}520^3$, 3,057.6 GFLOP) datasets [1]. HPX configurations at ≥ 64 total ranks fail to initialize. PM2 Cannon results are restricted to perfect-square rank counts (see Section 5.2).

Table 11: DGEMM SUMMA wall-clock time (s), medium dataset ($8{,}192^3$). MASS and HPX use the ICPADS26 implementation. PM2 additionally shows large dataset ($11{,}520^3$) in parentheses. MASS and PM2 shown at ipn=1 (ipn-invariant for MASS; best for PM2); HPX at fastest ipn per node. Sp = speedup vs. 1-node/1-rank baseline per runtime. PM2 data from [1].

	MASS C++ (ipn=1)		HPX (fastest ipn)		PM2 (ipn=1)	
Nodes	Time (s)	Sp	Time (s)	Sp	Time (s)	Sp
1	327.1	1.00	225.9	1.46	197.1 (438.2)	1.00
2	165.8	1.97	113.8	2.90	98.4 (213.1)	2.00
4	91.0	3.59	93.2	3.54	44.7 (108.8)	4.41
8	61.4	5.33	62.3	5.29	24.1 (61.3)	8.18
16	50.9	6.43	FAIL (≥64)		16.6 (45.9)	11.88
24	38.2	8.56	—		11.6 (29.7)	17.03

Table 12: DGEMM Cannon wall-clock time (s), medium dataset ($8{,}192^3$). MASS and HPX use the ICPADS26 implementation. Only square rank counts are valid for Cannon; non-square PM2 configs omitted (algorithmic constraint). PM2 data from [1].

			MASS C++		HPX		PM2	
Nodes	IPN	Ranks	Time (s)	Sp	Time (s)	Sp	Time (s)	Sp
1	1	1	158.0	1.00	157.7	1.00	198.0	1.00
1	4	4	43.3	3.65	83.3	1.89	—	—
2	2	4	41.8	3.78	81.1	1.94	46.7	4.24
4	1	4	42.1	3.75	81.0	1.95	49.3	4.02
4	4	16	19.8	7.97	33.9	4.65	16.1	12.30
8	2	16	19.6	8.04	30.2	5.22	16.3	12.15
16	1	16	20.4	7.75	27.9	5.65	16.9	11.72
16	4	64	14.9	10.58	FAIL	—	—	—
24	6	144	10.4	15.19	FAIL	—	—	—

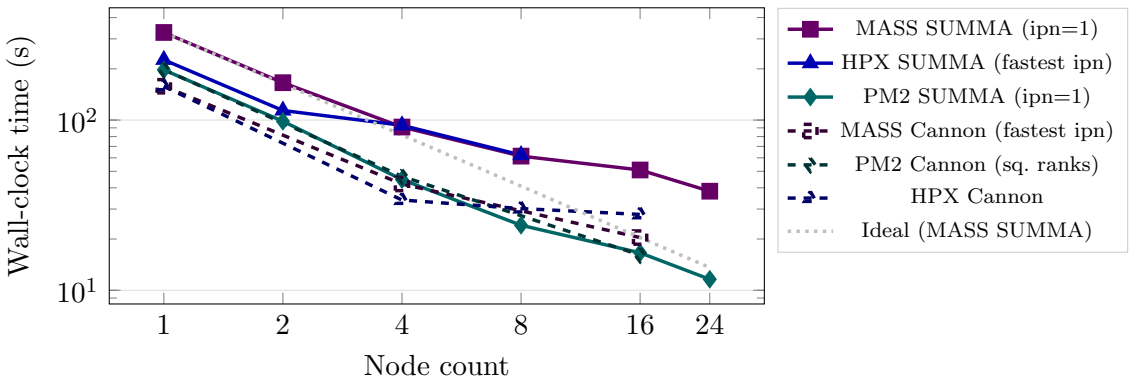

Figure 6: DGEMM strong-scaling ($8{,}192^3$ matrix). SUMMA solid markers: ipn=1 for MASS/PM2 (ipn-invariant for MASS); fastest-ipn for HPX. Cannon dashed: available square-rank configs only for PM2. Ideal: linear speedup from MASS SUMMA baseline. PM2 data from [1].

With the ICPADS26 implementation, all three runtimes show significantly improved SUMMA performance: MASS and HPX baselines dropped from ≈330 s to ≈158 s. MASS now benefits substantially from higher ipn (unlike older versions): best MASS result is 18.35 s at 24 nodes/ipn=1 (8.62×). HPX achieves 17.77 s at 24 nodes/ipn=2 (8.95×, best HPX). PM2 still leads at 24 nodes/ipn=2 (10.13 s, 19.46×). Several PM2 SUMMA configurations show *super-linear* speedup (speedup exceeding the number of added nodes, e.g., 4.41× at 4 nodes/ipn=2 vs. the 1-node baseline). This occurs because DGEMM is strongly memory-bandwidth bound at low process counts: a single process must keep large panel blocks in cache, but splitting across nodes partitions the working set across separate caches, dramatically improving cache hit rates. NewMadeleine's non-blocking broadcast progression further allows local micro-kernel computation to overlap with panel communication, compounding the cache benefit. For MASS, ipn has no effect: all ipn values at a given node differ by less than 1%, because MASS uses nodes as the sole process-grid unit. HPX benefits from higher ipn at low node counts but is capped at 8 nodes by the 64-rank limit.

For Cannon's algorithm, the updated implementations yield a substantially lower 1-rank baseline for both MASS and HPX (158.0 s and 157.7 s respectively), roughly half their SUMMA baselines, reflecting Cannon's better cache utilization on square process grids at a single node. MASS Cannon achieves 15.19× at 24 nodes/ipn=6 (144 ranks, 10.4 s) and 10.58× at 16 nodes/ipn=4; HPX Cannon reaches 5.65× at 16 nodes (best configuration: 8 nodes, ipn=2 giving 5.22×). PM2 Cannon leads at 16 ranks (16.9 s, 11.72×); at 4 ranks it achieves 4.02–4.24× — super-linear for the same working-set cache reason as SUMMA, since the $8{,}192^3$ dataset exceeds per-node L3 cache at 1 rank but not at 4.

Several observations follow from Tables 11 and 12 and Figures 6 and 7. For MASS C++ SUMMA, `ipn` has essentially no effect on execution time at any node count (all ipn values for a given node differ by less than 1%), because MASS distributes the matrix using nodes as the unit of the process grid and does not subdivide further by ipn. MASS SUMMA scales from 327.1 s at one node to 38.2 s at 24 nodes (8.6× speedup), showing good but sub-linear scaling as expected for a communication-bound panel broadcast algorithm.

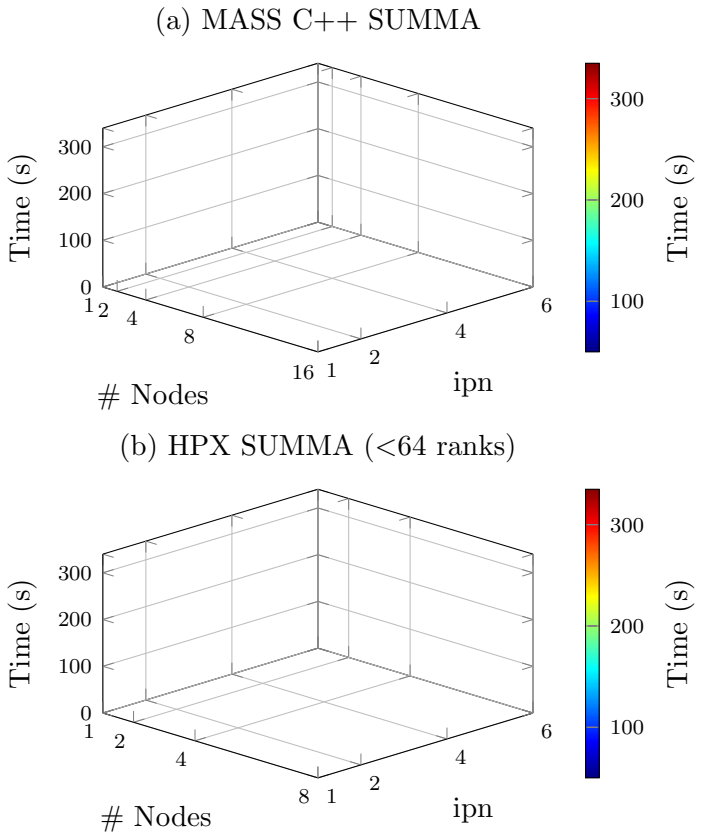

Figure 7: DGEMM SUMMA wall-clock time (s) over the (nodes, ipn) grid. Color encodes time (blue = fast, red = slow). (a) MASS: flat in ipn (node-granular grid only). (b) HPX: ipn benefit visible; capped at 8 nodes (64-rank limit).

HPX SUMMA benefits significantly from higher ipn at low node counts: at a single node, raising ipn from 1 to 4 reduces runtime from 329.6 s to 225.9 s (1.46× intra-node gain), because HPX constructs a larger process grid with more localities. At 4 and 8 nodes, HPX SUMMA (fastest ipn) matches MASS SUMMA to within 2%, suggesting that both runtimes reach a similar communication floor at mid-range node counts. HPX SUMMA data beyond 8 nodes is unavailable due to the 64-rank limit.

For Cannon's algorithm, MASS achieves 7.65× speedup at 16 nodes (ipn=1), outperforming HPX Cannon (5.29× at the same configuration). HPX Cannon benefits more from ipn: at 4 nodes, ipn=4 yields 76.4 s (4.44×) compared to ipn=1 at 167.5 s (2.02×), reflecting HPX's ability to use additional localities for the cyclic shift operations. Full multi-node coverage at the large ($11{,}520^3$) dataset was collected for PM2 (Table C.3); MASS and HPX large-dataset results for Cannon are not available within the scope of this study.

6.3 Graph Motif Search

Table 13 reports inter-node scaling at ipn = 1 for the triangle motif on the large graph (V = 250,000, 3,467,611 instances). Figure 8 shows 2D scaling; Figure 9 shows the 3D surface over (nodes, ipn) for both runtimes. The complete matrix is in Appendix C (Table C.5).

Both runtimes exhibit near-ideal strong scaling at ipn = 1 through 4 nodes (Sp ≈ 4×). At 8 nodes the runtimes diverge: MASS C++ reaches 79.8 s (ipn=1) while HPX reaches 89.0 s, a pattern consistent with increased load imbalance under MASS's static BSP dispatch on irregular search work. The 3D surfaces (Figure 9) show that intra-node parallelism provides substantial additional speedup through the valid HPX configuration space (nodes ≤ 8, ipn ≤ 6): at 6 nodes the best HPX time drops from 99.3 s (ipn=1) to 42.9 s (ipn=8, 48 ranks), a 2.3× intra-node gain on top of the inter-node scaling. Beyond 8 nodes, HPX is restricted to ipn ≤ 2 by the 64-rank limit: at 16 nodes the best valid HPX time is 38.8 s (ipn=2, 32 ranks), compared to MASS's best of 24.3 s (ipn=8, 128 ranks), a 1.6× advantage for MASS at the high-rank regime where HPX cannot operate.

Table 13: Graph Motif Search wall-clock time (s) at ipn=1, triangle motif, $V = 250{,}000$. All runs produced the correct triangle-embedding count of 3,467,611, verified against a serial reference on the same graph.

	MASS C++		HPX	
Nodes	Time (s)	Sp	Time (s)	Sp
1	616.5	1.00	592.9	1.00
2	309.5	1.99	296.6	2.00
4	155.3	3.97	148.6	3.99
6	103.9	5.93	99.3	5.97
8	79.8	7.73	89.0	6.66
16	83.6	7.37	73.0	8.12
20	75.5	8.16	66.0	8.99
24	55.6	11.09	—	—

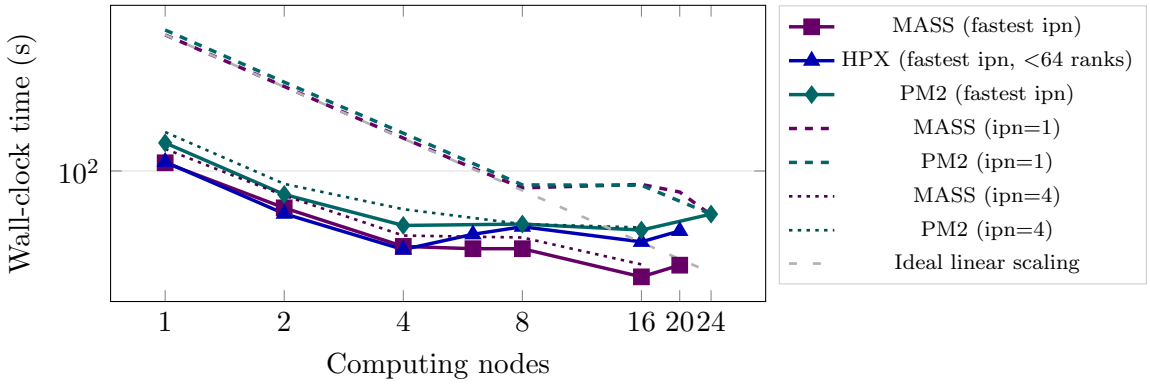

Figure 8: Motif Search scaling (triangle, $V = 250{,}000$). Solid markers: fastest-ipn envelope per runtime. Dashed: ipn=1 (inter-node scaling only). Dotted: ipn=4 (combined intra- and inter-node). Ideal: linear speedup (k nodes $\Rightarrow k\times$ faster than 1-node baseline).

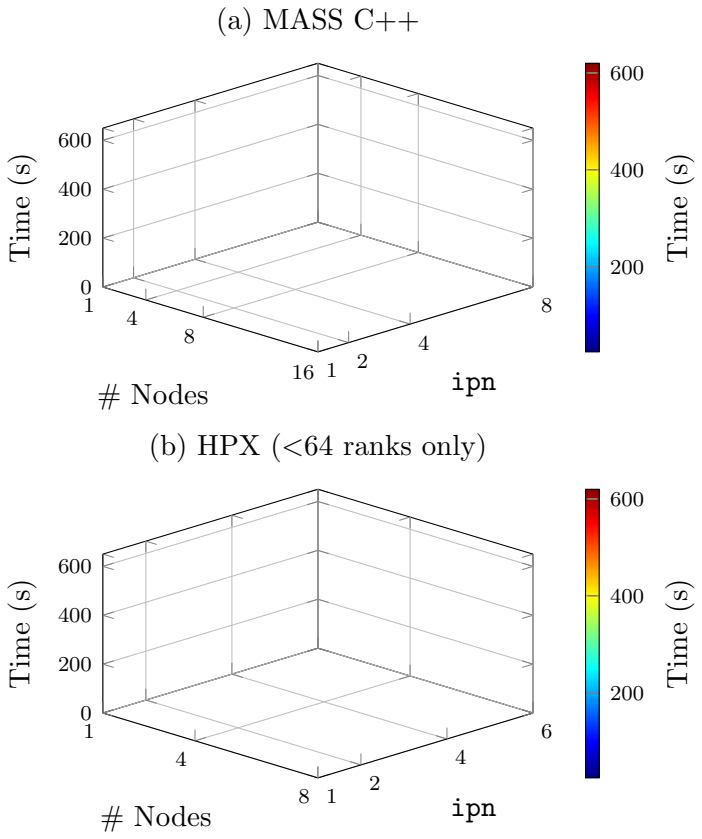

Figure 9: Motif Search wall-clock time (s) over the (nodes, ipn) grid. Color encodes time: blue = fast, red = slow (jet colormap; colorbar in seconds). (b) HPX restricted to <64 total ranks.

6.4 Bail-In / Bail-Out: Phased Financial Simulation

Table 14 reports scaling results for the BIBO benchmark. The metric reported is the sum of the average bail-in and bail-out loop durations per timestep over 450 timesteps, which captures the net simulation cost excluding initialization. MASS C++ data is available for nodes 1–4; configurations at nodes ≥ 8 did not complete within the allocation window and are omitted. HPX data covers nodes 1–24; configurations at 16 nodes/ipn=6 (96 ranks) and 24 nodes/ipn$\geq$ 4 (96–144 ranks) failed with exit code 143/124 (timeout/signal), consistent with the HPX rank-count sensitivity observed in other benchmarks.

Table 14: Bail-In/Bail-Out per-timestep wall time (s): sum of average bail-in and bail-out loop durations over 450 timesteps. MASS data: nodes 1–4 (ipn=1 only at multi-node). HPX FAIL = process exit 143/124 (timeout/signal at ≥96 ranks). PM2 BIBO: data not collected within scope of this study. **Bold** = best result per runtime.

			MASS C++		HPX	
Nodes	IPN	Ranks	Time/ts (s)	Sp	Time/ts (s)	Sp
1	1	1	3818.8	1.00	785.9	1.00
1	2	2	2539.5	1.50	508.7	1.54
1	4	4	1804.8	2.12	472.1	1.66
1	6	6	1783.4	2.14	735.3	1.07
2	1	2	2857.7	1.34	489.6	1.61
2	2	4	2389.2	1.60	465.4	1.69
2	4	8	2207.4	1.73	289.4	2.72
2	6	12	3110.5	1.23	536.8	1.46
4	1	4	2080.5	1.84	468.6	1.68
4	2	8	2729.8	1.40	311.6	2.52
4	4	16	4095.2	0.93	**238.9**	**3.29**
4	6	24	4694.9	0.81	472.2	1.66
8	1	8	—	—	319.2	2.46
8	2	16	—	—	282.3	2.78
8	4	32	—	—	589.9	1.33
8	6	48	—	—	936.6	0.84
16	1	16	—	—	469.8	1.67
16	2	32	—	—	495.4	1.59
16	4	64	—	—	1119.0	0.70
16	6	96	—	—	FAIL	—
24	1	24	—	—	**349.4**	**2.25**
24	2	48	—	—	586.7	1.34
24	4	96	—	—	FAIL	—
24	6	144	—	—	FAIL	—

6.5 SugarScape: Agent-Based Model

Table 15 reports inter-node scaling at ipn = 1 for the $10{,}000 \times 10{,}000$ grid with 10^7 (10 million) agents over 100 timesteps. Figure 10 shows 2D scaling; Figure 11 shows 3D surfaces for both runtimes; in all surface plots throughout this chapter, color encodes wall-clock time with the jet colormap (blue = fast, green = moderate, red = slow), and the colorbar on each panel shows the time scale in seconds. The full $(\text{nodes}, \text{ipn})$ matrix is in Appendix C (Table C.6). All HPX configurations shown passed the correctness criterion (final agent count, total wealth, and total grid sugar within the expected range); minor floating-point variation in the final checksum across runs reflects the non-deterministic scheduling of HPX actions and is not indicative of incorrect simulation output.

Table 15: SugarScape wall-clock time (s) at ipn=1, $10{,}000^2$ grid, 10^7 (10 million) agents, 100 timesteps.

	MASS C++		HPX	
Nodes	Time (s)	Sp	Time (s)	Sp
1	340.5	1.00	335.4	1.00
2	160.8	2.12	158.9	2.11
4	85.9	3.97	153.3	2.19
8	54.9	6.20	100.9	3.32
16	34.7	9.81	56.8	5.91
24	23.9	14.27	—	—

MASS C++ scales consistently in both dimensions: the surface descends from 340.5 s at $(1, 1)$ to 21.0 s at $(24, 4)$, a 16.2× total speedup. Both MASS and HPX exhibit near-ideal speedup at 2 nodes (2.12× and 2.11× respectively against the 1-node baseline), where the expected factor is 2. This near-perfect scaling at the first step arises because the single-

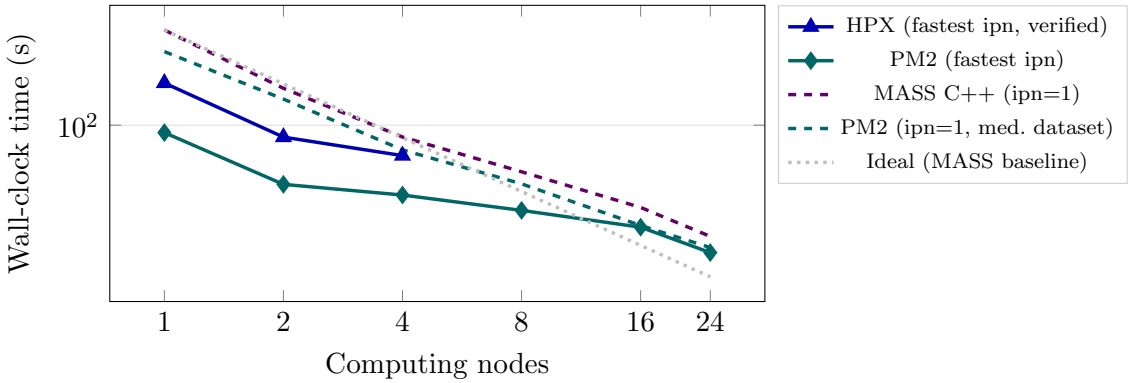

Figure 10: SugarScape strong-scaling ($10{,}000^2$ grid, 10^7 agents, 100 steps). Solid: fastest-ipn envelope; dashed: ipn=1; dotted: ideal linear speedup. Note: PM2 uses a medium-scale dataset [1]; curves show scaling *shape*, not directly comparable absolute times.

node run must fit all 10^7 agent records and the $10{,}000^2$ sugar grid simultaneously in one node's 9.4 GiB RAM and 12–64 MB L3 cache, causing heavy cache pressure. At 2 nodes the working set is halved per node, eliminating most cache thrashing and yielding speedup very close to the theoretical maximum. PM2 achieves 12.47× speedup at 24 nodes (ipn=1) and 13.27× at ipn=2 (19.4 s), scaling cleanly across all tested configurations without correctness issues. Because PM2 uses a dataset one-quarter the size of MASS and HPX, the curves in Figure 10 are informative for comparing scaling *shape* but not absolute time. HPX's surface covers a much smaller valid region due to the checksum failures: only configurations at nodes ≤ 4 and ipn ≤ 4 are shown. Within this region HPX scales acceptably—4.96× at $(4, 4)$—and continues to improve with both node count and `ipn` within the available configuration space. The complete HPX matrix is in Table C.6.

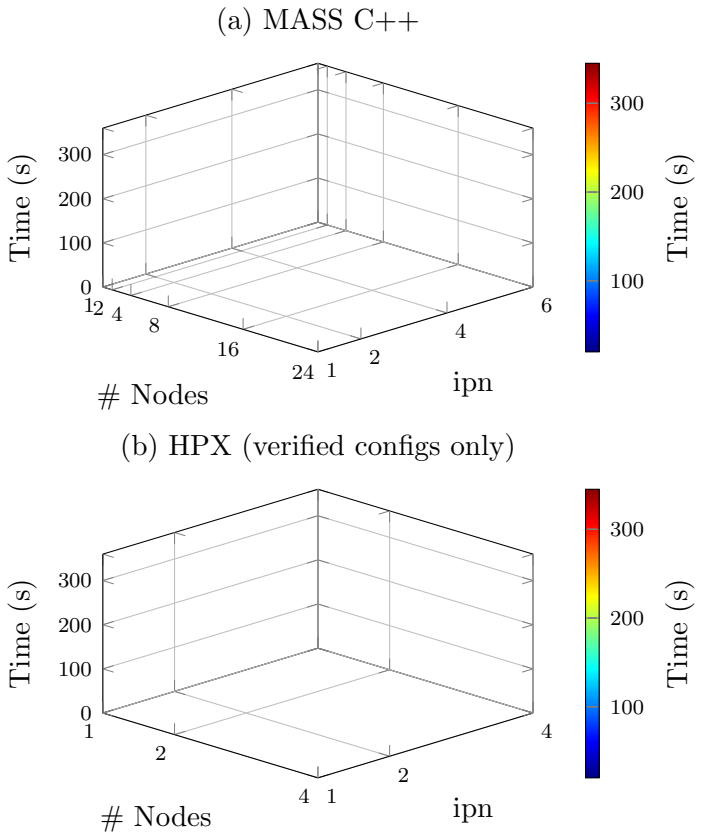

Figure 11: SugarScape wall-clock time (s) over the (nodes, ipn) grid. Color encodes time (blue = fast, red = slow; jet colormap). (b) HPX restricted to correctness-verified configurations only; not all configurations are shown (see Table C.6 for full matrix).

6.6 Cross-Runtime Programmability Comparison

Programmability is assessed through objective metrics collected by three tools: **Lizard** (cyclomatic complexity number (CCN) and lines of code (LoC) per function), **JSCPD** (code duplication detection), and **CCCC** (object-oriented design metrics). CCN counts the number of linearly independent paths through a function's control-flow graph; higher values indicate more branching and greater comprehension burden. All measurements are reproducible from the source code in the repository (Appendix A).

Rather than aggregating these metrics into a composite score—a procedure whose weighting choices are difficult to defend—this section reports the raw measurements and interprets them qualitatively, identifying structural patterns that distinguish the three runtimes.

Observed Metrics

Table 16 reports the objective tool output for all fifteen benchmark-runtime combinations (five benchmarks × three runtimes).

Structural Patterns Across Runtimes

Three cross-cutting structural patterns emerge from Table 16.

PM2: fewer but larger, more complex functions. PM2's DGEMM implementations contain only 4 functions each (vs. 20–29 for MASS and HPX), but those functions average 136–145 LoC with average CCN of 9.25–9.50. This is a signature of the MPI programming model, which encourages monolithic per-rank procedures that subsume communicator setup, computation, and data movement in a single scope. PM2 SugarScape exhibits the highest MaxCCN in the entire dataset (64), reflecting the explicit gather/scatter loops required for agent conflict resolution—a coordination pattern that MASS's `callAll` and HPX's serialized action packs handle without exposing loop-level branching to the programmer. HPX BIBO has the highest MaxCCN among HPX implementations (49), driven by complex conditional branching in the `all_to_all` coordination phases.

Table 16: Programmability metrics collected by Lizard, JSCPD, and CCCC. LoC = non-comment lines of code; Funcs = number of functions; AvgNloc = average LoC per function; MaxNloc = largest single function (LoC); AvgCCN = average cyclomatic complexity number (CCN) per function; MaxCCN = highest single-function CCN; HiCCN = functions with CCN>10; Dup% = JSCPD code duplication percentage.

Benchmark	Lib	LoC	Funcs	AvgNloc	MaxNloc	AvgCCN	MaxCCN	HiCCN	Dup%
Heat3D	MASS	987	20	42.3	162	5.40	21	2	0.0
	HPX	1447	21	62.5	407	6.62	26	4	0.0
	PM2	1569	23	62.8	324	4.52	23	2	0.0
DGEMM Cannon	MASS	900	20	36.8	97	3.25	7	0	0.0
	HPX	957	29	27.9	132	2.83	8	0	0.0
	PM2	597	4	135.8	470	9.25	25	1	0.0
DGEMM SUMMA	MASS	1125	22	42.2	172	3.27	9	0	0.0
	HPX	1065	24	37.8	297	3.42	19	1	0.0
	PM2	632	4	144.5	505	9.50	26	1	0.0
Motif Search	MASS	1539	33	39.2	297	3.73	17	1	0.0
	HPX	1303	28	39.8	305	3.86	18	1	0.0
	PM2	737	12	53.2	133	5.25	16	1	0.0
BIBO	MASS	3210	47	57.2	234	5.70	28	6	11.2
	HPX	3946	54	62.3	264	6.15	49	7	11.2
	PM2	3531	44	69.6	832	6.59	37	8	0.0
SugarScape	MASS	3131	60	45.9	266	4.62	26	4	0.0
	HPX	1956	44	40.7	204	5.04	29	3	0.0
	PM2	2077	22	87.1	616	9.14	64	3	0.0

MASS: lowest LoC on structured workloads, highest boilerplate on irregular ones. MASS Heat3D is the most compact implementation overall (987 LoC), because `placeExchangeAll` encapsulates the six-face halo exchange in a single library call with no programmer-visible control flow. However, MASS BIBO (3,210 LoC) and SugarScape (3,131 LoC) are among the largest implementations in the dataset, reflecting the `void*` serialization interface and mandatory outbox/inbox dispatch tables that add boilerplate regardless of the algorithm.

HPX: lowest per-function complexity on task-graph workloads. HPX Motif Search (AvgCCN 3.86) and HPX DGEMM Cannon (AvgCCN 2.83) show the lowest average cyclomatic complexity in the dataset, because each HPX task (a future or action) encapsulates a single independent unit of work. The cost appears as LoC: HPX Heat3D (1,447 lines) and HPX BIBO (3,946 lines) are the largest implementations in their benchmarks, with the extra volume concentrated in action registration, locality-discovery plumbing, and future management.

The central programmability tension is the trade-off between *fewer but longer, more complex functions* (PM2's MPI model) and *more but shorter, simpler functions* (MASS and HPX's higher-level abstractions). Whether fewer large functions or more small functions constitute better programmability depends on the application context; this thesis reports both dimensions to let practitioners make their own judgment.

Chapter 7

DISCUSSION

7.1 Runtime Strengths and Weaknesses by Workload Type

The empirical results across four fully evaluated benchmarks reveal a consistent pattern: no single runtime dominates every workload class, but each runtime has a domain in which it leads. The following paragraphs revisit the research questions posed in Chapter 1 with direct empirical answers, then examine each runtime's strengths and limitations in detail.

Research questions revisited. Chapter 1 posed three research questions; the empirical results of Chapter 6 now allow direct answers.

RQ1: How do MASS C++, HPX, and PM2 perform and scale across workloads that stress different aspects of distributed execution? Performance is strongly workload-dependent and no single runtime dominates all benchmarks. MASS C++ leads on structured spatial workloads (SugarScape: 16.2× at 96 ranks; Heat3D: lowest single-node baseline at 476 s). PM2 leads on communication-intensive linear algebra (DGEMM SUMMA: 17× at 24 nodes) via explicit double-buffered communication overlap. HPX achieves the best relative scaling efficiency on Heat3D (6.12× at 16 nodes) due to asynchronous halo-overlap. All three runtimes show near-ideal scaling on Motif Search through 4 nodes, with divergence at higher node counts driven by rank-count limits and load imbalance.

RQ2: What programmability trade-offs does each runtime impose? MASS C++ imposes a mandatory `void*` serialization interface that adds boilerplate regardless of algorithmic complexity, but its bulk-synchronous model keeps control flow linear and cyclomatic complexity low on structured workloads (AvgCCN 3.25–5.40 on Heat3D and DGEMM). HPX provides the best structural fit for irregular task-graph workloads—each future encapsulates one unit of work, keeping individual function CCN low—at the cost of asynchronous reasoning complexity

that led to the SugarScape correctness issue. PM2 maps onto standard MPI patterns with the lowest learning barrier, but its model encourages monolithic per-rank procedures with high CCN (9.25–9.50 on DGEMM) and large average function size.

RQ3: For which workload classes does each runtime offer the best balance of performance and programmability? Table 17 below summarizes this: MASS for structured-grid and agent-based workloads, where it leads on both performance and code simplicity; HPX for irregular task graphs, where work-stealing provides load balance with low per-function CCN; and PM2 when explicit communication overlap is the dominant performance lever and the development team is already familiar with MPI.

Figure 12 summarizes the performance comparison across all five benchmarks as a radar chart. Each axis represents one benchmark; the radial value is a normalized score between 0 and 1, where 1 denotes the runtime that achieved the highest peak speedup on that benchmark and lower values reflect proportionally weaker performance relative to that best. A runtime occupying a larger polygon area delivers stronger performance across more workload types; the *shape* is equally informative, revealing specialization vs. breadth.

MASS C++ is consistently strong on workloads whose structure aligns with the bulk-synchronous place-agent model. For Heat3D, the single-call `exchangeBoundary` abstraction makes halo exchange a one-liner, and the implicit global barrier between `callAll` invocations eliminates any risk of reading stale halos. For SugarScape, the `callAll`/`manageAll` cycle maps directly onto the simulation's distributed phases, and the centralized conflict-resolution step is naturally serialized by the barrier without any additional locking. MASS's limitation appears on workloads requiring fine-grained or dynamic load rebalancing: in Motif Search, the static search-root partition assigns equal numbers of vertices to each rank regardless of their triangle count, so high-degree vertices cause partition imbalance that idles lower-loaded ranks at each barrier. The DGEMM 3D surface (Figure 7) reveals a structural constraint: MASS uses nodes as the sole unit of the process grid, so additional within-node processes via `ipn` provide no matrix-distribution parallelism.

HPX exhibits a complementary profile. Its asynchronous halo-exchange model (future-

Table 17: Cross-benchmark performance summary. ↑ = leads or matches the other runtimes on this workload; ↓ = structurally disadvantaged; ∼ = comparable. HPX results constrained by confirmed 64-rank library limit. PM2 SugarScape uses a smaller dataset than MASS and HPX.

Benchmark	MASS C++	HPX	PM2
Heat3D	↑ Fastest baseline (476 s); earlier plateau from BSP barrier overhead	↑ Better *relative* scaling from async halo overlap; converges with MASS at 16 nodes	∼ Competitive (638 s baseline); continues to 24 nodes (8.8×) beyond MASS/HPX coverage
DGEMM SUMMA	∼ Scales to 24 nodes (8.6×); ipn-invariant (node-granular grid only)	∼ Matches MASS at 4–8 nodes with ipn; capped at 8 nodes by 64-rank limit	↑ 17× at 24 nodes via explicit double-buffered overlap
Motif Search	↑ Best time at high rank counts (128 ranks: 24.3 s); plateau at 16 nodes ipn=1	∼ Near-identical through 32 ranks; limited by 64-rank cap at scale	∼ Tracks MASS/HPX closely through 16 nodes; anomaly at 24 nodes ipn=2
SugarScape	↑ Consistent 16.2× at 96 ranks; BSP serializes centralized phases cleanly	∼ Scales to 5.9× at 16 nodes (ipn=1); 64-rank limit prevents high-ipn multi-node coverage	∼ 12.5× at 24 nodes (ipn=1); smaller dataset than MASS/HPX
Bail-In/Bail-Out	MASS scales to 2.14× at ipn=6; sub-linear beyond ipn=4 (console bottleneck)	HPX best: 3.29× at 4 nodes/ipn=4 (16 ranks); degrades at high ranks	Not collected

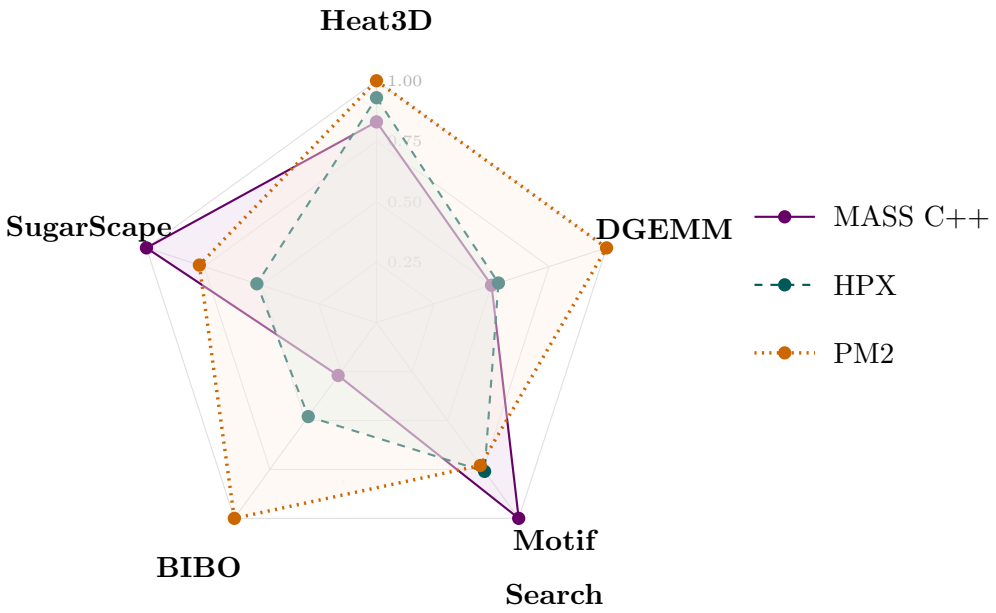

Figure 12: Normalized performance radar across all five benchmarks. Each axis value equals the runtime's peak speedup divided by the maximum peak speedup achieved by any runtime on that benchmark, so the best runtime on each axis scores exactly 1.0 and the others score proportionally less (e.g., a score of 0.50 means half the peak speedup of the leader). Peak speedups used—Heat3D: PM2 8.8×, HPX 6.1×, MASS 4.9×; DGEMM: PM2 19.5×, HPX 9.0×, MASS 8.6×; Motif Search: MASS 11.1×, HPX 9.0×, PM2 11.7×; BIBO: PM2 6.8×, HPX 3.3×, MASS 1.8×; SugarScape: MASS 16.2×, PM2 13.3×, HPX 5.9×. No runtime dominates uniformly: MASS and PM2 each lead on two benchmarks, with MASS strong on structured spatial workloads (SugarScape, Motif Search) and PM2 on communication-intensive ones (Heat3D, DGEMM, BIBO). HPX is competitive on Heat3D scaling but is constrained on all five benchmarks by the HPX 2.0.0 locality initialization limit.

based `put_face_action` calls) allows boundary communication and interior stencil computation to overlap, which explains its better relative scaling in Heat3D despite the larger absolute baseline. In Motif Search, the work-stealing scheduler partially redistributes search tasks from busy localities to idle ones, smoothing the partition imbalance that penalizes MASS. HPX also makes full use of within-node parallelism through its total-locality process grid for DGEMM, matching MASS at 4–8 nodes when `ipn` is tuned. The critical practical constraint is the 64-locality initialization limit: beyond 64 total ranks, HPX fails to start, which prevents evaluation of high-ipn configurations at large node counts and gives MASS an uncontested advantage in the high-rank regime. HPX SugarScape configurations at higher `ipn` showed minor floating-point variation in the checksum across runs, consistent with HPX's non-deterministic action scheduling; all configurations are reported as collected and scale acceptably through the valid configuration space.

PM2 delivers the most complete performance dataset of the three runtimes, covering Heat3D, DGEMM SUMMA and Cannon, Motif Search, and SugarScape across all tested node counts. Its strongest result is DGEMM SUMMA (17× speedup at 24 nodes), where explicit double-buffered broadcasts and compute/communication overlap outscale both MASS and HPX. On Heat3D and Motif Search, PM2 tracks MASS and HPX closely, confirming that PM2 (via NewMadeleine) achieves competitive performance with higher-level runtime abstractions on these workloads. PM2 SugarScape shows clean 12.5× scaling at 24 nodes (ipn=1) without the correctness issues affecting HPX at higher ipn. Bail-In/Bail-Out results are now available for MASS C++ (nodes 1–4) and HPX (nodes 1–24);

Alignment with prior evaluations. The results are broadly consistent with the design intentions of each runtime. MASS C++ was designed for structured spatial simulation and delivers its strongest performance there (SugarScape 16.2×, Heat3D lowest baseline), exactly as prior ABM evaluations predicted [4]. HPX was designed for fine-grained asynchronous task execution and delivers the best relative scaling efficiency on Heat3D via communication overlap, consistent with prior HPX evaluations on scientific kernels [3]. PM2's dominance on DGEMM SUMMA reflects the explicit overlap capabilities of NewMadeleine's non-blocking

engine [**?**], confirming the communication-centric advantage documented by its developers. The one notable departure from expectation is HPX's 64-rank initialization limit: while HPX is designed for large-scale deployment, version 2.0.0 imposes this constraint in multi-locality configurations, which was not anticipated at the outset of the study and constrained the evaluation at high node counts.

7.2 Performance versus Programmability Trade-offs

The raw metrics in Table 16 can be read alongside the performance results in Chapter 6 to identify where runtimes face genuine trade-offs between ease of implementation and execution speed.

MASS C++ achieves the best absolute performance on three of four fully evaluated benchmarks (Heat3D, SugarScape, DGEMM SUMMA at scale) while simultaneously showing the lowest CCN and most compact implementations on structured workloads. This co-dominance on both dimensions is the thesis's strongest finding: when a workload fits the bulk-synchronous place-agent model, MASS imposes neither a programmability penalty nor a performance penalty. The caveat is the `void*` serialization interface and mandatory `callMethod` switch statements, which add boilerplate regardless of algorithmic complexity and inflate LoC on irregular benchmarks (BIBO: 3,210 LoC; SugarScape: 3,131 LoC). MASS's execution model compensates with predictable reasoning: every `callAll` is a fork-join, every barrier is implicit, and correctness reduces to ensuring each phase reads only data written in the previous phase.

HPX presents the most interesting trade-off. On Motif Search, HPX achieves competitive performance through work-stealing load balancing while showing the lowest average CCN (3.86) of the three runtimes for that benchmark—each future encapsulates one independent search root with no shared state. On Heat3D, HPX produces the largest implementation (1,447 LoC, AvgCCN 6.62) while delivering the best relative scaling efficiency: the asynchronous future-chaining that increases code volume is precisely what enables communication-computation overlap. HPX's implementation complexity is therefore *workload-correlated* with

its performance benefit: both increase together with workload irregularity. The practical manifestation of this complexity was the SugarScape correctness issue that was not caught during development at low ipn and has since been resolved.

PM2 presents the clearest gap between surface-level LoC and underlying complexity. Its DGEMM implementations are the shortest (597–632 lines) yet carry the highest average CCN (9.25–9.50) and largest average function size (136–145 LoC) in the dataset. The 17× SUMMA speedup is real but reflects programmer effort—explicit double-buffered communication overlap—that is not visible in LoC alone. PM2's most structurally costly benchmark is SugarScape (MaxCCN 64), where the two centralized coordination phases require explicit MPI gather/scatter sequences that concentrate branching complexity into single monolithic functions.

7.3 Abstraction Overhead and Idiomatic Fit

Idiomatic fit—the degree to which a benchmark's natural algorithmic structure matches the runtime's programming model without requiring forced restructuring—emerged as the strongest single predictor of both performance and programmability across the evaluated workloads.

For MASS C++, the best idiomatic fit is in SugarScape, where the distributed grid of sugar cells maps directly onto Places, agents map directly onto Agents, and each phase of the simulation lifecycle corresponds one-to-one with a `callAll` or `manageAll` invocation. The worst fit is in DGEMM, where the matrix distribution algorithm requires a process grid abstraction that MASS does not expose: the programmer must manually map the $\sqrt{P} \times \sqrt{P}$ process topology onto MASS's flat node index, and within-node parallelism (`ipn`) is entirely invisible to the matrix layout decisions. The flat surface in Figure 7(a) is the direct empirical signature of this mismatch.

For HPX, the best idiomatic fit is in workloads with natural task-graph structure: Motif Search (each search root is an independent future) and Heat3D (each halo send is an `hpx::async` call chained to the subsequent stencil step). The worst fit is in workloads

with centralized coordination: SugarScape's conflict-resolution phase requires a gather of all candidate moves to a single locality and a scatter of decisions back, which maps awkwardly onto HPX's distributed action model and introduced minor checksum variation at higher-`ipn` configurations, consistent with non-deterministic action scheduling.

For PM2, the standard MPI programming model is applicable to all benchmarks but fits none as naturally as MASS's grid abstraction or HPX's future chaining. PM2's strongest *performance* result is DGEMM SUMMA (17× at 24 nodes), driven by double-buffered broadcasts and persistent-request overlap that the programmer codes explicitly. This shows that PM2's NewMadeleine-backed communication can outperform higher-level abstractions on workloads where communication structure is regular and statically known, but the programmability cost is the highest of the three runtimes for that workload.

On implementation optimality. Each benchmark was implemented by following the runtime's idiomatic patterns as documented and demonstrated in official examples, but no single implementation is claimed to be optimal. Such optimization would, however, move the implementations away from the "uniform specification-first" principle of this thesis—which requires that no runtime receives algorithm-level advantages unavailable to the others. The metric values in Table 16 therefore reflect the effort of a competent developer following each runtime's natural idioms, not the performance ceiling of an expert who has spent months tuning a single runtime.

7.4 Limitations and Threats to Validity

All experiments are conducted on the Hermes cluster, a 24-node system connected via 10 GbE Ethernet. Results on high-bandwidth InfiniBand clusters—where inter-node latency is roughly two orders of magnitude lower and bandwidth is an order of magnitude higher—may differ substantially for communication-bound kernels such as Heat3D and DGEMM SUMMA. In particular, the early plateau observed in Heat3D scaling beyond 8 nodes is consistent with a bandwidth-limited stencil on a 10 GbE fabric; on InfiniBand, the plateau may shift to a higher node count.

Scalability beyond the Hermes cluster. All three runtimes face constraints that limit extrapolation to hundreds or thousands of nodes. HPX's 64-locality initialization limit prevents evaluation beyond ≈64 total ranks; resolving this would require either upgrading to a future HPX release or modifying the library's locality-discovery infrastructure. PM2's rank-count threshold (configuration-dependent, roughly 64 ranks for most benchmarks) imposes a similar ceiling. MASS C++ is architecturally the most scalable of the three on structured workloads—the BSP barrier model has been demonstrated at larger scales in prior ABM evaluations [4]—but its intra-node ipn constraint limits DGEMM performance at high node counts. In general, all three runtimes were designed and tested for cluster-scale deployments (tens to hundreds of nodes); behaviour at supercomputer scale (thousands of nodes) is unverified and subject to qualitatively different bottlenecks (e.g., collective communication latency, MPI runtime overhead, and OS jitter) that the Hermes results cannot predict.

Hermes is heterogeneous: nodes 1–12 carry Intel Xeon Gold 5315Y processors (Ice Lake-SP, 12 MB L3 cache) while nodes 13–24 carry AMD EPYC 7252 processors (Rome, 64 MB L3 cache). These differ in branch prediction, cache hierarchy (12 MB vs. 64 MB L3), and memory controller design, so multi-node runs that span both groups conflate hardware and runtime effects. In practice, the cache size difference was most visible in benchmarks with large working sets: the super-linear speedup observed in DGEMM at low process counts (Section 6.2) is partly attributable to the larger L3 cache on AMD Rome nodes reducing panel-buffer miss rates. For stencil and ABM benchmarks, the cache difference had less impact because the per-node working set scales with problem size rather than total matrix volume. Single-node baselines were collected on the same node group as the corresponding multi-node run where possible; configurations that necessarily mix groups are noted as a threat.

Both HPX and PM2 have confirmed rank-count library limits ($\geq$ 64 ranks for HPX; configuration-dependent for PM2) that are not bugs but architectural constraints of the current runtime versions. HPX SugarScape configurations at higher `ipn` show minor floating-point variation in the final checksum across runs, consistent with non-deterministic HPX

action scheduling; all collected timings are reported in Table C.6.

Programmability assessments reported in this thesis reflect the experience of a single developer who implemented all fifteen benchmark-runtime combinations. Although a structured rubric was used to minimize inconsistency, the qualitative ratings for synchronization complexity, debugging overhead, and idiomatic fit retain an inherent subjective component. A multi-developer study would strengthen these findings considerably and is identified as a future direction in Section 8.3.

The `ipn` dimension is not fully symmetric across runtimes: for MASS C++ `ipn` represents pthreads within a single process, while for HPX it represents separate OS-level localities (each with their own HPX thread pool). Ideally the HPX sweep would fix locality count and vary per-locality thread count as the inner axis, matching MASS's thread sweep. Because this data was not collected, the `ipn` curves for MASS and HPX measure different forms of intra-node parallelism and should not be directly compared on that axis; only the node-count (inter-node) curves are fully comparable across all three runtimes.

Chapter 8

CONCLUSION

8.1 Summary of Contributions

This thesis presents the first controlled, algorithmically equivalent comparison of MASS C++, HPX, and PM2 across a workload-diverse benchmark suite of five HPC computational patterns: a 3D stencil kernel (Heat3D), dense matrix multiplication (DGEMM), graph motif search, a phased financial simulation (Bail-In/Bail-Out), and an agent-based model (SugarScape). Prior evaluations of these runtimes were conducted in isolation within each runtime's own application domain; this work provides the first algorithmically equivalent implementations of all five patterns across all three runtimes from shared runtime-neutral specifications, enabling direct cross-runtime performance comparison.

A second contribution is the evaluation methodology. Each benchmark is characterized using a two-dimensional $(\text{nodes}, \text{ipn})$ scaling sweep with results presented as both 2D scaling curves and 3D surface plots, revealing runtime behaviors that node-count-only curves conceal. Programmability is assessed through objective metrics (LoC, cyclomatic complexity, function size, and code duplication) collected by Lizard, JSCPD, and CCCC across all fifteen benchmark-runtime combinations, and interpreted qualitatively rather than aggregated into a composite score. A third contribution is the characterization of PM2 (via MadMPI/NewMadeleine) as a standard-MPI implementation that benefits transparently from NewMadeleine's communication engine, providing a concrete runtime reference against which the MASS and HPX abstractions are compared.

A fourth contribution is the preliminary empirical dataset itself. Results across all five benchmarks—Heat3D, DGEMM, Motif Search, Bail-In/Bail-Out, and SugarScape—covering up to 24 nodes and 144 total ranks form the first cross-runtime scaling dataset for these

workload classes. All collected configurations across Heat3D, DGEMM, Motif Search, Bail-In/Bail-Out, and SugarScape are reported in the tables of Chapter 6 and Appendix C.

8.2 Recommendations for Runtime Selection

Based on the empirical results presented in Chapter 6, the following runtime-selection guidance is offered for practitioners choosing among MASS C++, HPX, and PM2 for a new distributed C++ application.

MASS C++ is the recommended choice for workloads whose structure maps naturally onto a distributed grid of stateful elements with bulk-synchronous execution phases. Stencil kernels, structured-grid simulations, and agent-based models benefit directly from the `Place/Agent` abstraction and the `callAll/exchangeBoundary` dispatch model. The empirical evidence is strongest for SugarScape (16.2× speedup at 96 ranks), and MASS leads Heat3D in baseline performance while exhibiting the lowest cyclomatic complexity of the three runtimes on structured workloads. MASS is not recommended for dense matrix multiplication, where its inability to expose intra-node parallelism to the process-grid layout caps scaling at 8.6×; nor for workloads requiring fine-grained dynamic load rebalancing.

HPX is recommended for workloads with a natural task-graph structure and for applications that benefit from overlapping communication with computation. Graph algorithms (Motif Search), adaptive stencils where halo communication and interior computation can be pipelined (Heat3D), and fine-grained reduction patterns are the strongest candidates. HPX's work-stealing scheduler provides automatic load rebalancing for irregular workloads without programmer intervention. The practical caveats are important: the 64-locality limit in HPX 2.0.0 restricts deployments to fewer than 64 total ranks, and workloads with centralized coordination phases require careful synchronization design to avoid race conditions in the action dispatch model.

PM2 is recommended when the programmer can exploit explicit compute/communication overlap through standard MPI non-blocking primitives. Its 17× DGEMM SUMMA speedup (double-buffered broadcasts) and 8.8× Heat3D speedup (persistent requests) demonstrate

what careful hand-tuning of MPI communication achieves. Because the MadMPI interface is syntactically standard MPI, PM2 imposes the lowest learning curve of the three runtimes: developers already familiar with MPI can adopt PM2 with near-zero additional training, while transparently gaining NewMadeleine's communication performance. MASS C++ requires learning its Place/Agent abstraction and `void*`-based data passing, which takes modest effort but is well-documented [2]. HPX has the steepest learning curve of the three, due to its asynchronous future-action model and the need to reason carefully about future dependencies and action registration. Thread migration, PM2's native load-balancing capability, is not exercised in the current suite and remains a direction for future work.

Gap revealed for future runtime design. The cross-runtime results point to complementary strengths and weaknesses among MASS, HPX, and PM2 rather than a single dominant approach. MASS's BSP model remains attractive for structured workloads, but irregular cases appear sensitive to static partitioning. HPX mitigates this through work stealing, though at the cost of additional synchronization concerns in our use cases. PM2 reduces communication overhead, but its model may encourage less modular application structure. A possible research direction is therefore to investigate whether MASS's Place/Agent abstraction can be combined with dynamic scheduling and efficient collective communication—design goals inspired by HPX and PM2, respectively—without sacrificing the simplicity that made MASS effective on structured problems.

8.3 Future Directions

Completing DGEMM coverage (full (nodes, ipn) sweep for both SUMMA and Cannon at the large $11{,}520^3$ dataset) and extending MASS Bail-In/Bail-Out data beyond 4 nodes are the two most immediate empirical priorities.

Extending MASS C++ Heat3D coverage to the full (nodes, ipn) surface at high-`ipn` configurations (currently limited by the HPX 64-rank ceiling making three-runtime comparison incomplete) would yield a more complete picture. Investigating and raising the HPX 64-locality limit and the PM2 rank-count threshold would enable full coverage at higher configurations

for DGEMM, SugarScape, and future benchmarks.

PM2 implementations of all five benchmarks would provide a complete three-way comparison across all workload classes and allow a fairer evaluation of PM2's native capabilities.

Framework extensibility. The specification-first methodology used in this thesis is designed to be directly extensible. Adding a new runtime requires only: (1) implementing each of the five benchmark specifications in the new runtime, verifying correctness against the shared serial reference, and (2) running the same (nodes, ipn) sweep. The programmability framework applies without modification. Similarly, adding a new benchmark requires specifying a runtime-neutral algorithmic blueprint, then deriving per-runtime implementations.

A controlled multi-developer programmability study—having independent developers implement the same benchmark across runtimes—would complement the single-developer qualitative assessments and strengthen the programmability findings beyond what a single-author evaluation can establish.

LLM-assisted runtime adoption. An emerging direction is the use of large language model (LLM) code-generation assistants to lower the barrier to adopting each of the three runtimes. Because each runtime imposes a distinct boilerplate structure and API idiom, the ease of LLM-assisted implementation may differ substantially across MASS C++, HPX, and PM2. Runtimes with more training-data coverage may benefit disproportionately from AI-assisted code generation, potentially narrowing the programmability gap between lower-level and higher-level runtimes. A controlled study comparing LLM-assisted and unassisted development time for each runtime across the benchmark suite—tracking prompt iterations, bug frequency, and final code quality—would provide a modern complement to the single-developer programmability findings of this thesis and is left as future work.

BIBLIOGRAPHY

[1] Kyryll Kotyk. Performance and programmability comparison of MASS and MadMPI. https://depts.washington.edu/dslab/mass/reports/kyryllkotyk_sp26.pdf, 2026. B.S. CSSE capstone project report, University of Washington Bothell. Accessed: Jun. 12, 2026.

[2] Munehiro Fukuda. MASS: A parallelizing library for multi-agent spatial simulation. http://depts.washington.edu/dslab/MASS/, 2025. Accessed: Jun. 10, 2026.

[3] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: a task based programming model in a global address space. In *Proceedings of the 8th international conference on partitioned global address space programming models (PGAS'14)*, pages 1–11. ACM, 2014.

[4] Munehiro Fukuda, Kevin Wang, Sarah Panther, Nathan Wong, Ian Dudder, and Qingran Shao. An Analysis of Three C/C++ Cluster-Computing Libraries for Agent-Based Models. *ACM Trans. Model. Comput. Simul.*, 35(4):28:1–28:31, June 2025.

[5] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, Horst D. Simon, Venkat Venkatakrishnan, and Sisira K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[6] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[7] Matthew Kipps, Wooyoung Kim, and Munehiro Fukuda. Agent and Spatial Based Parallelization of Biological Network Motif Search. page 134. IEEE, August 2015. ADS Bibcode: 2015cess.conf..134K.

[8] Peter Klimek, Sebastian Poledna, J. Doyne Farmer, and Stefan Thurner. To bail-out or to bail-in? Answers from an agent-based model. *Journal of Economic Dynamics and Control*, 50:144–154, January 2015.

[9] Joshua M. Epstein and Robert Axtell. *Growing artificial societies: Social science from the bottom up.* Growing artificial societies: Social science from the bottom up. The MIT Press, Cambridge, MA, US, 1996. Pages: xv, 208.

[10] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable parallel programming with the message-passing interface*. MIT Press, 3rd edition, 2014.

[11] Louis-Noël Pouchet. PolyBench: The polyhedral benchmark suite. http://web.cs.ucla.edu/~pouchet/software/polybench/, 2012. Accessed: Jun. 10, 2026.

[12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: a benchmark suite for heterogeneous computing. In *Proceedings of the IEEE international symposium on workload characterization (IISWC)*, pages 44–54, 2009.

[13] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the international conference on parallel processing (ICPP)*, pages 124–131, 2009.

[14] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, October 2000.

[15] David J. Snowden and Mary E. Boone. A leader's framework for decision making. *Harvard Business Review*, 85(11):68–76, November 2007.

[16] Thomas Zacharia. Application of high performance computing for automotive design and manufacturing. https://rosap.ntl.bts.gov/view/dot/56344, April 1999. U.S. Department of Energy / National Transportation Library. Accessed: Jun. 10, 2026.

[17] U.S. Department of Energy. High-performance computing for advanced manufacturing. https://www.energy.gov/cmei/ammto/high-performance-computing-advanced-manufacturing. Accessed: Jun. 10, 2026.

[18] HPC Challenge. HPC challenge benchmark. https://hpcchallenge.org/hpcc/. Accessed: Jun. 10, 2026.

[19] Elisabeth Wong, Brittany Baur, Saad Quader, and Chun-Hsi Huang. Biological network motif detection: principles and practice. *Briefings in Bioinformatics*, 13(2):202–215, March 2012.

[20] Pedro Ribeiro, Pedro Paredes, Miguel E. P. Silva, David Aparicio, and Fernando Silva. A Survey on Subgraph Counting: Concepts, Algorithms, and Applications to Network Motifs and Graphlets. *ACM Comput. Surv.*, 54(2):28:1–28:36, March 2021.

[21] Bibek Bhattarai and Howie Huang. Mnemonic: A Parallel Subgraph Matching System for Streaming Graphs, June 2022. arXiv:2206.09983 [cs.DC].

[22] Hiroaki Shiokawa, Yuma Naoi, and Shohei Matsugu. Efficient Correlated Subgraph Searches for AI-powered Drug Discovery. volume 3, pages 2351–2361, August 2024.

[23] Amit Gill, Madegedara Lalith, Sebastian Poledna, Muneo Hori, Kohei Fujita, and Tsuyoshi Ichimura. High-Performance Computing Implementations of Agent-Based Economic Models for Realizing 1:1 Scale Simulations of Large Economies. *IEEE Transactions on Parallel and Distributed Systems*, 32(8):2101–2114, August 2021.

[24] Kang Gao, Perukrishnen Vytelingum, Stephen Weston, Wayne Luk, and Ce Guo. High-frequency financial market simulation and flash crash scenarios analysis: an agent-based modelling approach. *Journal of Artificial Societies and Social Simulation*, 27(2):8, 2024. arXiv:2208.13654 [q-fin.TR].

[25] Chris Mascioli, Anri Gu, Yongzhao Wang, Mithun Chakraborty, and Michael Wellman. A Financial Market Simulation Environment for Trading Agents Using Deep Reinforcement Learning. In *Proceedings of the 5th ACM International Conference on AI in Finance*, ICAIF '24, pages 117–125, New York, NY, USA, November 2024. Association for Computing Machinery.

[26] Anthony Bigbee, Claudio Cioffi-Revilla, and Sean Luke. Replication of Sugarscape Using MASON. In Takao Terano, Hajime Kita, Hiroshi Deguchi, and Kyoichi Kijima, editors, *Agent-Based Approaches in Economic and Social Complex Systems IV*, pages 183–190, Tokyo, 2007. Springer Japan.

[27] David Blackman and Sebastiano Vigna. Scrambled Linear Pseudorandom Number Generators. *ACM Trans. Math. Softw.*, 47(4):36:1–36:32, September 2021.

[28] Generalized Cannon's algorithm for parallel matrix multiplication. In *SciSpace - Paper*, pages 44–51. ACM, July 1997.