MASS JAVA LIBRARY TOWARDS ITS USE FOR A GRAPH DATABASE SYSTEM

Atul Ahire

A whitepaper submitted in partial fulfillment of the requirements of the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

2025

Project Committee:

Professor Munehiro Fukuda, Committee Chair

Professor Robert Dimpsey, Committee Member

Professor Brent Lagesse, Committee Member

Program Authorized to Offer Degree:
University of Washington

University of Washington

Abstract

MASS JAVA LIBRARY TOWARDS ITS USE FOR A GRAPH DATABASE SYSTEM

Atul Ahire

Chair of the Supervisory Committee:
Dr. Professor Munehiro Fukuda
Computer Science and Software Engineering

Distributed graph databases are foundational to modern applications such as social networks, recommendation systems, and transportation platforms. The MASS (Multi-Agent Spatial Simulation) framework—an agent-based parallel computing library for distributed systems—designed to simulate spatial computations by deploying mobile agents over distributed data structures such as graphs. MASS's agent-based graph computing model enables intuitive programming and parallel traversal, making it well-suited for scalable graph database applications. This research intends to address two core limitations in its graph database model: inefficient vertex placement and high communication overhead during agent migration. To improve data locality and reduce inter-node communication during graph traversal, we explored and integrated graph partitioning strategies such as METIS, which minimizes edge cuts and better aligns with graph topology. This approach significantly reduced inter node communication for graph benchmarks like Triangle Counting and Articulation Points by localizing vertex adjacency across fewer computing nodes.

The second challenge targeted the messaging inefficiencies caused by TCP-based agent migration. We redesigned the MASS communication infrastructure using Aeron, a high-performance UDP messaging library, to support low-latency, high-throughput data exchange. This refactoring introduced a dual communication pattern: a structured master-worker protocol for control synchronization and a peer-to-peer mesh for direct agent exchanges. By combining optimized graph partitioning with Aeron-based messaging, the

enhanced MASS system reduced the communication latency and increased agent migration efficiency across large, distributed graphs.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Table Number Page

LISTINGS

Chapter 1

INTRODUCTION

As data continues to grow not only in volume but also in structural complexity, modern analytics systems face increasing pressure to support computation over richly connected, non-linear data structures. While traditional big data streaming frameworks such as Apache Spark [1], Hadoop MapReduce [2], and Apache Storm [3] have proven effective for processing large-scale textual or tabular datasets, they are inherently limited when it comes to analyzing complex and mutable data structures like trees, graphs, or spatial networks. These systems are designed to split data into independent chunks for parallel computation, which disrupts the continuity of relationships that are essential in structured datasets. Consequently, such platforms are not well-suited for applications where locality, connectivity, and dynamic structure are critical—for example, in social networks, transportation systems, or biological interaction graphs.

To address these limitations, agent-based computing has emerged as a powerful paradigm. Unlike traditional data-parallel models, agents can be embedded directly into a distributed data structure and move, communicate, and evolve in response to local conditions. This makes agents particularly well-suited for graph-based applications, where computational logic often depends on the topology of the network and localized interactions between neighboring vertices. In the context of distributed graph databases, agents offer a flexible and programmable means of traversing, modifying, and querying graphs while preserving structural integrity and supporting asynchronous execution.

To explore and harness these advantages, the Multi-Agent Spatial Simulation (MASS) Java framework has been adopted as the foundation for building an Agent-Based Graph Database system [4]. MASS is designed to support the creation and execution of lightweight agents over a spatially distributed memory model. In this system, agents can operate on graph vertices, execute OpenCypher queries [5], and facilitate concurrent multi-user

access to shared graph data through the newly introduced Distributed Shared Graph (DSG) abstraction. This architecture has shown promising results in improving graph creation and query performance compared to conventional distributed key-value stores like Hazelcast [6].

However, despite its conceptual strengths, the current implementation reveals several performance challenges. First, graph traversal is slowed down by a round-robin vertex distribution strategy that leads to poor data locality, increasing inter-node communication overhead [7]. Second, MASS uses a traditional TCP-based communication system that introduces critical performance bottlenecks, such as the overhead of establishing and maintaining connections between all participating nodes and thread management overhead during an agent's migration to remote nodes.

This capstone project aims to address these limitations by optimizing vertex placement across nodes and improving execution performance in the MASS-based distributed graph system. Project goals include:

- Conducting a literature survey on efficient graph partitioning techniques.
- Implementing graph partitioning algorithms to distribute vertex places across the cluster system.
- Replacing the traditional TCP communication model with a high-performance UDP-based protocol.
- Evaluating performance benchmarks post-implementation to assess improvements in execution efficiency.

Chapter 2

BACKGROUND AND CHALLENGES

This chapter provides an overview of the underlying MASS architecture, its application to agent-based graph processing, and the associated performance limitations. Finally, the key challenges associated with current design—particularly in terms of vertex placement and communication inefficiencies—are discussed. These challenges serve as the motivation for the optimization efforts undertaken in this project.

2.1 Multi-Agent Spatial Simulation Architecture

Multi-Agent Spatial Simulation (MASS) is a parallel computing library designed for distributed memory systems, enabling scalable agent-based computations across a cluster of nodes. It consists of two primary abstractions: Places and Agents. Places form a multi-dimensional distributed array where each element, or `"place"`, resides on a specific node and is globally indexed. These places serve as the data environment that can store state and interact with other places. Agents represent the active computational entities that reside within places, performing analysis and migrating across the distributed array to simulate dynamic behavior and process graph-based data.

MASS achieves parallelism through multithreaded processes running on each cluster node, with the number of threads typically aligned to the available CPU cores. MASS deploys processes over cluster system using Java Secure Channel (JSch), their communication is handled over a network with Transmission Control Protocol (TCP) sockets. As illustrated in Figure 2.1, the MASS architecture distributes both computation and data across the cluster, allowing agents to operate concurrently over large-scale spatial and graph-based datasets.

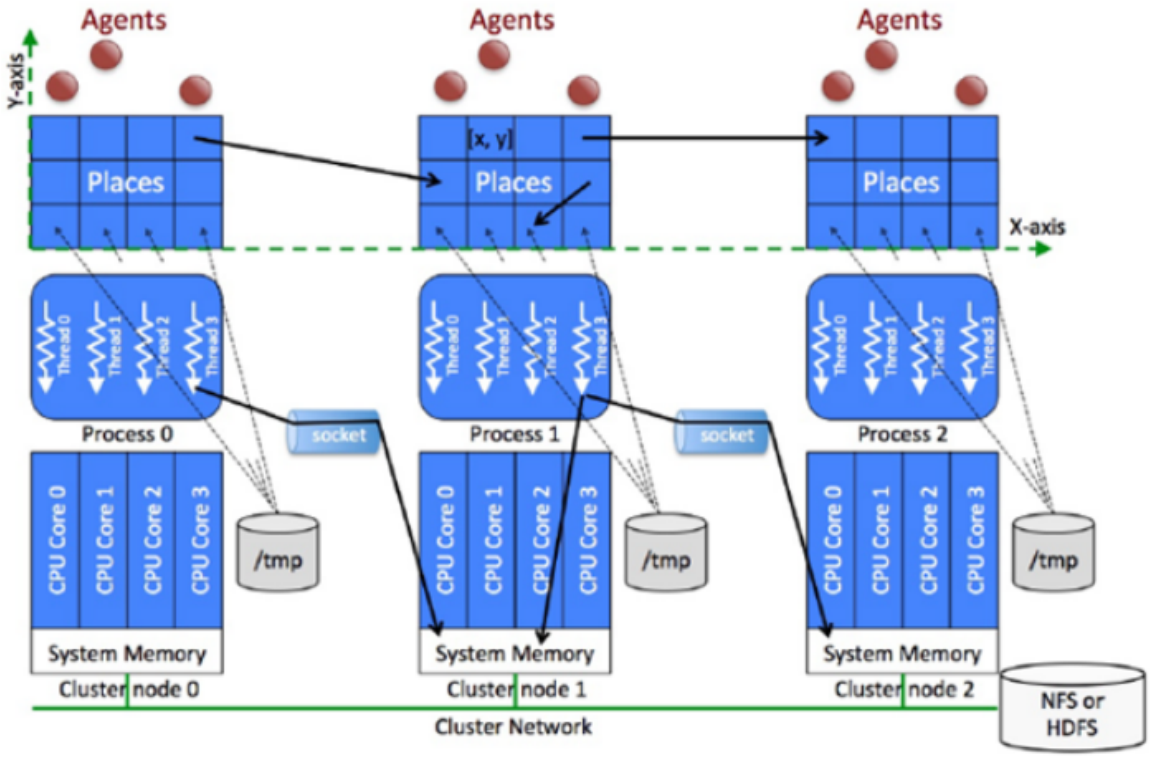

Figure 2.1: MASS library architecture [8]

2.2 Agent-based Graphs in MASS

MASS has already supported a graph data structure called GraphPlaces that extends from the Places class. GraphPlaces store graphs in adjacency list format, which means for each vertex we store its corresponding attributes and its neighbors as a list. When a graph is inserted into a GraphPlaces it will be distributed in the cluster by storing information of vertices in different computing nodes. The vertices in GraphPlaces are represented by VertexPlace objects, and VertexPlace extends from the MASS Place class. As mentioned previously, it stores the list of outgoing edges and the information about itself.

To accommodate real-world graph database requirements—such as labeled vertices with arbitrary types—the system decouples the logical vertex ID (e.g., names like `"Ann"` or `"Jim"`) from its underlying numeric representation required for agent migration and distributed

indexing. This is achieved through a distributed HashMap, which maps object-based identifiers (e.g., strings) to sequential integer IDs. This map is accessible from all nodes and ensures global consistency in vertex referencing.

Once an integer ID is assigned, the system uses a modulo-based hash function to determine which node will store the vertex:

$$\text{Computing Node ID} = \text{Vertex Integer ID mod Number of Computing Nodes} \quad \text{(Eq. 2.1)}$$

Within each node, the vertex is stored in a `Vector<VertexPlace>`, and its position is computed using:

$$\text{Index in Vector} = \frac{\text{Vertex Integer ID}}{\text{Number of Computing Nodes}} \quad \text{(Eq. 2.2)}$$

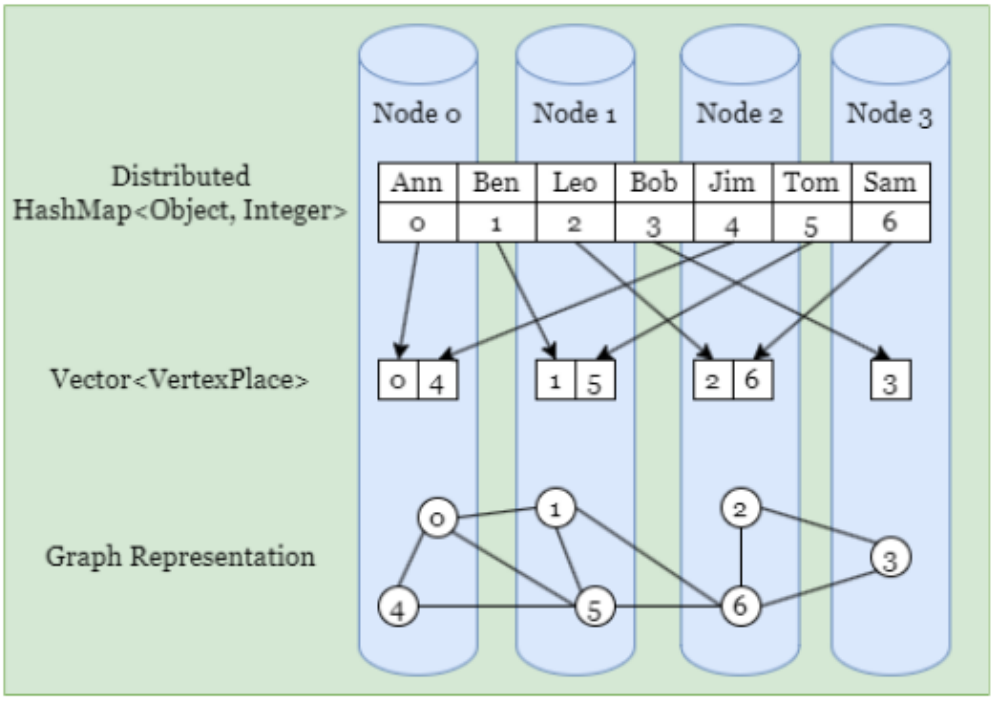

Figure 2.2: Current vertex place distribution [5]

2.3 Technical Challenges

As illustrated in Figure 2.2, vertices of the graph are distributed across computing nodes using a round-robin strategy using simple hash function given in Eq. 2.1, which disregards the underlying graph topology. As a result, adjacent or highly connected vertices are often placed on different nodes, leading to poor data locality. This fragmentation forces agents performing traversal or query operations to frequently communicate across nodes, significantly increasing network overhead and degrading performance.

The TCP-based communication architecture in MASS creates a significant performance bottleneck due to its reliance on a full mesh topology where every node maintains persistent socket connections to all other nodes, coupled with synchronous, blocking I/O operations. This approach forces computation to halt during network transfers, with threads remaining idle while waiting for message transmission and acknowledgment. The problem compounds during agent migration events, where the Java's inefficient serialization process executes synchronously on the main thread, and the system's thread-per-destination model spawns pair of threads (one dedicated reader, one dedicated writer) per connection by separating read and write operations, ensuring nodes could always receive incoming data even while transmitting, thus preventing the deadlock. TCP's inherent head-of-line blocking prevents independent agent migrations from proceeding if earlier packets are delayed, while its congestion control mechanisms can throttle throughput during the bursty communication patterns typical in multi-agent simulations. The sequential processing of migration requests prevents effective parallelization, resulting in severely degraded performance as the number of computing nodes and migrating agents increases.

These performance limitations motivate the work undertaken in this project. The system is enhanced by implementing graph-aware vertex placement strategies and replacing TCP communication with a high-performance UDP-based messaging protocol. The effectiveness of these changes is evaluated using benchmark comparisons with the previous TCP-based, round-robin architecture.

Chapter 3

RELATED WORK

3.1 Graph Distribution Strategies

Graph partitioning is a fundamental technique in distributed graph processing, aiming to divide a graph into smaller subgraphs to optimize computational efficiency and minimize inter-node communication. The strategy for distributing graph data across computing nodes significantly impacts performance and scalability in distributed systems. Several algorithms and approaches have been developed to address this challenge [9].

The Kernighan–Lin algorithm [10] is a heuristic method that seeks to minimize the edge cut between partitions by iteratively swapping vertex pairs to improve partition quality. Although it can produce high-quality partitions, its computational complexity makes it less suitable for very large graphs.

Community detection-based partitioning, such as the Louvain method, optimizes modularity to uncover densely connected subgraphs, effectively identifying communities within the graph. This approach is particularly beneficial for social networks and other graphs where community structures are prominent.

Facebook's Balanced Label Propagation (BLP) algorithm introduces a scalable solution for partitioning massive graphs. BLP combines the simplicity of label propagation with constraints to ensure balanced partition sizes. In practice, BLP has demonstrated significant improvements in reducing inter-node communication and balancing computational loads across partitions [11].

Hash-based partitioning assigns vertices to partitions based on a hash function applied to their identifiers, ensuring an even distribution. However, it doesn't consider the graph's topology, potentially leading to increased inter-node communication. Hazelcast employs a hash-based partitioning strategy to distribute data across its cluster nodes [6]. When a data entry is added, Hazelcast serializes the key into a byte array, applies a hashing

algorithm, and then calculates the partition ID by taking the modulo of the hash result with the total number of partitions. By default, Hazelcast uses 271 partitions, ensuring a fine-grained distribution of data. This approach allows for even data distribution and facilitates scalability, as adding new nodes results in minimal data movement due to the consistent hashing mechanism.

Multilevel graph partitioning algorithms represent a sophisticated approach to graph partitioning that operates in three distinct phases: coarsening, initial partitioning, and uncoarsening with refinement. This multilevel approach significantly reduces computational complexity while maintaining high-quality partitioning results [12].

Based on the advantages of multilevel partitioning, METIS was selected as the graph partitioning library for this implementation due to its proven multilevel k-way partitioning algorithms and scalability for large-scale graphs [13]. METIS provides the necessary balance between partition quality and computational efficiency required for distributed graph processing in MASS Java.

3.2 Communication Systems in Distributed Graph Databases

Distributed graph processing frameworks implement various communication architectures to support efficient graph traversal and agent movement across cluster nodes. Apache Giraph, built on Hadoop's infrastructure, employs a Bulk Synchronous Parallel (BSP) computation model where vertex-centric processing occurs in synchronized Super-steps [14]. Its master-worker architecture coordinates global barriers and message passing between iterations. However, as Ching et al. demonstrate, Giraph's reliance on Hadoop's underlying MapReduce communication patterns introduces substantial synchronization overhead during each super step, especially when compared to MASS's more flexible agent migration capabilities that can occur asynchronously [15].

Neo4j's clustering architecture implements a causal consistency model with a leader-follower replication scheme for distributed deployments [16]. Communication between cluster members occurs through a dedicated Raft protocol implementation for cluster coordination and topology management. Raft is designed to provide atomic communication guarantees, ensuring that operations are applied in a consistent and fault-tolerant manner

across the cluster. However, these strong consistency and ordering guarantees come at the cost of increased communication latency and limited throughput. Despite being one of the most mature graph database systems, Neo4j's communication layer is primarily optimized for transactional workloads rather than continuous agent-based simulation scenarios. According to Webber, Neo4j's communication patterns emphasize safety guarantees over raw performance, making it less suitable for high-throughput, low-latency requirements similar to those in MASS where agents frequently migrate between computational nodes [17].

JanusGraph (formerly Titan) addresses distributed graph processing through a modular storage and indexing backend approach, supporting various underlying datastores like Cassandra and HBase [18]. Its communication architecture delegates much of the inter-node coordination to these backend systems, with additional messaging for transaction coordination across the cluster. While this approach offers flexibility, Vaquero et al. note that it introduces additional network hops and serialization costs during distributed traversals [19]. These overheads become particularly pronounced in workloads resembling MASS's agent movement patterns, where entities frequently traverse partition boundaries.

We adopted Aeron's UDP-based communication architecture for MASS to address critical performance limitations identified in the original TCP implementation, as detailed in our technical challenges analysis. MASS's agent-centric computational model requires high-frequency, low-latency communication to support continuous agent migration across distributed computational nodes. Our initial design explored a broadcast messaging approach to coordinate agent migrations cluster-wide, but our analysis revealed exponential growth in serialization and deserialization overhead as graph size increased, with communication overhead due to large message payloads containing extensive agent state information. To address these scalability concerns, we transitioned to a direct node-to-node messaging paradigm using Aeron's UDP transport, which eliminates broadcast overhead by enabling each computational node to synchronously transmit only the specific agents destined for target machines. Aeron's lockless ring buffers and memory-mapped communication architecture provide sub-millisecond latency through optimized memory access patterns while bypassing kernel overhead, perfectly aligning with MASS's requirements for near real-time agent coordination across distributed graph storage systems.

Chapter 4

ENHANCED VERTEX DISTRIBUTION STRATEGY

This chapter presents two enhanced vertex distribution implementations designed to improve graph partitioning performance in distributed systems. First, we explore a Hazelcast-inspired partitioning technique that employs fixed-partition hash-based distribution to achieve balanced workload allocation and simplified cluster management. Subsequently, we describe an advanced implementation using the METIS graph partitioning library [13], which aims to minimize edge cuts while maintaining balanced vertex distribution across the cluster through sophisticated multilevel graph partitioning algorithms.

4.1 Hazelcast Inspired Partitioning Strategy

The Hazelcast-inspired partitioning strategy implements a hash-based vertex distribution approach that prioritizes simplicity and balanced workload distribution over graph structure awareness. Unlike complex multilevel partitioning algorithms like METIS, this strategy employs a fixed-partition model with deterministic hash functions to ensure uniform vertex distribution across cluster nodes. The approach guarantees consistent and predictable partitioning behavior while maintaining scalability and load balance in distributed graph processing environments.

4.1.1 Design

The Hazelcast-inspired partitioning design introduces a fundamentally different approach to graph distribution that emphasizes predictability and balanced resource utilization over structural optimization. The core design philosophy centers on creating a fixed number of logical partitions that remain constant regardless of graph size or cluster configuration changes.

The partitioning strategy operates through a two-level hierarchical mapping system.

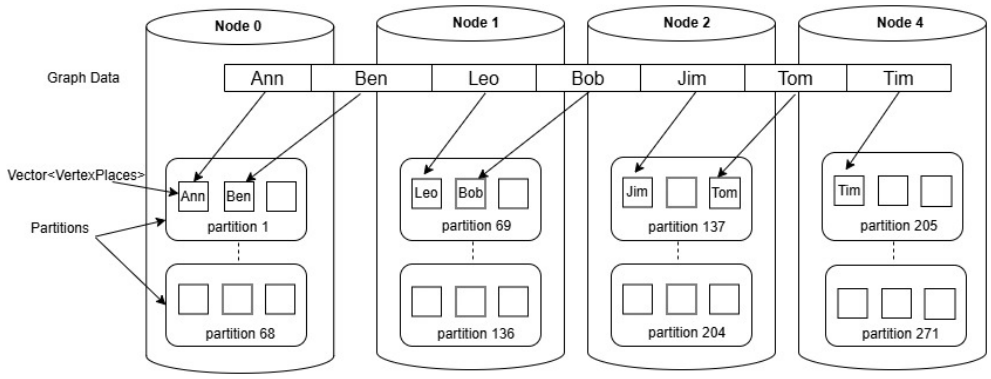

Figure 4.1: Hazelcast inspired graph partitioning

At the first level, vertices are assigned to logical partitions using a simple modular hash function:

$$\text{Partition ID} = \text{Vertex ID mod TOTAL\_PARTITIONS (271)} \tag{4.1}$$

This deterministic assignment ensures uniform distribution across all partitions while maintaining computational simplicity. The fixed partition count of 271 was chosen to provide sufficient granularity for load balancing while avoiding excessive overhead in partition management operations.

At the second level, these logical partitions are dynamically mapped to physical cluster nodes using:

$$\text{Node ID} = \text{Partition ID mod Number of Nodes} \tag{4.2}$$

This dual-level approach provides several architectural advantages. First, it decouples the partitioning logic from the cluster topology, enabling seamless scaling operations where nodes can be added or removed without requiring complete repartitioning. Second, it guarantees deterministic behavior across all cluster nodes for vertex distribution.

The design addresses the fundamental challenge of inter-partition communication by accepting that hash-based distribution will inevitably create cross-partition edges. Rather than attempting to minimize these connections, the system is designed to handle them

efficiently through optimized remote reference resolution mechanisms and comprehensive partition-to-node mapping tables maintained on each node.

The workflow architecture ensures that each computing node can independently determine vertex ownership and partition assignments without requiring centralized coordination.

4.1.2 Implementation

The implementation of the Hazelcast-inspired partitioning strategy integrates seamlessly with the existing MASS GraphPlaces infrastructure through a streamlined vertex distribution pipeline. The core implementation focuses on three fundamental operations: partition initialization, vertex assignment, and distributed graph construction.

The partition initialization process begins with the `initializePartitions()` method, which establishes the foundational mapping structures:

```
1 private void initializePartitions() {
2     int clusterSize = MASS.getSystemSize();
3     for (int i = 0; i < TOTAL_PARTITIONS; i++) {
4         partitions.put(i, new ArrayList<>());
5         partitionToNodeMap.put(i, i % clusterSize);
6     }
7 }
```

Listing 4.1: Partition Initialization

Lines 2 retrieve the current cluster size from the MASS framework. Lines 3-6 iterate through all TOTAL_PARTITIONS (271 partitions), where line 4 initializes each partition as an empty ArrayList container, and line 5 establishes the deterministic partition-to-node mapping using modular arithmetic. This method creates 271 empty partition containers and establishes the partition-to-node mapping using modular arithmetic.

The vertex assignment implementation leverages the hash-based distribution through the `addVertexOnNode()` method:

```
1 public boolean addVertexOnNode(int nodeID, int partition, int vertexID,
      Object vertexInitParams) {
2     // Calculate partition ownership
```

```
3       int targetPartition = vertexID % TOTAL_PARTITIONS;
4       int targetNode = targetPartition % MASS.getSystemSize();
5
6       // Validate node assignment
7       if (nodeID != targetNode) {
8           return addRemoteVertexUpdated(targetNode, targetPartition, vertexID,
                vertexInitParams);
9       }
10
11      // Local vertex creation
12      partitions.computeIfAbsent(targetPartition, k -> new ArrayList<>());
13      VertexPlace vertexPlace = objectFactory.getInstance(getClassName(),
            vertexInitParams);
14      vertexPlace.setIndex(new int[]{vertexID});
15      partitions.get(targetPartition).add(vertexPlace);
16
17      return true;
18  }
```

Listing 4.2: Vertex Assignment Implementation

Lines 3-4 perform the core hash-based partition assignment calculations, where line 3 determines the target partition using vertex ID modulo total partitions, and line 4 calculates the responsible node using partition ID modulo cluster size. Lines 7-9 handle remote delegation scenarios by validating node ownership and forwarding vertex creation requests to the appropriate remote node when the current node is not responsible for the calculated partition. Lines 12-15 manage local vertex creation by ensuring the target partition exists, instantiating a new VertexPlace object with the provided parameters, setting the vertex index, and adding it to the appropriate partition container. The implementation performs real-time hash calculations to determine correct vertex placement and automatically delegates to remote nodes when necessary. This approach ensures that vertices are consistently placed according to the hash-based distribution regardless of which node initiates the vertex creation operation.

The Hazelcast-inspired partitioning strategy provides a robust and scalable alternative

to complex graph-aware partitioning approaches. While it may not achieve the optimal edge cut minimization of sophisticated algorithms like METIS, it offers significant advantages in terms of implementation simplicity, deterministic behavior, and balanced workload distribution. The fixed-partition hash-based approach ensures consistent performance characteristics and simplified cluster management operations, making it particularly suitable for dynamic distributed environments where predictability and operational simplicity are prioritized over structural optimization.

4.2 METIS Multilevel Graph Partitioning Strategy

Given an undirected graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$, the goal is to partition V into k disjoint subsets $V_1, V_2, \ldots, V_k$ such that:

$$\bigcup_{i=1}^{k} V_i = V \quad \text{and} \quad V_i \cap V_j = \emptyset \quad \text{for } i \neq j.$$

The objective is to minimize the edge cut:

$$\Gamma(V_1, V_2, \ldots, V_k) = \sum_{i=1}^{k} \sum_{j=i+1}^{k} \left| \{(u, v) \in E : u \in V_i, v \in V_j\} \right|, \tag{4.3}$$

subject to the balance constraint:

$$|V_i| \leq (1 + \epsilon) \left\lceil \frac{|V|}{k} \right\rceil \quad \text{for some imbalance factor } \epsilon. \tag{4.4}$$

To solve this problem efficiently, METIS employs a multilevel graph partitioning strategy that consists of three main phases: coarsening, initial partitioning, and uncoarsening with refinement. This process is illustrated in Figure 4.2.

Coarsening Phase

The algorithm begins by creating a hierarchy of progressively coarser graphs $G_0, G_1, \ldots, G_m$, where G_0 is the original input graph. This is done by merging vertex pairs using heavy-edge or random matching. Heavy-edge matching prioritizes vertices connected by higher-weight edges, which helps preserve the graph's structure. The coarsening continues until the number of vertices falls below a predefined threshold, resulting in a small graph G_m that is suitable for efficient direct partitioning.

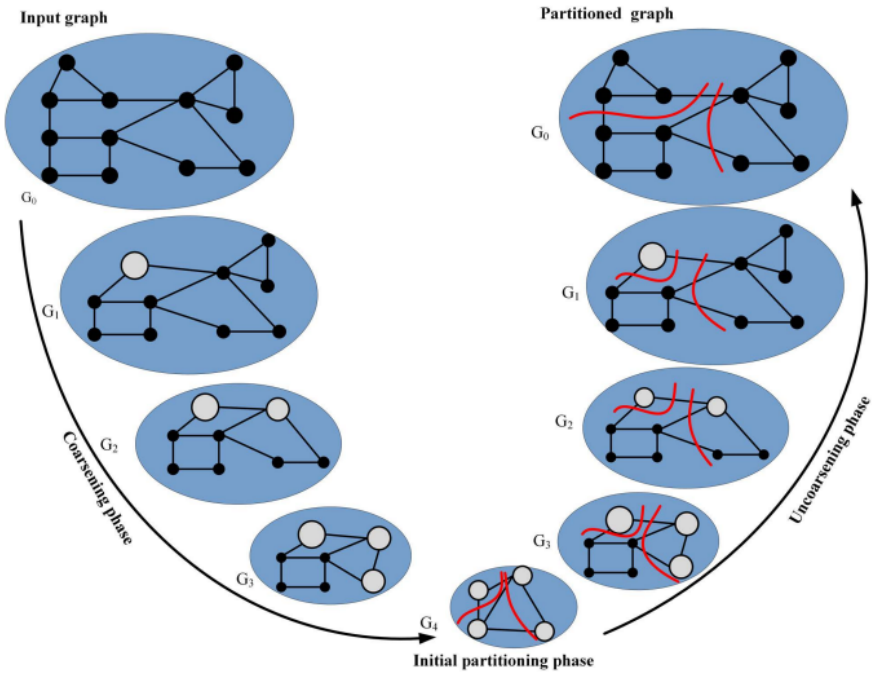

Figure 4.2: Multilevel graph partitioning phases in METIS [12]

Initial Partitioning Phase

At the coarsest level G_m, METIS computes a high-quality k-way partition using either recursive bisection or a direct method based on heuristics like the Kernighan–Lin algorithm. Because G_m is significantly smaller, these methods are computationally feasible and effective.

Uncoarsening and Refinement Phase

The computed partition on G_m is then projected back through each level of the hierarchy to G_0. At each stage, the partition is refined using local optimization techniques such as the Kernighan–Lin algorithm. This refinement ensures that the final partition minimizes edge cuts and satisfies balance constraints. By combining global structure from the coarsening phase with local optimization during refinement, METIS achieves both high-quality and scalable partitioning.

4.2.1 Design

The integration of METIS partitioning into the MASS GraphPlaces infrastructure required a fundamental redesign of how graph data is distributed and managed across the cluster. Our approach leverages METIS's sophisticated partitioning capabilities while maintaining compatibility with MASS's existing distributed computing framework and message-passing infrastructure.

The computational workflow initiates with the master node performing comprehensive graph data ingestion from standardized formats including UW-Bothell proprietary DSL and Property Graph specifications. Subsequently, the master node executes graph preprocessing operations that transform the input adjacency representation into Compressed Sparse Row (CSR) format [20], which serves as the required input specification for METIS partitioning algorithms. The core partitioning phase involves invoking METIS through Java Native Interface (JNI) [21] mechanisms to execute multilevel k-way graph partitioning algorithms.

The fundamental challenge in this distributed implementation concerns the non-deterministic nature of METIS optimization heuristics, which can produce varying partition assignments due to internal randomization and local optimization strategies. To solve this issue, we implemented a seeding mechanism where a fixed seed value is passed during each METIS execution. This guarantees deterministic vertex-to-partition mappings across all computational nodes.

The second challenge involved developing an efficient mechanism for each node to identify, read, and store only the graph partitions relevant to that node. Rather than loading the complete partitioned graph and discarding unused portions, each node selectively loads and retains only the vertices assigned to its partition. This approach minimizes memory usage while maintaining the ability to resolve inter-partition edge references.

Each node maintains a partition-to-vertex mapping table that enables quick determination of vertex ownership without requiring inter-node communication during the loading process. The workflow concludes with the master node transmitting `LoadGraph()` messages to all worker nodes, triggering parallel initialization of graph processing pipelines.

For simplicity, Figure C.2 shows each worker node's `"Load Graph"` operation as an

abstraction of the full processing pipeline. In practice, each node performs equivalent operations locally, including file parsing, CSR conversion, METIS execution, mapping generation, and data indexing. This architecture promotes horizontal scalability while ensuring deterministic and consistent behavior across distributed resources.

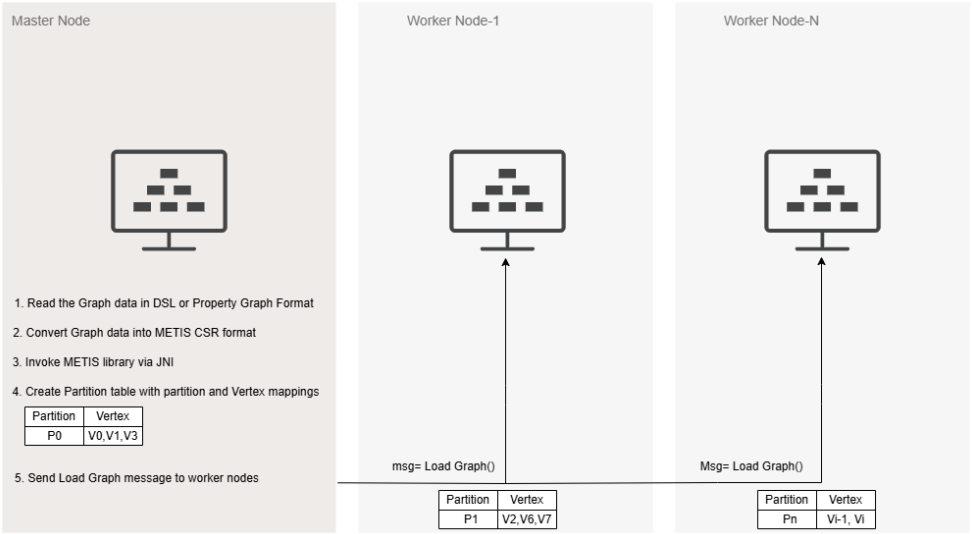

Figure 4.3: Workflow: vertex distribution with METIS graph partitioning

4.2.2 Implementation

This implementation provides graph partitioning capabilities using the METIS graph partitioning library through Java Native Interface (JNI). The code converts graph data from UW-Bothell proprietary DSL format to METIS-compatible Compressed Sparse Row (CSR) format and performs k-way partitioning.

Algorithm 1 is used for partitioning the graph using METIS. The function begins by initializing the current computing node's rank which is an auto-incremented integer process id assigned to each computing node during MASS initialization (master computing node always initialized with process id = 0), a vertex counter, and a local index counter (lines 1–2). It reads the graph data from a DSL-formatted input file, parsing each line to construct a full adjacency list where each vertex is mapped to a list of its weighted neighbors. This

representation is essential for later transformation into a format compatible with METIS (line 2).

Algorithm 1: Load Local Graph Data with METIS Partitioning

Input: DSL graph file path

Output: Number of loaded vertices

```
1  Initialize rank = auto-incremented integer process id, vertex count, and local index
     counter;
2  Read graph from DSL file and build adjacency list;
3  Convert adjacency list to METIS CSR format;
4  if number of computing nodes == 1 then
5  |   Assign all vertices to partition 0;
6  else
7  |   Use METIS JNI to partition graph;
8  |   Map vertex → partition;
9  Assign local index to vertices for current computing node;
10 foreach local vertex do
11 |   Add vertex to current computing node;
12 |   Add corresponding edges to current computing node;
13 |   Increment vertex count;
14 return vertex count;
```

Following the data ingestion, the method converts the graph into METIS's expected Compressed Sparse Row (CSR) format. This involves creating the xadj array that marks the starting index of neighbors for each vertex, and the adjncy array that stores adjacency relationships in a flattened structure (line 3). These arrays together provide the structural blueprint for METIS's graph partitioning logic.

Depending on the number of partitions that is equal to the total computing nodes, the algorithm assigns all vertices to a single partition (in the single node mode) or invokes the JNI-bound METIS partitioning engine for k-way partitioning (lines 4–8). In the METIS path, the graph is partitioned using multilevel heuristics, and the resulting vertex-to-partition assignments are recorded in a lookup map for efficient access.

Once partitioning is complete, the current computing node identifies which vertices belong to its partition and assigns them local indices to support subsequent distributed graph operations (line 9). Finally, the method iterates through each locally owned vertex to insert it into the MASS node and attach the corresponding weighted edges based on the earlier adjacency list (lines 10–13). The function concludes by returning the total number of vertices successfully loaded by the node. The same algorithm runs on each computing node in the MASS and creates a graph in parallel across the cluster system.

The complete distributed graph partitioning mechanism, message passing infrastructure, vertex management, and error handling code is provided in the Appendix A for comprehensive reference, including the coordination of partitioning results across multiple compute nodes and the integration with the broader MASS simulation framework.

Chapter 5

AERON-BASED UDP COMMUNICATION SYSTEM

This chapter introduces the Aeron-based [22] communication model adopted in the MASS framework, designed to replace traditional TCP sockets with a high-performance, lock-free UDP messaging system. Aeron provides reliable message delivery through UDP by utilizing embedded media drivers, zero-copy semantics, and stream ordering guarantees, which make it an ideal candidate for latency-sensitive distributed simulations.

5.1 Design

Aeron is a high-performance, ultra-low latency messaging library providing efficient publish-subscribe messaging over UDP with lock-free data structures, zero-copy operations, and CPU cache-aware memory layouts. It has an embedded media driver architecture that eliminates inter-process communication overhead, automatic message fragmentation and reassembly, built-in flow control with back-pressure handling, and support for both multicast and unicast communication patterns with guaranteed message ordering within streams. MASS adopts Aeron to replace TCP-based communication due to its superior performance characteristics for distributed simulation workloads, particularly the ability to handle high-frequency agent migrations and place data exchanges without the latency penalties and connection overhead associated with traditional socket-based messaging systems.

The architecture employs a four-phase initialization sequence that establishes both communication patterns simultaneously. As illustrated in Figure 5.1, Phase 1 begins with `MASS.init()` creating the Master Node (PID=0) as the central coordination point for cluster management and global state synchronization. Phase 2 extends the system through SSH-based remote process launching, where the master node establishes `MProcess` instances on each compute node, creating the distributed worker infrastructure necessary for parallel simulation execution.

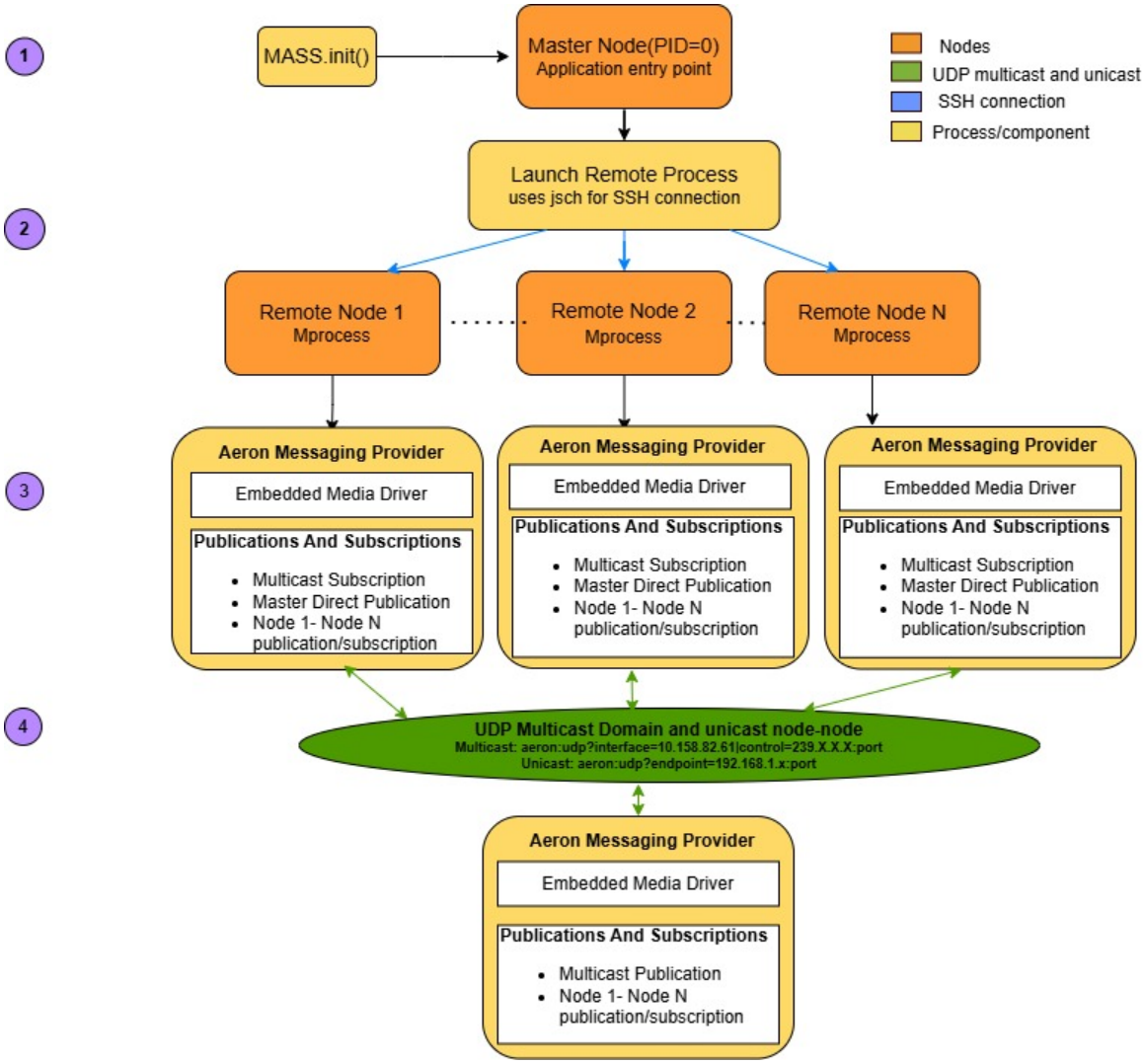

Figure 5.1: Four-phase Aeron initialization and communication patterns in MASS.

Phase 3 implements the Master-Worker communication pattern through dedicated Aeron streams connecting the master node to each worker. This hierarchical structure provides centralized control for operations requiring global coordination including graph initialization commands, barrier synchronization points and other messages. The master node maintains direct publication channels to each worker while workers establish subscription channels for receiving commands and publication channels for sending acknowledgments and status updates back to the master.

Phase 4 establishes the Peer-to-Peer communication pattern through the `ExchangeHelper` class, creating a fully connected mesh network where any node can communicate directly with any other node without routing through the master. This eliminates the master node as a potential bottleneck during high-volume operations such as agent migrations. Each node maintains dedicated Aeron publications to every other node in the cluster, enabling concurrent multi-directional data flows that scale independently of cluster size.

The dual-pattern approach leverages Aeron's embedded media driver architecture, where each compute node runs its own media driver instance within the process boundary. This design eliminates inter-process communication overhead while providing isolation between application logic and network operations. The embedded drivers handle message fragmentation, flow control, and network interface management transparently, allowing the MASS framework to focus on simulation logic rather than low-level networking concerns.

Communication channels utilize UDP multicast for cluster-wide broadcasts and UDP unicast for direct node-to-node messaging. Multicast channels handle global coordination messages including barrier synchronization, graph loading, and cluster shutdown commands. Unicast channels manage specific inter-node communications such as agent migration data. This channel separation optimizes network utilization by avoiding unnecessary message delivery while maintaining message ordering guarantees within each communication stream.

Despite these advantages, we encountered a challenge in the agent migration workflow when sending messages larger than 16 megabytes. Aeron's UDP transport faces inherent limitations where individual messages cannot exceed the configured term buffer size (typically 16MB) due to memory allocation constraints and network MTU restrictions, while Java object serialization of large agent collections can easily produce payloads exceeding hundreds of megabytes.

We implemented a custom fragmentation logic as shown in Figure 5.2. When a serialized message exceeds the `MAX_MESSAGE_SIZE_BYTES` threshold, the system fragments it into smaller chunks with a configurable size (leaving overhead space for headers) and assigns a unique fragmentation ID for message reconstruction. The fragmentation process begins by sending metadata containing the total fragment count, original message size, and fragmentation identifier, followed by sequential transmission of numbered fragments containing the

actual data payload.

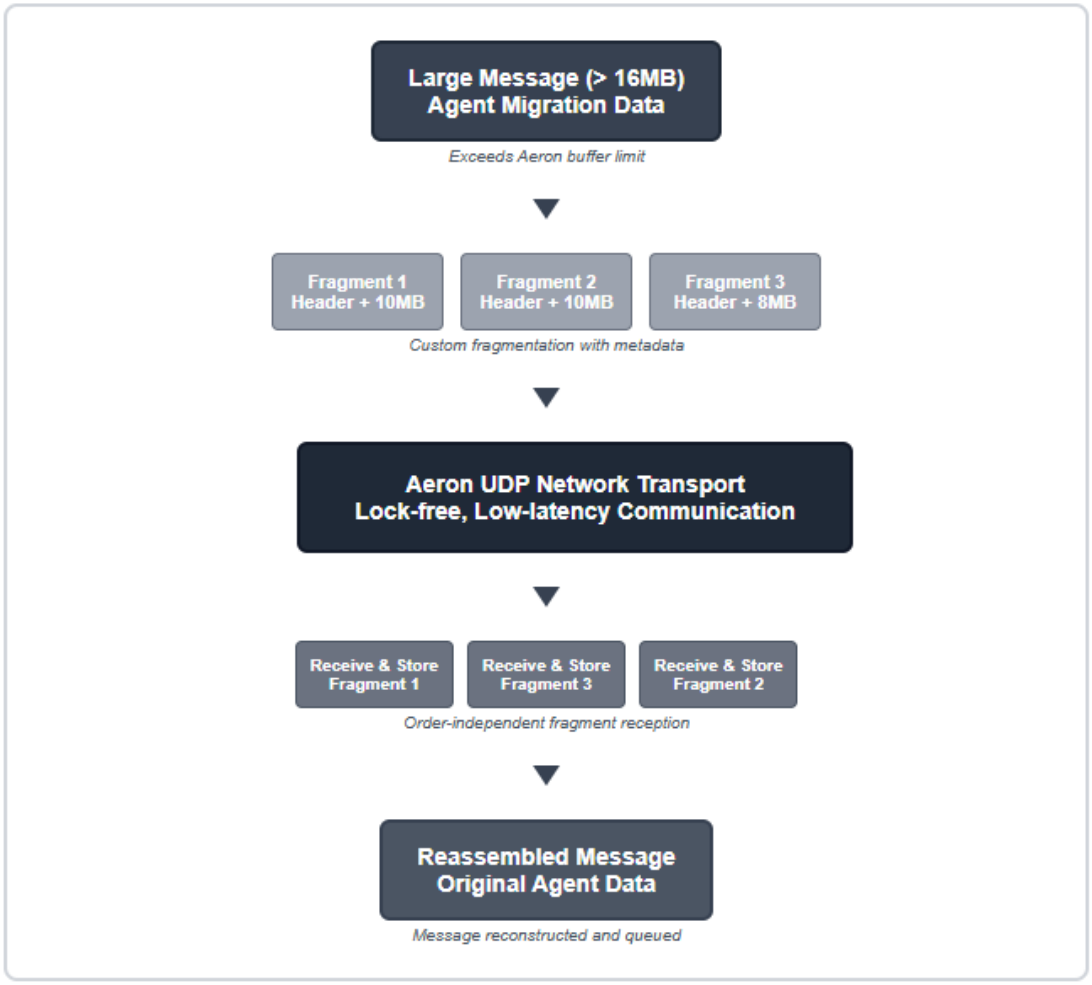

Figure 5.2: Custom fragmentation and reassembly mechanism for agent migration

At the receiving end, the system maintains concurrent hash maps to store fragmented metadata and accumulate incoming fragments, automatically reconstructing the original message once all fragments are received and placing the complete message in the appropriate rank-specific message queue.

5.2 Implementation

The following listing demonstrates how peer-to-peer communication is initialized using Aeron's embedded media driver and unicast subscriptions.

```
1   // Create embedded media driver with minimal footprint
2   mediaDriver = MediaDriver.launch(new MediaDriver.Context()
3           .dirDeleteOnStart(true)
4           .dirDeleteOnShutdown(true)
5           .termBufferSparseFile(true)
6           .threadingMode(io.aeron.driver.ThreadingMode.DEDICATED)
7           .aeronDirectoryName(CommonContext.getAeronDirectoryName() +
                AERON_DIRECTORY_SUFFIX));
8
9   // Create Aeron instance
10  aeron = Aeron.connect(new Aeron.Context()
11          .aeronDirectoryName(mediaDriver.aeronDirectoryName()));
12
13  // Initialize message queues for each rank
14  for (int i = 0; i < size; i++) {
15      if (i != rank) { // No need for queue to self
16          messageQueues.put(i, new LinkedBlockingQueue<>());
17      }
18  }
19
20  // Set up subscription to receive messages first
21  nodePubs = new Publication[size];
22  String localEndpoint = hosts.get(rank) + ":" + port;
23  String subChannel = AERON_URL_PREFIX + localEndpoint + AERON_URL_SUFFIX;
24  exchangeSubscription = aeron.addSubscription(subChannel,
        EXCHANGE_STREAM_ID);
```

Listing 5.1: Peer-to-Peer Aeron Initialization for Node Messaging

Listing 5.1 initializes the core components for Aeron-based peer-to-peer communication within a distributed MASS system. Lines 1-7 launch an embedded Aeron media driver with a minimal memory and file system footprint, ensuring dedicated threading and auto-

matic cleanup of temporary directories for each MASS run. Lines 9-11 establish an `Aeron` instance connected to the directory created by the media driver, forming the foundation for message exchange. In lines 13-18, the system sets up dedicated message queues using `LinkedBlockingQueue` for all other nodes in the cluster (excluding self), allowing each node to store and process incoming messages independently based on the sender's rank. This design ensures thread-safe, rank-specific message handling across the peer network. Line 21 initializes a `Publication[]` array, enabling each node to maintain a separate outgoing publication channel to every other peer. Finally, lines 22–24 define and register a unique Aeron subscription for the local node, allowing it to receive peer-to-peer messages via a specific UDP unicast channel constructed using the host's IP address and port. Our initial architectural approach employed a broadcast messaging system utilizing multicast channels, with each computing node subscribing to the multicast channel rather than implementing peer-to-peer messaging over unicast channels. However, empirical evaluation revealed that this broadcast design exhibited exponential growth in serialization and deserialization overhead, compounded by increasingly large message payloads as graph size scaled. To mitigate these performance bottlenecks, we transitioned to a direct node-to-node messaging paradigm leveraging Aeron's UDP transport protocol, which facilitates synchronous transmission of targeted agents to their designated destination nodes while eliminating broadcast-related overhead. This architectural foundation establishes a scalable, bidirectional communication infrastructure across the entire cluster topology..

The complete implementation, including peer-to-peer exchange helpers, fragmentation utilities, and error handling routines, is provided in the Appendix B.

Chapter 6

EVALUATION

This chapter presents the experimental evaluation of our enhanced MASS-based distributed graph processing system. We assess the effectiveness of our two core contributions: (1) Hazelcast inspired and METIS-based vertex distribution strategy, and (2) an Aeron-based high-performance communication infrastructure. We evaluated the system performance in comparison to the traditional Round Robin vertex distribution and TCP-based message passing. Our evaluation includes agent migration workflows and graph traversal operations through a benchmark Triangle Counting application.

6.1 Performance Benchmarking Methodology

To evaluate the improvements introduced in this work, we use the Triangle Counting application previously used by Chris Ma from DSLab. This benchmark allows us to measure both computational efficiency and inter-node communication overhead, as it inherently involves significant agent migration and graph traversal.

6.1.1 Triangle Counting Application

The Triangle Counting application (see Figure 6.1) uses MASS agents that traverse a graph in a manner that ensures each triangle is counted exactly once. Initially, agents are spawned at every `VertexPlace`. They traverse along edges to neighboring vertices, restricted to only follow edges directed to a lower vertex ID to avoid duplicate counting. After two such migrations, an agent checks whether it can return to its original vertex via an edge. If successful, a triangle is identified, and the counter is incremented. This application highlights the role of localized agent migration and its dependency on communication and partitioning efficiency.

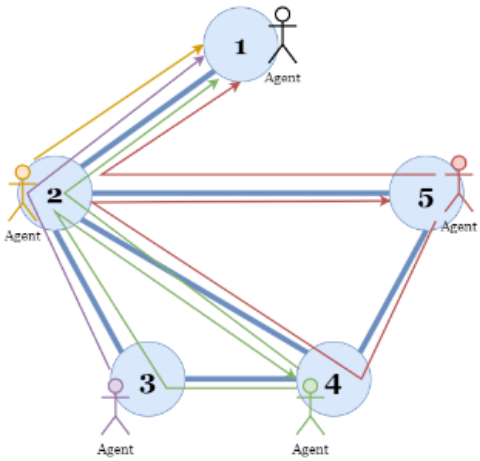

Figure 6.1: Triangle counting with agent-based migration [5]

6.2 Environment Setup

The experiments were conducted on the HERMES cluster at the University of Washington Bothell, comprising 24 computing nodes. We identified optimal performance at 8 nodes. Table 6.1 details the hardware used.

Table 6.1: HERMES Cluster environment.

# Computing Nodes	# Logical CPU Cores	CPU Model	Memory
3	4	Intel Xeon 5150 @ 2.66 GHz	16GB
4	8	Intel Xeon E5410 @ 2.33 GHz	16GB
1	4	Intel Xeon 5220R @ 2.20 GHz	16GB

6.2.1 Benchmark Settings

Table 6.2 shows the graph datasets used for the triangle counting application. Experiments were run using 1 to 8 computing nodes. All values represent averages over three runs.

Table 6.2: Graph data used for Triangle Counting application.

Number of Vertices	Number of Edges	Number of Triangles
3000	293804	192146
5000	467400	197440
10000	989990	200053

6.3 Performance Comparison: Round Robin vs Hazelcast Inspired Partitioning

We evaluated the performance of the Triangle counting application on multiple computing nodes with different graph sizes ranging from 1K to 20K vertices. As illustrated in Figure 6.2, the performance benchmark comparing Hazelcast-inspired partitioning against Round Robin strategy for triangle counting across 8 computing nodes reveals that Hazelcast consistently underperforms, with execution times being 3-5% higher across all graph sizes.

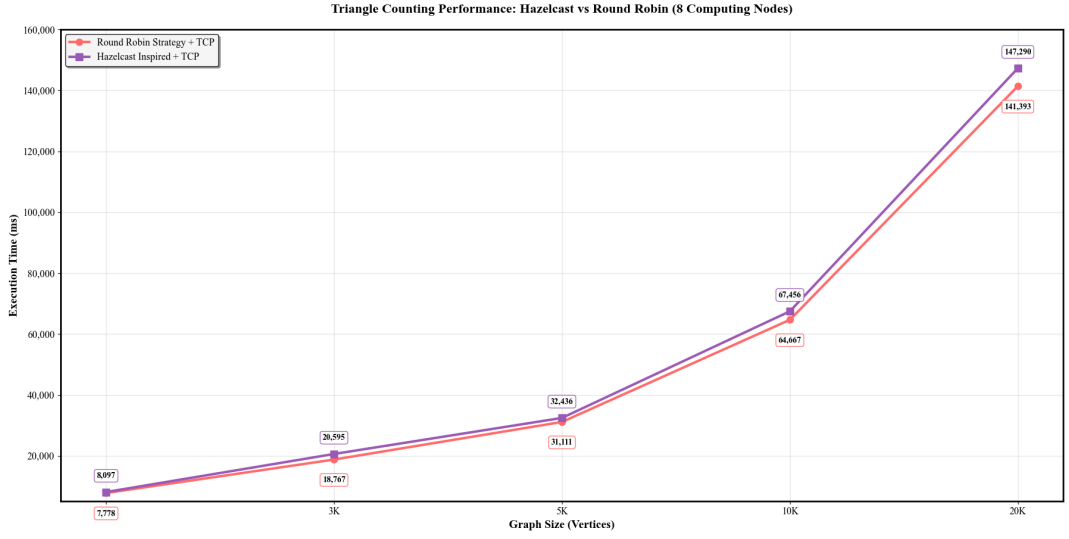

Figure 6.2: Triangle counting execution performance with hazelcast inspired partitioning

This pattern persists across larger graphs, with the 20K vertex case showing Round

Robin at 141,393 ms compared to Hazelcast's 147,290 ms (4.2% overhead). This performance degradation occurs because, despite its sophisticated fixed-partition architecture with 271 logical partitions, Hazelcast-inspired partitioning remains fundamentally structure-agnostic like Round Robin, distributing vertices based on hash functions rather than graph connectivity patterns. The additional overhead is coming from partition management, hash calculations, and partition-to-node mapping operations, while providing no benefits for triangle counting operations that require extensive neighbor-to-neighbor traversal patterns.

6.4 Performance Comparison: Round Robin vs METIS partitioning

To evaluate our graph partitioning enhancement, we compared the Round Robin distribution strategy with METIS-based partitioning in triangle counting application. Figure 6.3 illustrates the execution time for the Triangle Counting application across varying graph sizes (1K, 3K, 5K, 10K, and 20K vertices) using 8 computing nodes.

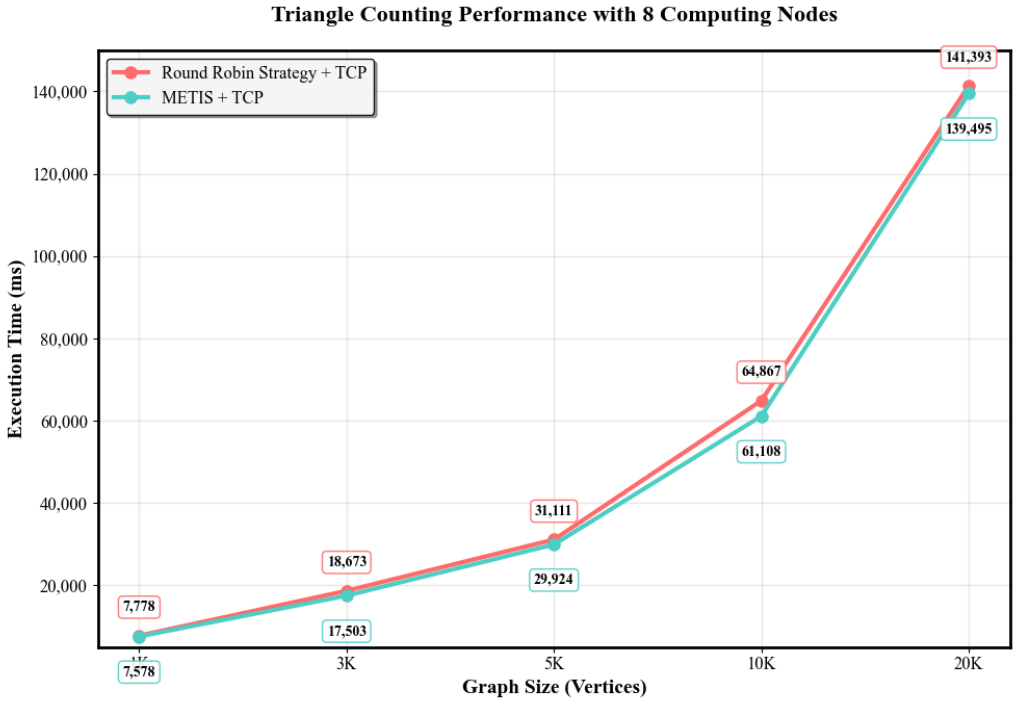

Figure 6.3: Triangle counting execution performance with METIS partition strategy

Although METIS consistently outperforms the Round Robin approach across all graph sizes by reducing execution times—primarily due to improved data locality—its performance gains were not as significant as initially anticipated. Ideally, METIS's capability to co-locate highly connected vertices on the same machine should have resulted in substantially reduced inter-node communication and improved runtime.

To investigate this discrepancy, we examined the most computationally intensive operation in MASS: agent migration. As shown in Figure 6.4, we compare the number of agents processed locally versus remotely under both METIS and Round Robin partitioning strategies. In the Triangle Counting application, agents migrate along edges and are expected to operate locally when neighboring vertices are assigned to the same node. However, if the destination vertex resides on a different node, agents need to be migrated remote machine to perform further triangle counting computation.

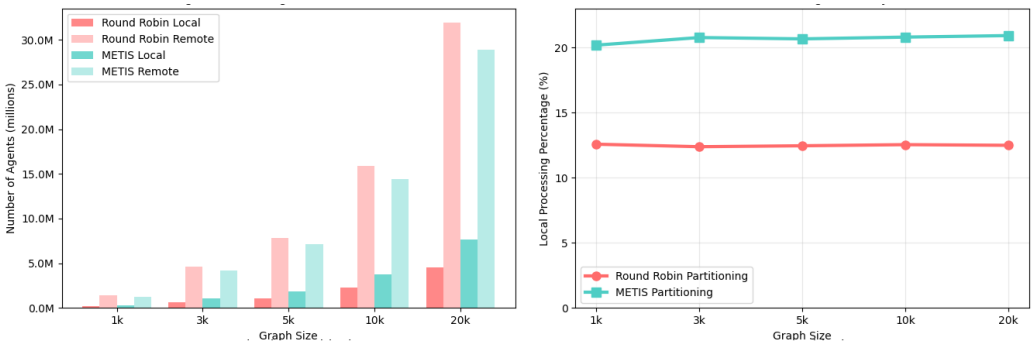

Figure 6.4: Local vs remote agents processing in triangle counting

The right subplot quantifies the percentage of agents processed locally, where METIS consistently achieves a higher locality percentage (21%) compared to Round Robin (12%). This confirms that METIS improves vertex placement and agent locality, but also highlights the potential for further enhancement in partitioning strategies or agent migration mechanisms to maximize intra-node computation and minimize communication overhead.

6.5 Performance Comparison: TCP vs Aeron's UDP communication

To evaluate the impact of Aeron-based communication compared to traditional TCP sockets, we conducted same triangle counting experiments using three different graph sizes (3K, 5K, and 10K vertices) across varying numbers of computing nodes (2, 4, and 8). The results, shown in Figures 6.5, 6.6, and 6.7, consistently indicate that the METIS with Aeron configuration outperforms the Round Robin with TCP baseline with 2 and 4 computing nodes.

However, as the number of nodes increases to 8, the performance gap narrows, with execution times converging to TCP execution time. This trend suggests diminishing returns from Aeron at higher node counts. Aeron's performance begins to degrade as the number of computing nodes increases, primarily due to the $\mathcal{O}(n^2)$ explosion of unidirectional publications it requires for peer-to-peer communication. For instance, at 8 nodes, Aeron maintains 56 unidirectional publications compared to TCP's 28 bidirectional socket connections.

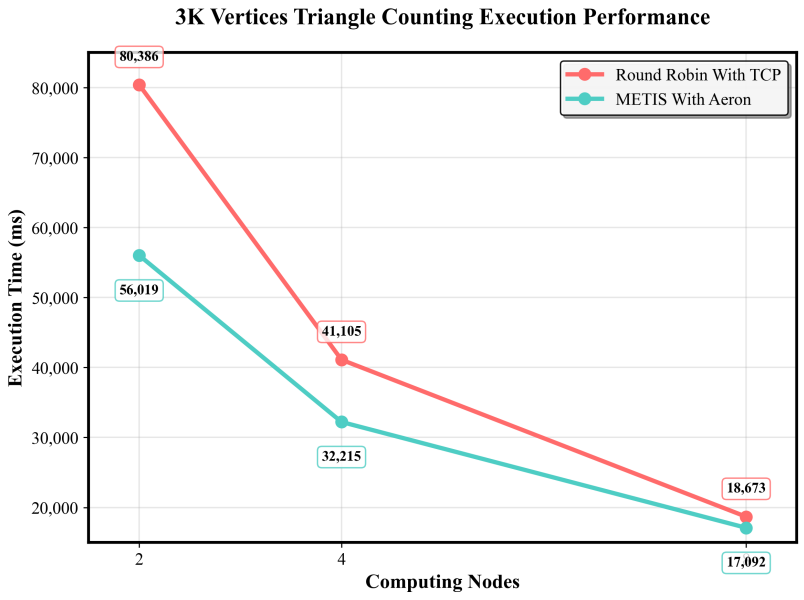

Figure 6.5: Triangle Counting Time for Graph Size 3K

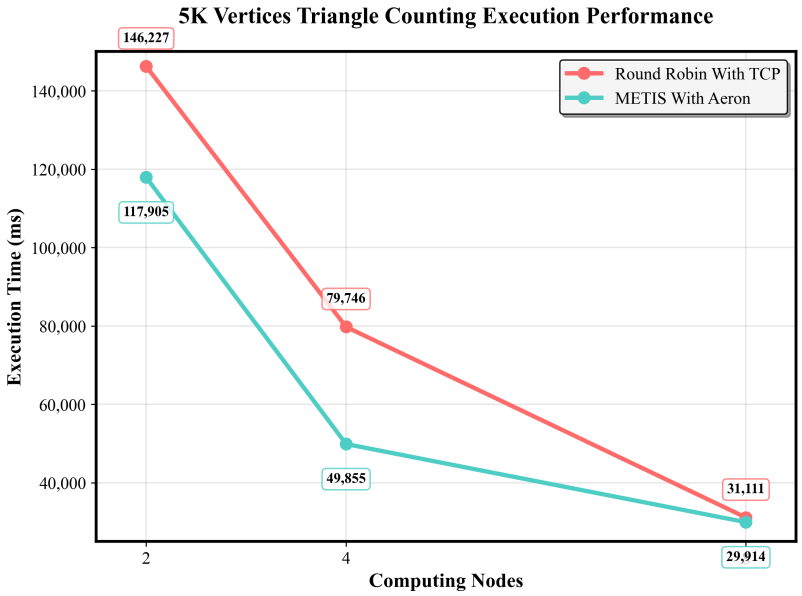

Figure 6.6: Triangle Counting Time for Graph Size 5K

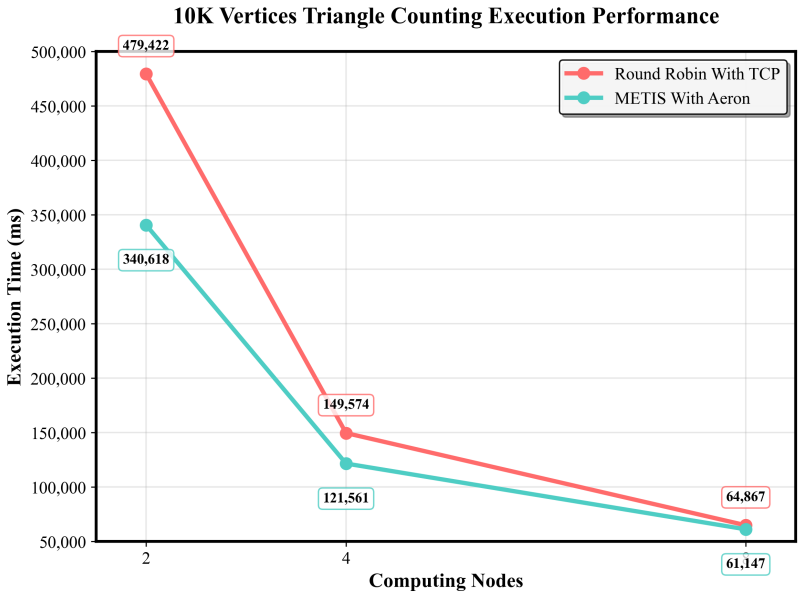

Figure 6.7: Triangle Counting Time for Graph Size 10K

Chapter 7

CONCLUSION

This chapter concludes the paper by summarizing the key contributions and outlining potential directions for future work.

7.1 Summary of Contributions

In this project, we improved the performance of the MASS (Multi-Agent Spatial Simulation) system by adding two major components: METIS for better graph partitioning and Aeron for faster communication between nodes. METIS helps divide the graph more intelligently by grouping connected vertices together, which reduces the need for agents to move across different nodes. At the same time, we replaced the traditional TCP-based communication system with Aeron, which uses UDP and provides faster message delivery with lower latency.

We compared our new METIS partitioning and Aeron-based system against the older Round Robin and TCP setup. Our experiments showed that the new setup performs better when communication involves less computing nodes. However, we also observed some limitations at larger cluster sizes due to increased communication overhead, which opens up opportunities for future improvements.

7.2 Limitations & Future Work

The primary limitation of the current implementation lies in the $\mathcal{O}(n^2)$ communication scaling bottleneck, which arises from the agent migration patterns that require every node to potentially communicate with every other node. This is compounded by the creation of $n \times (n-1)$ unidirectional publication channels in Aeron's peer-to-peer communication model, which can overwhelm system resources. For example, with 8 computing nodes, the system must manage over 56 active publication channels, leading to excessive memory consumption,

frequent garbage collection triggered by message serialization, and increased CPU overhead. Moreover, the synchronous nature of MASS's agent migration barriers forces all nodes to wait for the slowest migration event to complete, which reduces the effectiveness of Aeron's low-latency transport and negates its performance advantages as the cluster size increases beyond 8 to 16 nodes.

Based on these observed limitations, we propose several directions for future improvement. First, the peer-to-peer publication model of Aeron can be optimized from $\mathcal{O}(n^2)$ to $\mathcal{O}(\log n)$ by implementing shared communication channels and tree- or ring-based routing strategies. This would reduce the number of direct connections while maintaining efficient and scalable connectivity across the distributed cluster. Second, the agent migration and lifecycle management in MASS can be further optimized to reduce unnecessary communication and master-worker synchronization overhead.

BIBLIOGRAPHY

[1] Apache Spark. *Apache Spark*. `https://spark.apache.org`. Accessed on May 28, 2025.

[2] Apache Hadoop. *Apache Hadoop*. Accessed: 2025-05-28. URL: `https://hadoop.apache.org/`.

[3] Apache Storm. *Apache Storm*. Accessed: 2025-05-28. URL: `https://storm.apache.org/`.

[4] Harshit Rajvaidya. "An Agent-Based Graph Database". Unpublished manuscript, DSLab, University of Washington Bothell, 2024.

[5] Yuan Ma. "An Implementation of Multi-User Distributed Shared Graph". Unpublished manuscript, DSLab, University of Washington Bothell, 2025.

[6] Hazelcast, Inc. *Data Partitioning and Replication — Hazelcast Documentation*. Accessed: 2025-05-28. URL: `https://docs.hazelcast.com/hazelcast/5.4/architecture/data-partitioning`.

[7] Michelle Dea. "Development of an Agent-Based Database Benchmarking Dataset". Unpublished manuscript, DSLab, University of Washington Bothell, 2025.

[8] J. Timothy Chuang and Munehiro Fukuda. "A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems". In: *Proceedings of the 16th IEEE International Conference on Computational Science and Engineering (CSE 2013)*. Sydney, Australia: IEEE, Dec. 2013, pp. 143–150.

[9] Hlib Mykhailenko, Fabrice Huet, and Giovanni Neglia. "Comparison of Edge Partitioners for Graph Processing". In: *Proceedings of the International Conference on Computational Science and Computational Intelligence (CSCI)*. Las Vegas, USA, 2016. URL: `https://hal.science/hal-01401338`.

[10] Brian W. Kernighan and Shen Lin. *Kernighan–Lin Algorithm*. Accessed: 2025-05-28. URL: https://en.wikipedia.org/wiki/Kernighan%E2%80%93Lin_algorithm.

[11] Jure Ugander and Lars Backstrom. "Balanced Label Propagation for Partitioning Massive Graphs". In: *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM '13)*. 2013, pp. 507–516. DOI: 10.1145/2433396.2433461. URL: https://doi.org/10.1145/2433396.2433461.

[12] T. A. Ayall et al. "Graph Computing Systems and Partitioning Techniques: A Survey". In: *IEEE Access* 10 (2022), pp. 118523–118550. DOI: 10.1109/ACCESS.2022.3219422.

[13] George Karypis and Vipin Kumar. *METIS*. Accessed: 2025-05-28. URL: https://en.wikipedia.org/wiki/METIS.

[14] Grzegorz Malewicz et al. "Pregel: A System for Large-Scale Graph Processing". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2010, pp. 135–146.

[15] Avery Ching et al. "One Trillion Edges: Graph Processing at Facebook-Scale". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1804–1815.

[16] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. 2nd. O'Reilly Media, 2015.

[17] Jim Webber. "A Programmatic Introduction to Neo4j". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. 2012, pp. 217–218. DOI: 10.1145/2384716.2384777. URL: https://doi.org/10.1145/2384716.2384777.

[18] JanusGraph Project. *JanusGraph Documentation*. Accessed: 2025-05-28. URL: https://docs.janusgraph.org/.

[19] Luis M. Vaquero et al. *Adaptive Partitioning for Large-Scale Dynamic Graphs*. Accessed: 2025-05-28. URL: https://dl.acm.org/doi/10.5555/2672596.2672664.

[20] *METIS/manual/manual.pdf at master · KarypisLab/METIS · GitHub*. URL: https://github.com/KarypisLab/METIS/blob/master/manual/manual.pdf (visited on 05/28/2025).

[21] Oracle Corporation. *Java Native Interface Specification: Introduction*. Accessed: 2025-05-28. URL: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html.

[22] Martin Thompson and Todd Montgomery. *Aeron: High-Throughput Low-Latency Messaging*. Accessed: 2025-05-28. URL: https://github.com/real-logic/aeron.

Appendix A

GRAPH PARTITIONING CODE

```
// loadDSLFileUsingMETIS loads graph data using METIS for partitioning.
  public void loadDSLFileUsingMETIS(String filePath) throws IOException,
      FileNotFoundException {
      MASS.getLogger().debug("Loading DSL File and partitioning with METIS
          ");
      // Send partitions to each node
      MASS.getRemoteNodes().forEach(node -> node.sendMessage(new Message(
              Message.ACTION_TYPE.MAINTENANCE_LOAD_DSL_FILE_USING_METIS,
              getHandle(),
              // Send all nodes the current vertex offset and path of file
                  to load.
              new Object[] { filePath })));

      // Handle local vertices on this node
      int vertexCount = loadLocalGraphData(filePath);

      // Wait for all other nodes to complete and aggregate vertex counts
      for (MNode node : MASS.getRemoteNodes()) {
          Message msg = node.receiveMessage();
          if (msg.getAction() != Message.ACTION_TYPE.ACK) {
              MASS.getLogger().debug("Failed to load DSL file on node " +
                  node.getPid());
          }
          vertexCount += msg.getAgentPopulation();
      }

      // Update index counter based on total vertices processed
      nextVertexID += vertexCount;

      // Broadcast update to ensure global consistency
```

```
27          if (sharedPlaceName != null) {
28              Message m = new Message(Message.ACTION_TYPE.
                    DSG_NEXT_VERTEXID_UPDATE, MASSBase.getUserName(),
                    nextVertexID,
29                      sharedPlaceName);
30              localMessaging.sendMessage(m);
31          }
32
33      }
```

Listing A.1: METIS Graph partitioning code

Appendix B

AERON COMMUNICATION IMPLEMENTATION

The following listing provides the complete implementation of Aeron-based communication in MASS, including ExchangeHelper mesh setup, message queue handling, and fragmentation utilities.

```
/**
 * ExchangeHelper contains methods for communicating with peers using Aeron
    UDP
 * transport
 */
public class ExchangeHelper {

    // Stream IDs for different message types
    private static final int EXCHANGE_STREAM_ID = 2001;
    private static final int ACK_STREAM_ID = 2002;

    // Maximum message size (16MB)
    private static final int MAX_MESSAGE_SIZE_BYTES = 16777216;

    // Operation timeout
    private static final long OPERATION_TIMEOUT_MS = 300000;

    // Idle strategy for spin waiting
    private final IdleStrategy idleStrategy = new SleepingIdleStrategy();

    // Clock for timeouts
    private final EpochClock clock = new SystemEpochClock();

    // Buffer for outgoing messages
    private final UnsafeBuffer sendBuffer = new UnsafeBuffer(
            BufferUtil.allocateDirectAligned(MAX_MESSAGE_SIZE_BYTES, 64));
```

```
26
27      // Aeron publications for each remote node
28      private Publication[] nodePubs;
29
30      // Subscription for receiving messages
31      private Subscription exchangeSubscription;
32
33      // Message queues for each rank
34      private ConcurrentHashMap<Integer, LinkedBlockingQueue<Message>>
            messageQueues = new ConcurrentHashMap<>();
35
36      // Subscriber running flag
37      private AtomicBoolean running = new AtomicBoolean(true);
38
39      // Reference to Aeron
40      private Aeron aeron;
41
42      // Reference to MediaDriver
43      private MediaDriver mediaDriver;
44
45      // Thread for message reception
46      private Thread receiverThread;
47
48      private static final String AERON_URL_PREFIX = "aeron:udp?endpoint=";
49      private static final String AERON_URL_SUFFIX = "|term-length=128m|mtu
            =65504|socket-sndbuf=16m|socket-rcvbuf=16m";
50      private static final String AERON_FLOW_CONTROL_STRATEGY = "|fc=min";
51      private static final String AERON_DIRECTORY_SUFFIX = "
            _mass_agent_migration";
52
53      // Fragmentation related fields
54      private final AtomicInteger messageIdCounter = new AtomicInteger(0);
55      private static final int MAX_FRAGMENT_SIZE = 60000; // this is max
            fragment size we can send using mtu=65504 which is max default mtu
56      private final Map<Integer, byte[][]> fragmentsReceived = new
            ConcurrentHashMap<>();
```

```
57      private final Map<Integer, MessageFragmentMetadata> fragmentMetadata =
            new ConcurrentHashMap<>();
58
59      /**
60       * Establish connections to peers for agent migration
61       *
62       * @param size  The number of nodes in the cluster
63       * @param rank  The rank ID of this node
64       * @param hosts The peer hosts with which to establish connections
65       * @param port  The port number used for communication
66       */
67      public void establishConnection(int size, int rank, Vector<String> hosts
            , int port) {
68          MASS.getLogger().debug("Initializing optimized Aeron connections for
                agent migration");
69
70          // Create embedded media driver with minimal footprint
71          mediaDriver = MediaDriver.launch(new MediaDriver.Context()
72                  .dirDeleteOnStart(true)
73                  .dirDeleteOnShutdown(true)
74                  .termBufferSparseFile(true)
75                  .threadingMode(io.aeron.driver.ThreadingMode.DEDICATED)
76                  .aeronDirectoryName(CommonContext.getAeronDirectoryName() +
                        AERON_DIRECTORY_SUFFIX));
77
78          // Create Aeron instance
79          aeron = Aeron.connect(new Aeron.Context()
80                  .aeronDirectoryName(mediaDriver.aeronDirectoryName()));
81
82          if (aeron == null) {
83              MASS.getLogger().error("Aeron instance not available - MASS
                    messaging not properly initialized");
84              System.exit(-1);
85          }
86
87          // Initialize message queues for each rank
```

```
88          for (int i = 0; i < size; i++) {
89              if (i != rank) { // No need for queue to self
90                  messageQueues.put(i, new LinkedBlockingQueue<>());
91              }
92          }
93
94          // Set up subscription to receive messages first
95          nodePubs = new Publication[size];
96          String localEndpoint = hosts.get(rank) + ":" + port;
97          String subChannel = AERON_URL_PREFIX + localEndpoint +
                AERON_URL_SUFFIX;
98          exchangeSubscription = aeron.addSubscription(subChannel,
                EXCHANGE_STREAM_ID);
99
100         // Give subscription time to initialize before creating publications
101         try {
102             Thread.sleep(1000);
103         } catch (InterruptedException e) {
104             Thread.currentThread().interrupt();
105         }
106
107         // Create publications to all other nodes
108         for (int i = 0; i < size; i++) {
109             if (i == rank)
110                 continue; // Skip self
111
112             String endpoint = hosts.get(i) + ":" + port;
113             String channel = AERON_URL_PREFIX + endpoint + AERON_URL_SUFFIX;
114
115             nodePubs[i] = aeron.addPublication(channel, EXCHANGE_STREAM_ID);
116
117             // Implement retry logic for connection
118             int maxRetries = 5;
119             boolean connected = false;
120
121             for (int retry = 0; retry < maxRetries; retry++) {
```

```
122                 MASS.getLogger().debug("Attempt " + retry + 1 + " to connect
                        to node " + i + "...");
123
124                 // Check for connection with a shorter timeout per attempt
125                 long deadline = clock.time() + 5000; // 5 seconds per
                        attempt
126                 while (!nodePubs[i].isConnected() && clock.time() < deadline
                        ) {
127                     idleStrategy.idle();
128                 }
129
130                 if (nodePubs[i].isConnected()) {
131                     connected = true;
132                     MASS.getLogger().debug("Connected to node " + i + " on
                            attempt " + retry + 1);
133                     break;
134                 }
135
136                 if (retry < maxRetries - 1) {
137                     MASS.getLogger()
138                             .debug("Connection attempt " + retry + 1 + "  to
                                    node " + i + " failed, retrying...");
139                     try {
140                         // Implement exponential backoff
141                         Thread.sleep(1000 * (retry + 1));
142                     } catch (InterruptedException e) {
143                         Thread.currentThread().interrupt();
144                     }
145                 }
146             }
147
148             if (!connected) {
149                 MASS.getLogger().debug("Could not connect to node " + i + "
                        at " + endpoint + "after " + maxRetries
150                             + "attempts. Continuing anyway.");
151                 // Not exiting the application, just continuing with other
```

```
                    nodes
152             }
153         }
154
155         // Start receiver thread
156         receiverThread = new Thread(this::receiveMessages);
157         receiverThread.setDaemon(true);
158         receiverThread.start();
159
160         printAeronConfiguration();
161         MASS.getLogger().debug("Aeron connections established for
                ExchangeHelper");
162     }
163 }
```

Listing B.1: Aeron Communication Logic

Appendix C

HOW TO RUN PROGRAMS

This appendix provides step-by-step instructions for running MASS Java applications using two different graph partitioning approaches along with Aeron's UDP communication implementation.

C.1 Hazelcast-inspired Partitioning (Git Branch - aahire/dsg-improvements)

This branch implements distributed graph processing with Hazelcast-inspired partitioning for improved performance.

Prerequisites

- Java 8 or higher
- Maven 3.6+
- Git

C.1.1 Setup and Execution

Step 1: Build MASS Java Core

```
# Clone and checkout the DSG improvements branch
git clone https://bitbucket.org/mass_library_developers/mass_java_core.git
cd mass_java_core
git checkout aahire/dsg-improvements

# Build the core library
mvn -DskipTests clean package install
```

Step 2: Build Triangle Counting Application

```
# Navigate to MASS Java Applications repository
cd ../mass_java_appl/Graphs/TriangleCounting
git checkout develop

# Build the application
mvn clean package

# Copy application JAR to root folder
cp target/TriangleCounting-1.0.0-RELEASE.jar .
```

Step 3: Prepare Configuration Create a `nodes.xml` file in your execution directory:

```
<nodes>
        <node>
                <master>true</master>
                <hostname>hermes1.uwb.edu</hostname>
                <masshome><your application checkedout dir>/mass_java_appl/
                    Graphs/TriangleCounting</masshome>
                <username><your username></username>
                <privatekey>/home/NETID/<your username>/.ssh/id_rsa</
                    privatekey>
                <port><port number></port>
        </node>
        <node>
                <hostname>hermes2.uwb.edu</hostname>
               <username><your username></username>
                <masshome><your application checkedout dir>/mass_java_appl/
                    Graphs/TriangleCounting</masshome>
                <port><port number></port>
               <privatekey>/home/NETID/<your username>/.ssh/id_rsa</
                   privatekey>
        </node>
        <! -- Add additional remote nodes as needed -- >
    </nodes>
```

Step 4: Run Triangle Counting Application

```
java -jar TriangleCounting-1.0.0-RELEASE.jar \
<Path to sample DSL graph files>/graph_w.csv
```

C.1.2 Expected Output

The application will output triangle count statistics and performance metrics using the Hazelcast-style distributed partitioning approach.

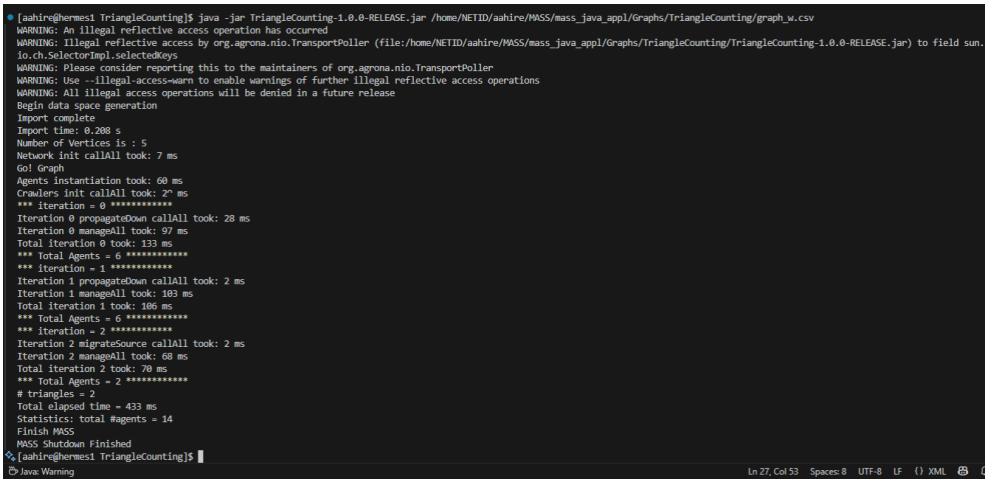

Figure C.1: Workflow: Hazelcast inspired graph partitioning output

C.2 METIS Graph Partitioning and Aeron Implementation (Git Branch: aahire-metis-implementation)

This branch uses METIS library for sophisticated graph partitioning to optimize load distribution across compute nodes and aeron communication implementation details.

Setup and Execution

Step 1: Install GKLib Library

```
# Clone and build GKLib
git clone https://github.com/KarypisLab/GKlib.git
cd GKlib

# Configure with proper C source definition to avoid type errors
make config cc=gcc prefix=~/local/gklib CFLAGS="-D_POSIX_C_SOURCE=199309L"
make
make install
```

Update GKLib with proper linking:

```
# Clean any previous builds
rm -rf /home/NETID/<your username>/GKlib/build/Linux-x86_64
mkdir -p /home/NETID/<your username>/GKlib/build/Linux-x86_64
cd /home/NETID/<your username>/GKlib/build/Linux-x86_64

# Configure with CMake
cmake -D BUILD_SHARED_LIBS=ON \
      -DCMAKE_C_FLAGS="-D_POSIX_C_SOURCE=199309L" \
      -DCMAKE_INSTALL_PREFIX=/home/NETID/<your username>/local/gklib \
      /home/NETID/<your username>/GKlib

make
make install
```

Step 2: Install METIS Library

```
# Clone METIS repository
git clone https://github.com/KarypisLab/METIS.git
cd /home/NETID/<your username>/METIS

# Modify CMakeLists.txt to link with GKlib
sed -i '/add_library(metis ${METIS_LIBRARY_TYPE} ${metis_sources})/ s/$/\
    ntarget_link_libraries(metis GKlib)/' libmetis/CMakeLists.txt

# Configure rpath for proper library linking
sed -i '/^CONFIG_FLAGS \?= / s,$, -DCMAKE_INSTALL_RPATH=/home/NETID/aahire/
    local/gklib/lib -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=ON,' Makefile

# Build and install METIS
make config shared=1 gklib_path=/home/NETID/<your username>/local/gklib
    prefix=/home/NETID/<your username>/local/metis
make
make install
```

Step 3: Build MASS Java Core

```
# Navigate to MASS core directory and checkout METIS branch
cd <your checkout folder>/mass_java_core
git checkout aahire-metis-implementation

# Build the core library with METIS support
mvn clean package install
```

Step 4: Build Triangle Counting Application

```
# Build the application for METIS implementation
cd ../mass_java_appl/Graphs/TriangleCounting
git checkout aahire-metis-implementation
mvn clean package
```

Step 5: Create Execution Script Create a shell script `run_triangle_counting_metis.sh`:

```
#!/bin/bash

# Set the paths for dynamic libraries
export LD_LIBRARY_PATH=/home/NETID/<your username>/local/metis/lib:/home/
    NETID/<your username>/METIS/build/libmetis

# Run the Java application with METIS partitioning
java -cp "/home/NETID/<your username>/MASS:/home/NETID/<your username>/local
    /libs/*" \
    -Djava.library.path=/home/NETID/<your username>/MASS/mass_java_core/
        target/native \
    -jar TriangleCounting-1.0.0-ASE.jar \
    /home/NETID/<your username>/MASS/mass_java_appl/Graphs/TriangleCounting/
        graph_w.csv
```

Step 6: Execute the Application

```
# Make the script executable
chmod +x run_triangle_counting_metis.sh

# Run the application
./run_triangle_counting_metis.sh
```

Expected Output

The application will output triangle count results with performance metrics showcasing the benefits of METIS-based graph partitioning and Aeron's communication implementation.

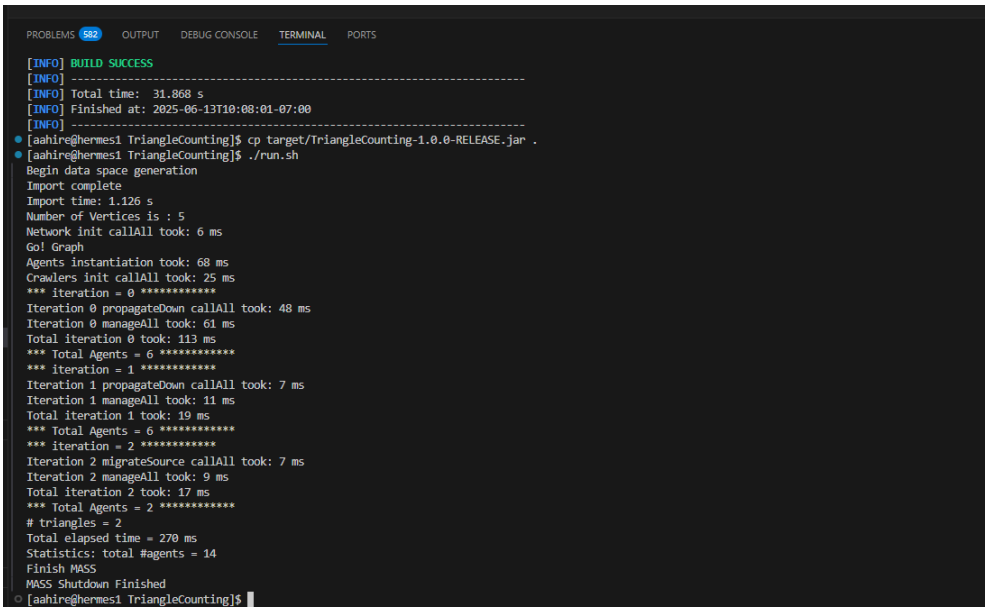

Figure C.2: Workflow: METIS partitioning with Aeron communication's output