© Copyright [2026]

Bo Fu

Enhancing Parallelization of Agent-based Graph Computing

Bo Fu

A master thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2026

Reading Committee:

Munehiro Fukuda, Chair

Min Chen

Robert Dimpsey

Program Authorized to Offer Degree:

Computer Science and Software Engineering

University of Washington

Abstract

Enhancing Parallelization of Agent-based Graph Computing

Bo Fu

Chair of the Supervisory Committee:
Munehiro Fukuda
School of Science, Technology, Engineering & Mathematics

The demand for distributed data processing grows as modern applications involve increasingly large and complex datasets. Traditional distributed computing frameworks, such as Apache Spark and Hadoop MapReduce, are effective for large-scale data processing but are not always well suited for graph computation. The MASS (Multi-Agent Spatial Simulation) Java library instead provides an agent-based approach to distributed graph computation and has been proven effective for graph computing applications and graph database. However, the performance of

MASS Java remains limited in some cases because graph applications often require many agent operations, which introduces significant overhead.

To address these limitations, this thesis introduces several enhancements for improving agent execution performance in MASS Java and evaluates them using graph computing applications and graph database queries. The evaluation shows that the enhancements can improve MASS Java performance in both graph computing and graph database query execution. In addition, this thesis identifies a major overhead in the current MASS graph database and proposes a solution to reduce it. Overall, this thesis contributes to the optimization and evaluation of MASS Java for graph applications and provides useful guidance for future development.

TABLE OF CONTENTS

List of Figures iii

List of Tables v

Listings vi

Chapter 1. Introduction 1

1.1 Motivation 2

1.2 Research Contributions 2

Chapter 2. Previous Work 4

Chapter 3. Related Work 7

3.1 Product Viewpoints 7

3.2 Technical Viewpoints 9

Chapter 4. Improvements of Parallel Performance 12

4.1 Agent Allocation Enhancement 12

4.2 GraphPlaces Multithreading 13

4.3 Hazelcast-inspired Graph Partitioning 14

4.4 Agent Initialization Control 15

4.5 Asynchronous Migration and Data Exchange 16

4.6 Lambda-based Reductive Computation 19

Chapter 5. Evaluation 21

5.1 Evaluation Design 21
5.2 Results 23
5.2.1 Triangle Counting 24
5.2.2 ACO-Based TSP 29
5.2.3 BFS 30
5.2.4 Wave2D 32
5.2.5 GraphDB 33
5.2.6 Further Performance Improvement 40
Chapter 6. Conclusion 42
6.1 Contributions 42
6.2 Limitations 43
6.3 Future Work 44
BibliographY 45
Appendix A Listings 48

LIST OF FIGURES

Figure 1 MASS Java Places and Agents Architecture [5] 4
Figure 2 Distributed HashMap Design in MASS Graph Database [7] 5
Figure 3 Comparison of Agent Allocation Strategies 13
Figure 4 GraphPlaces Multithreading Logic 14
Figure 5 Comparison of Graph Partitioning Strategies 15
Figure 6 Comparison of Agent Initialization Strategies 16
Figure 7 Comparison of Traditional and Asynchronous Agent Migration Flow 17
Figure 8 Comparison of Traditional and Asynchronous Data Exchange Flow 18
Figure 9 Execution Result Dataflow using Lambda Expression 19
Figure 10 Hierarchy-based Lambda Expression Execution 20
Figure 11 Lambda-based Reduction at Thread Level 20
Figure 12 Parallel Performance of TC - Random using Multithreading 24
Figure 13 Memory Usage of TC - Random on Different Number of Nodes 25
Figure 14 Parallel Performance of TC - Partitioned using Different Number of Threads 26
Figure 15 Parallel Performance of TC - Random using Agent Initialization Control 27
Figure 16 Parallel Performance of TC - Sparse using Agent Initialization Control 27
Figure 17 Parallel Performance of TC - Random using Graph Partitioning 28
Figure 18 Parallel Performance of TC - Partitioned using Graph Partitioning 28
Figure 19 Parallel Performance of ACO-Based TSP using Different Features 29
Figure 20 Collection Performance of ACO-Based TSP using Lambda-Based Reduction 30
Figure 21 Parallel Performance of BFS - Random using Asynchronous Migration 31
Figure 22 Parallel Performance of BFS - Sparse using Asynchronous Migration 31
Figure 23 Parallel Performance of Wave2D using Different Features 32
Figure 24 Parallel Performance of Wave2D using Different Features on 16 Nodes 33
Figure 25 Parallel Performance of IMDB Queries using Multithreading 34
Figure 26 Parallel Performance of IMDB Queries using Graph Partitioning 35

Figure 27 Parallel Performance of IMDB Queries using Agent Initialization Control 36
Figure 28 Parallel Performance of IMDB Queries using Asynchronous Migration 37
Figure 29 Performance Comparison of IMDB Queries on Single Node 38
Figure 30 Parallel Performance Comparison of IMDB Queries on Multiple Nodes........ 38
Figure 31 Performance Comparison of Shipment Queries on Single Node 39
Figure 32 Parallel Performance Comparison of Shipment Queries on Multiple Nodes... 39
Figure 33 Parallel Performance of Shipment Queries using Replicated Map on 4 Nodes 41

LIST OF TABLES

Table 1 Feature-Application Evaluation Matrix 22
Table 2 Graph Data Used for Evaluation 23

LISTINGS

Listing 1 Local Agent Bag Initialization ... 48
Listing 2 Local Agent Bag Based Execution ... 48
Listing 3 Local Agent Removal ... 48
Listing 4 Multithreaded GraphPlaces callAll() ... 48
Listing 5 Hazelcast-inspired Mapping Between VertexID and OwnerID ... 48
Listing 6 Agent Initialization Check on Places ... 49
Listing 7 Agent Constructor with Initialization Control ... 49
Listing 8 Asynchronous Migration Logic on Master Machine ... 50
Listing 9 Asynchronous Migration Logic on Worker Machines ... 50
Listing 10 Asynchronous Data Exchange Logic ... 51
Listing 11 Lambda-based Reduction on the Master Node ... 51
Listing 12 Lambda-based Reduction at Node Level ... 52
Listing 13 Lambda-based Reduction at Thread Level ... 52

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor and committee chair, Professor Fukuda, for his guidance, encouragement, and support throughout my thesis work. My thanks also go to my committee members for their time and valuable suggestions on my work. I am grateful to my colleagues and friends in the Distributed Systems Lab for their helpful discussions and advice during this work. I also deeply appreciate my girlfriend, Yi Xu, for her companionship throughout my graduate studies. Finally, I want to give special thanks to my parents for their unconditional love and support, which have been an important source of strength as I pursued my master's degree.

Chapter 1. INTRODUCTION

The demand for distributed data processing has increased rapidly as modern applications involve increasingly large and complex datasets. Traditional distributed computing frameworks, such as Apache Spark [1] and Hadoop MapReduce [2], provide effective support for large-scale batch processing and data analytics. However, these frameworks are not always well suited for computations over complex and irregular data structures like graphs, where vertices and edges may require frequent communication, traversal, and state updates.

Agent-based modeling (ABM) provides a useful computational model for complex simulations by representing a system as a collection of autonomous agents that interact with each other and with their environment. Existing ABM frameworks, such as Repast HPC [3] and FLAME [4], support distributed agent-based simulation, but they are not primarily designed for graph computation or graph database query execution. As a result, applying these frameworks to graph applications may introduce inefficiencies in data access, communication, and execution.

To address these limitations, previous researchers in the Distributed Systems Lab (University of Washington, Bothell) designed and implemented Multi-Agent Spatial Simulation (MASS) Java library [5]. MASS provides two main distributed abstractions: *Places* and *Agents*. The *Places* class represents distributed data or locations, while the *Agents* class contains mobile computational entities that can move across places and interact with them. For graph computing applications, MASS Java represents vertices as *GraphPlaces* [6] to support graph computation in a distributed environment.

1.1 MOTIVATION

Previous research has evaluated the performance of MASS Java in graph computing applications [6]. Recent work [7], [8] further extended MASS Java to support graph database query execution by introducing *PropertyVertexPlace*, which represents a vertex in a graph database. Together, these efforts show that MASS Java has strong potential for distributed graph applications, including both graph computing applications and graph database query execution.

Despite these results, the performance of MASS Java remains limited in some cases. Graph applications in MASS Java often require many agent operations, including agent initialization, execution, migration, and data collection. These operations can introduce significant overhead when many agents frequently move across distributed graph structures, access data, or collect results. Therefore, further optimization is needed to reduce these overheads and improve the parallel performance of MASS Java for graph applications.

1.2 RESEARCH CONTRIBUTIONS

This research makes the following contributions:

1. Implements and explores enhancements for improving parallel performance in MASS Java, including agent allocation logic optimization, multithreaded graph computation, agent initialization control, asynchronous migration, graph partitioning algorithm, and lambda-based reductive computation support.
2. Evaluates these enhancements using MASS Java applications including Triangle Counting, Breadth-First Search, Ant Colony Optimization-based Traveling Salesman Problem, and 2-Dimensional Wave Simulation.

3. Benchmarks the MASS graph database using two datasets and compares its performance with mainstream graph database systems like ArangoDB and Neo4j.
4. Identifies the distributed HashMap's communication overhead as a major performance limitation in the current MASS Java graph database implementation and proposes a possible solution to reduce it.

These contributions provide useful insights for the MASS development team, academic users, researchers in related fields, and users of graph database systems.

Chapter 2. PREVIOUS WORK

The Multi-Agent Spatial Simulation (MASS) Java library is a parallel computing library designed for multi-agent and spatial simulation. Its core architecture is based on two abstractions: *Places* and *Agents*. As shown in Figure 1, *Places* is a multi-dimensional array where each element is represented by *Place* and dynamically allocated over a cluster of computing nodes. *Agents* are execution instances that can reside on a place, migrate to other places and interact with places and other agents [5].

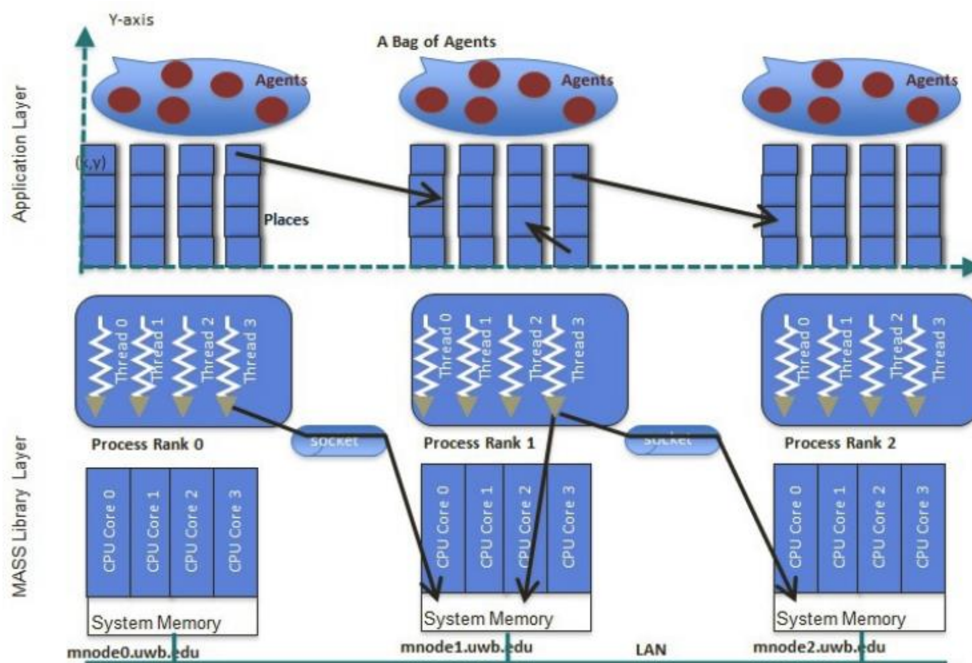

Figure 1 MASS Java Places and Agents Architecture [5]

Parallel computation in MASS is mainly supported through three methods: *callAll()*, *manageAll()* and *exchangeAll()*. The *callAll()* method invokes a specific function on all places or agents. The *manageAll()* method performs agent management operations, including spawning, termination and migration. The *exchangeAll()* method invokes a specified function on all places and then exchanges the function results with neighbors after execution [5].

To support agent-based graph computation, previous contributors extended the *Places* class to *GraphPlaces* and the *Place* class to *VertexPlace* [6]. *GraphPlaces* is designed for graph data storage, with each graph vertex represented as a *VertexPlace,* and edges are stored in a list.

Later contributors further extended MASS Java's capabilities and built the foundation for an interactive graph database system. This extension introduced *PropertyGraphPlaces*, a distributed collection of *PropertyVertexPlace* objects that store properties, incoming relationships, and outgoing relationships [7], [8]. This extension also integrated Cypher, a graph query language originally developed for Neo4j. The inclusion of Cypher enables users to perform pattern matching (MATCH), insertion (CREATE) and results filtering (WHERE) on the graph. In addition, this implementation introduced a distributed HashMap to map complex database keys, such as string keys, to indices used by MASS Java, as shown in Figure 2.

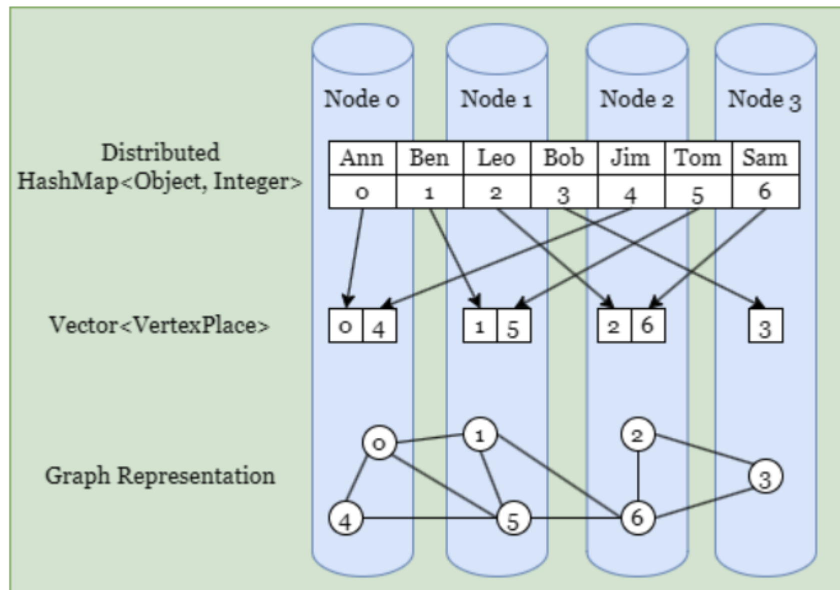

Figure 2 Distributed HashMap Design in MASS Graph Database [7]

Recent studies [8], [9], [10] explored the parallel performance of MASS Java using benchmarks from graph computing applications and graph database query execution. These studies compared MASS Java with graph computing and graph database systems, including GraphX [11],

ArangoDB [12], and Neo4j [13]. The benchmark results show that MASS Java can achieve competitive performance and outperform these systems in certain scenarios. However, the results also indicate that current MASS Java does not consistently outperform these systems across all benchmarks. Its performance remains limited in some cases, particularly in graph database query execution, where the overhead of distributed execution can reduce the benefits of parallelization. These studies [8], [14] have discussed possible causes of these limitations, which provide important directions for the enhancements explored in this research.

Chapter 3. RELATED WORK

In recent years, the increasing scale and complexity of modern applications have encouraged the development of many parallel computing frameworks and agent-based modeling (ABM) libraries. This chapter reviews related work from two perspectives: the product viewpoint, which discusses representative ABM libraries and parallel programming frameworks, and the technical viewpoint, which compares existing techniques with the enhancements implemented in this thesis.

3.1 PRODUCT VIEWPOINTS

Agent-based modeling (ABM) is an approach for modeling systems composed of autonomous and interacting agents. As real-world systems become increasingly complex, conventional modeling tools often struggle to capture their dynamic interactions, leading to the widespread adoption of ABM [15].

Repast for High Performance Computing (Repast HPC) [3], [16] is an agent-based modeling system intended for large-scale distributed computing platforms. It is implemented in C++ and uses Message Passing Interface (MPI) to support distributed agent-based simulations. Repast HPC provides good performance on distributed computing platforms and supports MPI-based collective communication operations. It also supports grid and N-dimensional space simulations, making it suitable for large-scale spatial ABM applications. Despite these advantages, Repast HPC is not specifically optimized for graph computation or graph database query execution. In addition, its reliance on C++ and MPI may make it difficult to learn and use compared with Java-based libraries.

Flexible Large-scale Agent Modeling Environment (FLAME) [4], [17], [18] is an agent-based modeling framework for developing models that can run on high-performance computing platforms. Unlike frameworks that require users to directly implement parallel execution logic,

FLAME provides ways to automatically generate simulation programs from model specifications. This design makes FLAME relatively user-friendly and allows flexible agent definition and initialization. However, FLAME is still mainly designed for general agent-based simulation and lacks optimization for graph applications. In addition, the original FLAME implementation may suffer from communication overhead, though its GPU-based version has better performance [19].

While ABM frameworks support large-scale simulations through autonomous and interacting agents, parallel computing libraries provide more general mechanisms for communication, scheduling, and data processing. Open MPI [20] is an open-source implementation of the Message Passing Interface, a widely used standard for parallel programming. Although it provides low-level control over process communication and can achieve minimal overhead, this flexibility increases the programming complexity and requires manual management of data distribution and synchronization. Dynamic load balancing and multithreading also require manual implementation [21], which makes Open MPI less convenient for developing complex graph applications.

Hadoop MapReduce [2] is a parallel data processing framework designed for large-scale batch processing. It provides good scalability for large datasets, fault tolerance, and dynamic task scheduling over cluster nodes. However, its batch-oriented model, disk-based I/O, and synchronization overhead make it less suitable for fine-grained tasks. Also, the Map and Reduce model limits its ability to process complex applications, such as graph applications, which often involve irregular data access, complex relationships, and frequent communication among vertices.

GraphX [11] is a graph processing framework built on Apache Spark [1]. It extends Spark by providing graph abstractions for representing vertices, edges, and associated properties. Since it is built on Spark, it benefits from Spark's task scheduling, data partitioning mechanisms, lambda expression support, and memory-based computation. Although these features make GraphX

effective for some graph computing applications, the immutability of RDDs makes it less suitable for dynamic graph computation. Furthermore, its vertex-centric computation model may be less flexible than agent-based models and can introduce high communication overhead when applied to graph computation that requires frequent interaction among vertices.

GraphLab [22] is a graph-parallel framework designed for large-scale machine learning and data mining applications. It supports asynchronous graph computation, making it effective for some machine learning applications over graph datasets. However, asynchronous execution increases programming complexity, and its parallelism can be influenced by the consistency settings. In addition, GraphLab is also based on vertex-centric computation, which can be less flexible than agent-based models and introduce high communication overhead when the application requires frequent interaction among vertices.

Overall, the reviewed systems provide useful support for large-scale simulation, parallel execution, and graph processing, but they also have limitations on graph computation and database query execution. Although some graph processing frameworks like GraphX provide support for graph computation, they are usually based on vertex-centric model, which can be less flexible than agent-based models for dynamic processing. In addition, these systems cannot be directly reused as a graph database system because they lack native support for property graph data structures, distributed indexing and query parsing. Therefore, MASS Java provides a useful foundation for agent-based graph computation and graph database query execution, while the challenge is to improve its parallel performance.

3.2 TECHNICAL VIEWPOINTS

This research improves parallel performance in MASS Java through agent allocation enhancement, GraphPlaces multithreading, Hazelcast-inspired graph partitioning, agent initialization control,

asynchronous migration and data exchange, and lambda-based reductive computation. Although similar ideas have appeared in previous systems or prior MASS Java work, the methods in this thesis are not direct replications of those approaches. Instead, they are redesigned for MASS Java's agent-based execution model and its graph computation requirements.

Agent allocation enhancement and GraphPlaces multithreading are both extensions of previous MASS Java work [5]. The original agent allocation method involved high synchronization overhead, and the enhancement in this thesis reduces this overhead by improving the allocation logic. Previous MASS Java work also introduced multithreading for N-dimensional space simulations, while this research extends multithreading support to *GraphPlaces*. Therefore, these two enhancements are different from previous work because they address synchronization and thread-level execution problems in agent-based graph computation.

The graph partitioning method in this thesis is inspired by Hazelcast [23], which distributes key-value data across cluster members through partitions. However, Hazelcast's partitioning mechanism is designed for key-value pairs and maps keys to partitions through hashing. In contrast, this research applies partitioning to graph structures. It uses key-based partitioning without hashing and groups nearby vertices into partitions before assigning them to nodes. This design considers vertex locality and can reduce cross-node communication during graph computation.

Agent initialization control is also supported in FLAME [17]. However, FLAME only allows users to define agent locations using model specifications before execution, while this research provides a more dynamic runtime method by allowing users to control the initialization through customized functions. Instead of using fixed initialization settings, agents are created only when the graph vertices satisfy specified conditions.

Previous graph processing frameworks provide different execution models for managing communication and synchronization. GraphX [11] uses a stage-based graph execution model, while GraphLab [22] supports a more fully asynchronous computation model. The method in this thesis differs from both approaches because it is designed for agent-based rather than vertex-centric graph computation. It does not make the entire system fully asynchronous. Instead, it applies asynchronous optimization to specific MASS Java operations, especially agent migration and place data exchange.

Reductive computation has been supported in different parallel computing frameworks. Hadoop MapReduce [2] separates computation into Map and Reduce stages, where intermediate results are generated and then aggregated. Spark [1] also supports lambda-based reductive computation through its programming interface. This research adapts the lambda-based reductive computation into MASS Java's agent-based model. Furthermore, it is integrated with the multithreading enhancement discussed earlier, forming a hierarchical parallel reduction mechanism across the master node, remote nodes, and threads.

Overall, the methods proposed in this thesis differ from existing techniques. They improve MASS Java's parallel performance by increasing single-node parallelism through multithreading and agent allocation enhancements, while reducing multi-node communication overhead through the other proposed methods. These methods may also provide useful references for related systems, including distributed batch-processing frameworks, vertex-centric graph systems, and agent-based modeling frameworks, because communication overhead remains a major challenge in all these systems.

Chapter 4. IMPROVEMENTS OF PARALLEL PERFORMANCE

This chapter presents the design and implementation of the methods used to improve the parallel performance of MASS Java. These methods include agent allocation enhancement, GraphPlaces multithreading, Hazelcast-inspired graph partitioning, agent initialization control, asynchronous migration and data exchange, and lambda-based reductive computation.

4.1 AGENT ALLOCATION ENHANCEMENT

Figure 3 shows the design of the original and improved agent allocation logics. In the original logic, all threads shared a single global agent bag containing all agents. Each thread retrieved agents sequentially from this global bag, which introduced high synchronization overhead. Moreover, because agents were randomly assigned to threads, each thread could process agents located in many different places, resulting in poor data locality. To address these issues, the proposed design constructs separate local agent bags for each thread and assigns agents to these bags according to their locations.

The implementation includes the creation and maintenance of local agent bags and the execution logic for agents. Specifically, during the initialization stage of *callAll()*, each thread creates a *ThreadLocal* queue based on the places assigned to it. In the following execution stage, each thread retrieves agents from its local queue until the queue becomes empty. Finally, when an agent terminates or migrates, the corresponding agent reference should be removed from its original place. In addition, to remain consistent with the original implementation based on the global agent bag named *AgentList*, this implementation also maintains the references in the *AgentList* structure. The implementation code is shown in Listings 1 to 3 in the Appendix A.

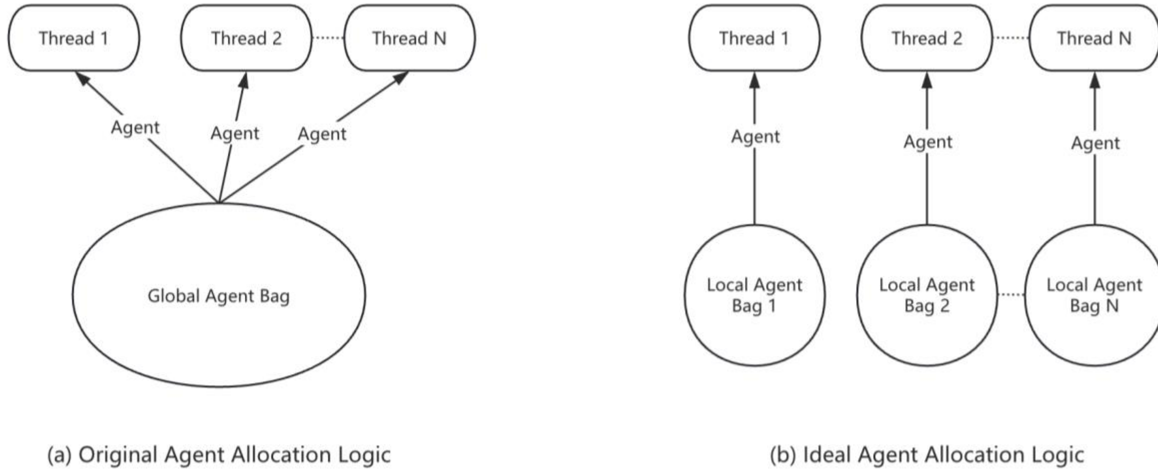

Figure 3 Comparison of Agent Allocation Strategies

4.2 GRAPHPLACES MULTITHREADING

Figure 4 shows how this research project implements the multithreading feature on *GraphPlaces*. In the previous MASS Java implementation, graph applications could not effectively use multithreading. As a result, only one thread executed operations like *callAll()* on *GraphPlaces* or agents. This limited the utilization of multi-core CPUs and reduced the parallel performance of graph applications.

In the new implementation, multiple threads are used to process different *GraphPlaces* or agents at the same time. The graph is first divided into multiple groups of vertices, and these vertex groups are assigned to different threads. Each thread then executes the required method on its assigned vertices in parallel. After all threads finish their local execution, a barrier is used to synchronize the threads before the program moves to the next stage. This method improves the parallelism of graph computation on a single node, which can further improve the overall parallel performance. The code is shown in Listing 4 in Appendix A.

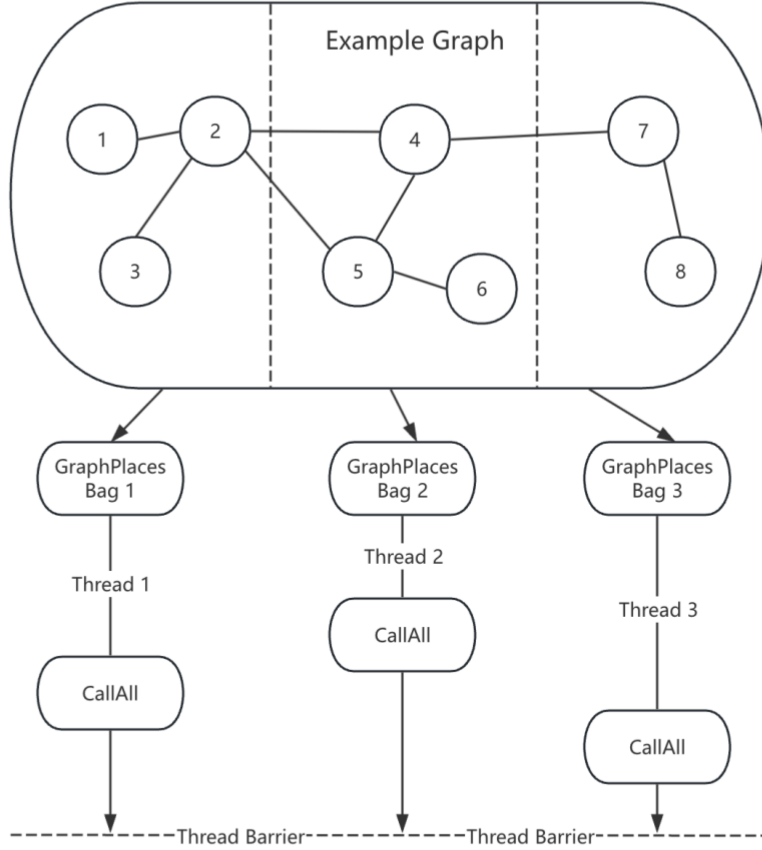

Figure 4 GraphPlaces Multithreading Logic

4.3 HAZELCAST-INSPIRED GRAPH PARTITIONING

Figure 5 shows the comparison of Round-Robin and Hazelcast-inspired graph partitioning. Previous MASS used a Round-Robin strategy, as shown in Equation 1. In this approach, adjacent vertices are often distributed across different nodes, even when they are close to each other in the graph structure. This can reduce data locality and increase communication overhead when agents need to move between neighboring vertices or access adjacent vertex data.

$$ownerID = vertexID \bmod NUM_NODES \quad (1)$$

The proposed implementation uses a Hazelcast-inspired logic, as shown in Equation 2. In this approach, vertices are first divided into different partitions based on the customized partition size. Then, these partitions are assigned to different nodes. Therefore, vertices with nearby IDs are

grouped into the same partition before node assignment. This design can improve locality when nearby vertex indices also represent closely connected vertices in the graph structure. However, if vertex indices are not correlated with graph locality, the partitioning strategy may provide limited benefit or even introduce imbalance. The code is shown in Listing 5 in Appendix A.

$$ownerID = \lfloor vertexID/PARTITION_SIZE \rfloor \; mod \; NUM_NODES \tag{2}$$

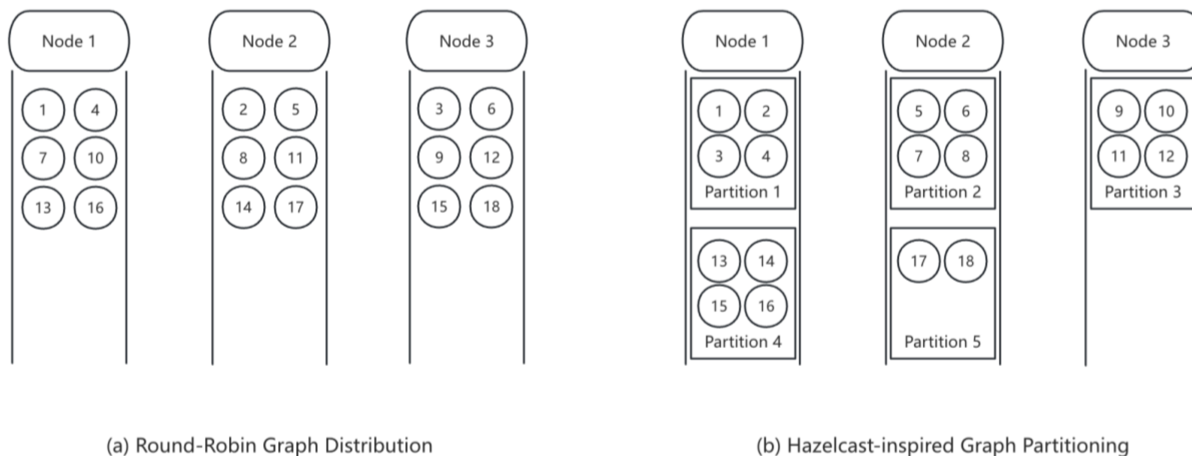

Figure 5 Comparison of Graph Partitioning Strategies

4.4 Agent Initialization Control

Figure 6 shows the comparison between the original and new agent initialization logic. Previous MASS lacked control over agent initialization, so agents were initialized on all graph vertices (known as *GraphPlaces*). However, most agents may terminate immediately after initialization in realistic applications like graph database queries, which introduces high additional initialization overhead. To address this issue, the new logic adds a condition check before agent initialization, so agents are initialized only on places that satisfy the given condition.

The implementation adds new *Agents* constructors that allow users to pass a specific *functionID* and an argument. This function invocation style is consistent with other MASS execution statements like *callAll()*. During initialization, each place executes the customized

function and only initializes an agent if the return value is *true*. The code is shown in Listings 6 and 7 in Appendix A.

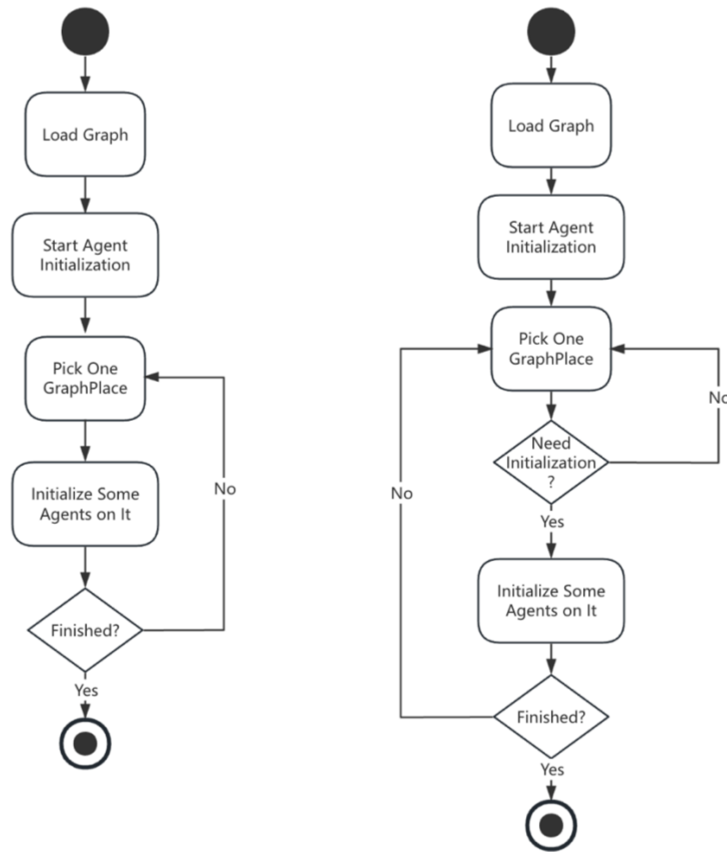

Figure 6 Comparison of Agent Initialization Strategies

4.5 ASYNCHRONOUS MIGRATION AND DATA EXCHANGE

In the previous MASS Java, each operation such as *callAll(), manageAll(),* and *exchangeAll()* is executed independently, and every step requires synchronization between the master and worker nodes. While this design is suitable for single-step execution, it becomes inefficient for iterative computations.

For applications that repeatedly execute the same sequence of operations, such as Breadth-First Search (BFS), the iteration logic must be manually implemented by repeatedly invoking these

operations. As shown in Figure 7 (a), for N iterations, the execution requires 6N message exchanges between the master and workers, leading to significant communication overhead.

In the new design, *callAll()*, *manageAll()*, and agent-level *exchangeAll()* are executed within a unified iterative flow. This design is not fully asynchronous at the system level, but it reduces repeated master-worker synchronization. As shown in Figure 7 (b), only a single condition check and synchronization are performed at the end of each iteration. As a result, each iteration requires two message exchanges, reducing the total number of message exchanges from 6N to about 2N.

The implementation introduces two functions, *doWhile()* and *doAll()*. The *doWhile()* function first initializes the required parameters for later execution and then starts a loop to asynchronously execute *callAll(), manageAll(),* and agent-level *exchangeAll().* At the end of each iteration, the master node synchronizes with workers and checks the number of agents on all nodes. If the total number of agents is zero, the loop ends. The *doAll()* logic follows a similar logic but uses a fixed number of iterations as the exit condition. The code is shown in Listings 8 and 9 in Appendix A.

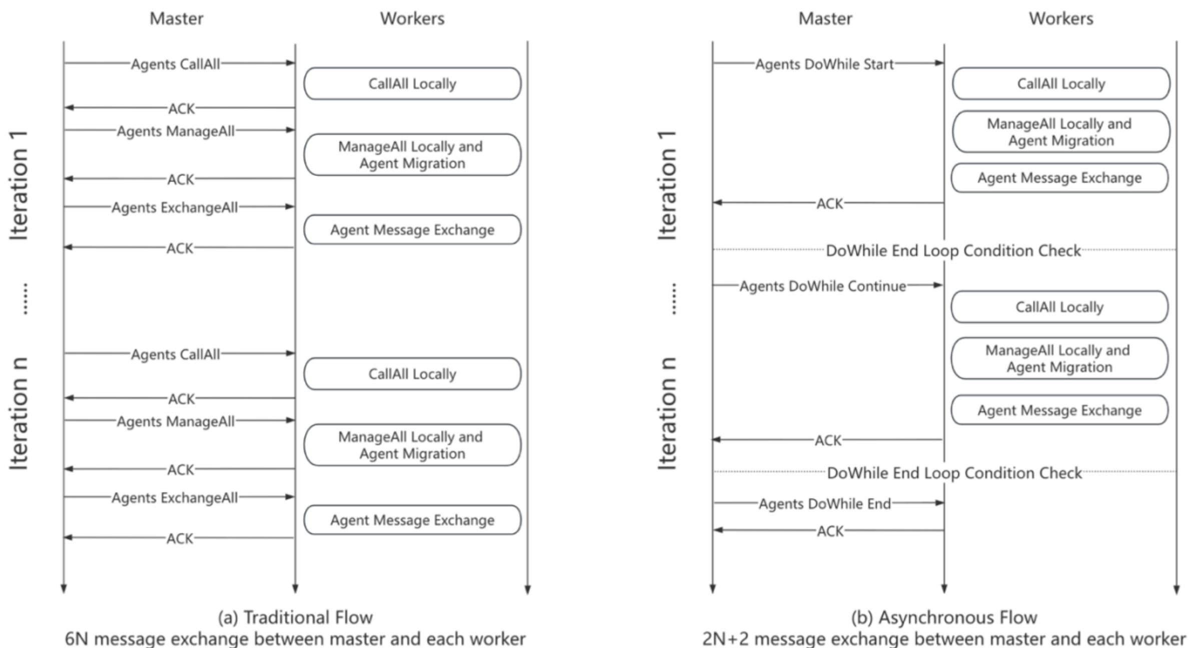

Figure 7 Comparison of Traditional and Asynchronous Agent Migration Flow

The asynchronous design for places is similar to that for agents, but it has two main differences. First, place-level *exchangeAll()* already includes peer-to-peer synchronization, so no extra barrier is required. Second, place-based applications, such as 2-Dimensional Wave Simulation (Wave2D), usually run for a fixed number of iterations. In contrast, agent-based applications may need additional exit conditions, such as stopping BFS when the total number of agents becomes zero. Because of these differences, place-level execution can be more fully asynchronous. As shown in Figure 8, the previous place-level iteration using *callAll()* and *exchangeAll()* required 4N message exchanges for N iterations. In the new design, the whole iterative execution can run asynchronously, so only two message exchanges are needed.

In the implementation of place-level *doAll()*, the master node sends the number of iterations to the worker nodes during initialization. Then, all nodes independently complete N iterations. After the loop finishes, the master node synchronizes with workers to ensure that all nodes have completed execution. The code is shown in Listing 10 in Appendix A.

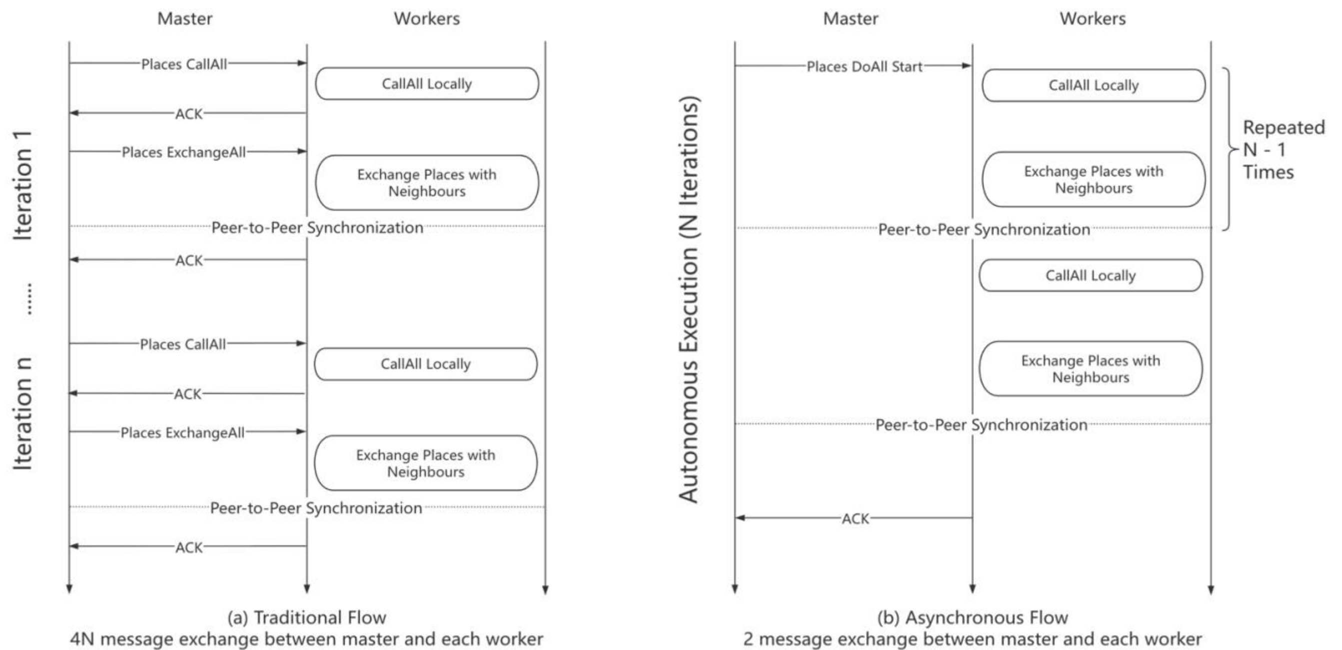

Figure 8 Comparison of Traditional and Asynchronous Data Exchange Flow

4.6 LAMBDA-BASED REDUCTIVE COMPUTATION

In the previous MASS Java implementation, reductive computation required users to manually collect all data from remote worker nodes to the master node. After the data collection step, the computation was performed on the master node. This introduced communication overhead and limited parallelism. Figure 9 shows the new design. In this design, every node reads data from local memory and performs the computation locally. Then, only the partial results are sent to the master node and get aggregated. This approach can improve the data collection performance in applications that require frequent reductions, such as Ant Colony Optimization-based Traveling Salesman Problem (ACO-based TSP).

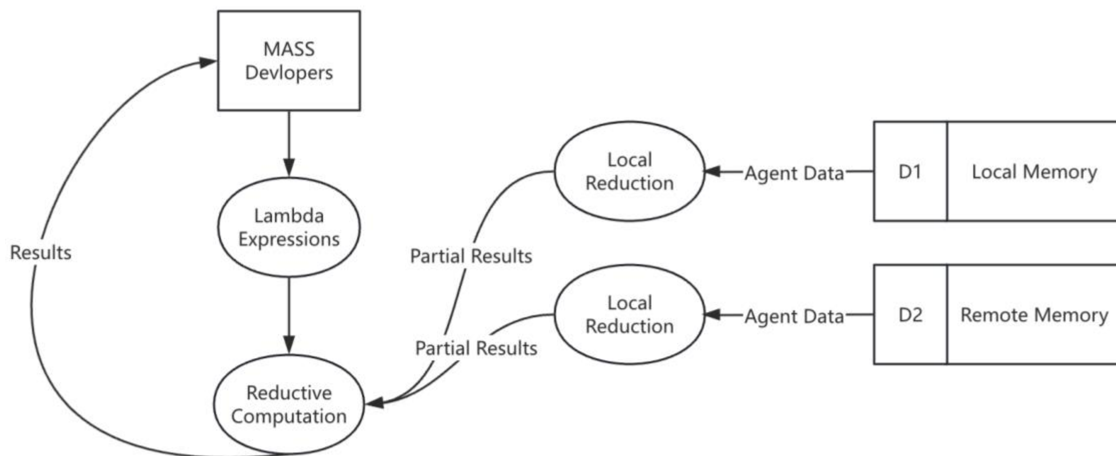

Figure 9 Execution Result Dataflow using Lambda Expression

This implementation uses a hierarchical execution process. The reductive computation is represented as two lambda expressions and passed to the main node through the *reduce()* method. As shown in Figure 10, the lambda expressions are then sent to all nodes through method invocation and message passing. On each node, the lambda expressions are further passed to threads and executed. This design improves both the node-level and thread-level parallelism.

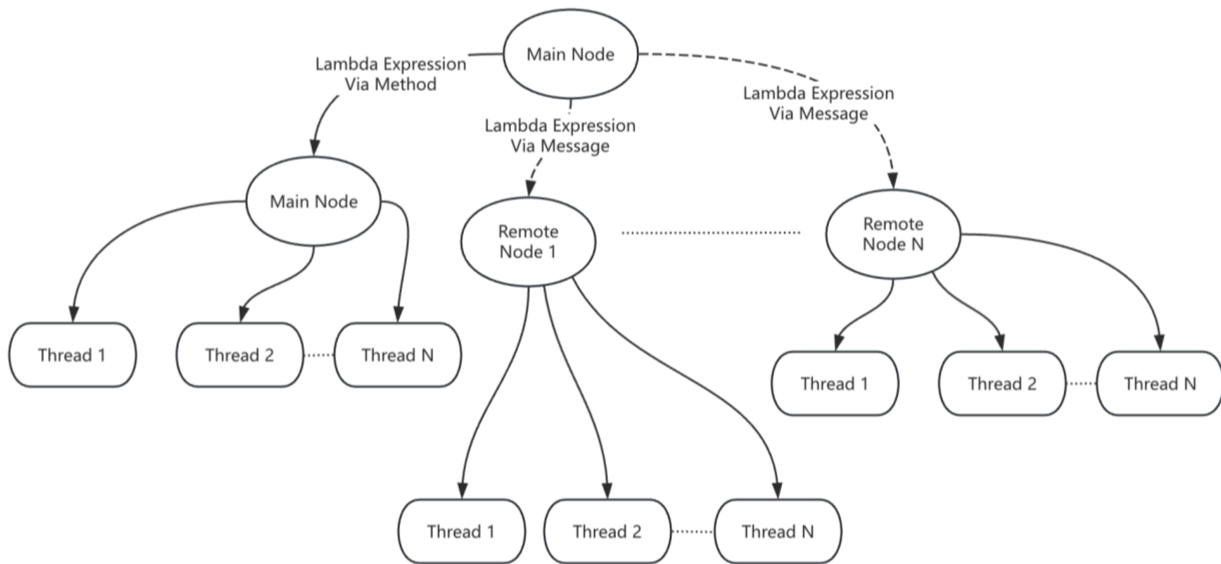

Figure 10 Hierarchy-based Lambda Expression Execution

The reduction process at the thread level is shown in Figure 11. The two lambda expressions are called *mapper* and *reducer*. The *mapper* maps each agent to an intermediate result while the *reducer* aggregates the received results into a single result. The aggregation process is also performed on local nodes and then on the master node to produce the final result. The code for the entire implementation is shown in Listings 11 to 13 in Appendix A.

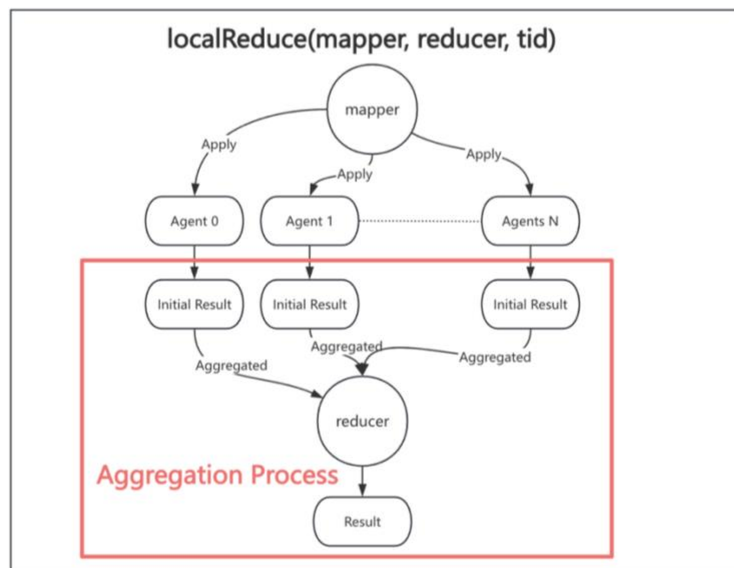

Figure 11 Lambda-based Reduction at Thread Level

Chapter 5. EVALUATION

This chapter evaluates the methods introduced in previous chapters, including the evaluation design, results, and analysis. First, this chapter describes the benchmark applications and datasets. Then, it presents the results and analyzes how the methods affect the performance of MASS Java.

5.1 EVALUATION DESIGN

This evaluation uses both graph computing applications and graph database query execution to test the proposed methods. As shown in Table 1, the selected applications include Triangle Counting (TC), Ant Colony Optimization-based Traveling Salesman Problem (ACO-based TSP), Breadth-First Search (BFS), 2-Dimensional Wave Simulation (Wave2D), and Graph Database (GraphDB) query execution. These applications are used to evaluate the performance impact of different configurations of MASS Java.

The configurations in Table 1 are defined as follows:

1. Previous MASS: The MASS Java implementation without the improvements proposed in this thesis.
2. New MASS: The MASS Java implementation with the improved agent allocation logic.
3. #1: The configuration that applies the multithreading feature on top of New MASS.
4. #2: The configuration that applies the Hazelcast-inspired graph partitioning feature on top of New MASS.
5. #3: The configuration that applies the agent initialization control feature on top of New MASS.

6. #4: The configuration that applies the lambda-based reductive computation on top of New MASS.
7. #5: The configuration that applies the asynchronous migration and data exchange feature on top of New MASS.
8. ArangoDB and Neo4j: External graph database systems for comparison with the graph database query execution in MASS Java.

In the tables, the checked sign means that the corresponding configuration is evaluated using the application, and N/A means that the configuration is not applied to the application because it is not expected to improve its performance.

Table 1 Feature-Application Evaluation Matrix

Application	Previous MASS	New MASS	#1	#2	#3	#4	#5	ArangoDB and Neo4j
TC - Random	✓	✓	✓	✓	✓	N/A	N/A	N/A
TC - Partitioned	✓	✓	✓	✓	N/A	N/A	N/A	N/A
TC - Sparse	✓	✓	N/A	N/A	✓	N/A	N/A	N/A
ACO-based TSP	✓	✓	N/A	N/A	N/A	✓	✓	N/A
BFS	✓	✓	N/A	N/A	N/A	N/A	✓	N/A
Wave2D	✓	N/A	N/A	N/A	N/A	N/A	✓	N/A
GraphDB	✓	✓	✓	✓	✓	N/A	✓	✓

Table 2 shows the datasets used in the evaluation. The three TC graphs differ in size and structure, making them suitable for evaluating the performance improvements of the proposed features under different settings. The ACO-based TSP dataset is relatively small, but the numbers of ants and iterations are sufficiently large to evaluate the impact of the proposed features. The BFS evaluation uses two different graphs, which allows this research to compare the performance improvement of asynchronous migration under different graph structures.

For graph database query execution, this research uses two datasets with different graph topologies. The IMDB dataset is smaller and contains movie-related entities, such as movies, actors, directors, and genres. Its topology mainly contains short relationship paths, where related entities can usually be reached within a small number of hops. Therefore, it is suitable for single-feature evaluation and shallow query execution. In contrast, the Shipment dataset is larger and has a more staged structure, where entities represent different steps in the shipment process. This structure naturally creates longer multi-hop paths, making it more suitable for evaluating deeper graph queries and comparing MASS Java with other graph database systems.

Table 2 Graph Data Used for Evaluation

Dataset	Number of Vertices	Number of Edges	Other Parameters
TC - Random	3000	147728	
TC - Partitioned	1000	62000	8 partitions
TC - Sparse	30000	59717	
ACO-based TSP	36	630	10000 ants, 20 iterations
BFS - Random	3000	147728	
BFS - Sparse	10000	46046	
IMDB	1097	4065	
Shipment	30000	40000	

The experiments were conducted on the HERMES cluster at the University of Washington Bothell, which consists of 24 computing nodes. Each experimental node was equipped with an Intel(R) Xeon(R) Gold 5315Y CPU @ 3.20GHz, 4 physical CPU cores, 8 hardware threads, and 16 GB of memory. In this evaluation, up to 16 nodes were used for the multi-node experiments.

5.2 RESULTS

This section presents the evaluation results and discusses the possible reasons for the observed performance changes. To improve the reliability of the evaluation results, each computing application experiment was executed three times, and the average execution time was reported. For graph database query execution, this research used the built-in benchmark mode. Specifically,

each query was warmed up once and then executed five times, and the average execution time was used as the result. During the evaluation, cluster machines were reserved to avoid running experiments together with other users' workloads. This reduced possible interference from resource contention and helped maintain consistency across repeated runs.

5.2.1 *Triangle Counting*

This subsection presents all evaluation results for the Triangle Counting (TC) application.

Figure 12 compares the parallel performance of Previous MASS, New MASS, and the multithreaded MASS for Triangle Counting on the randomly generated graph. The results show that the new allocation logic improves the performance of Triangle Counting because it provides better CPU utilization and cache efficiency during agent execution. The multithreaded version further improves performance by increasing single-node parallelism. However, the improvement decreases as the number of nodes increases, suggesting that communication overhead becomes the main overhead in the multi-node setting and offsets the benefit of using more computing nodes.

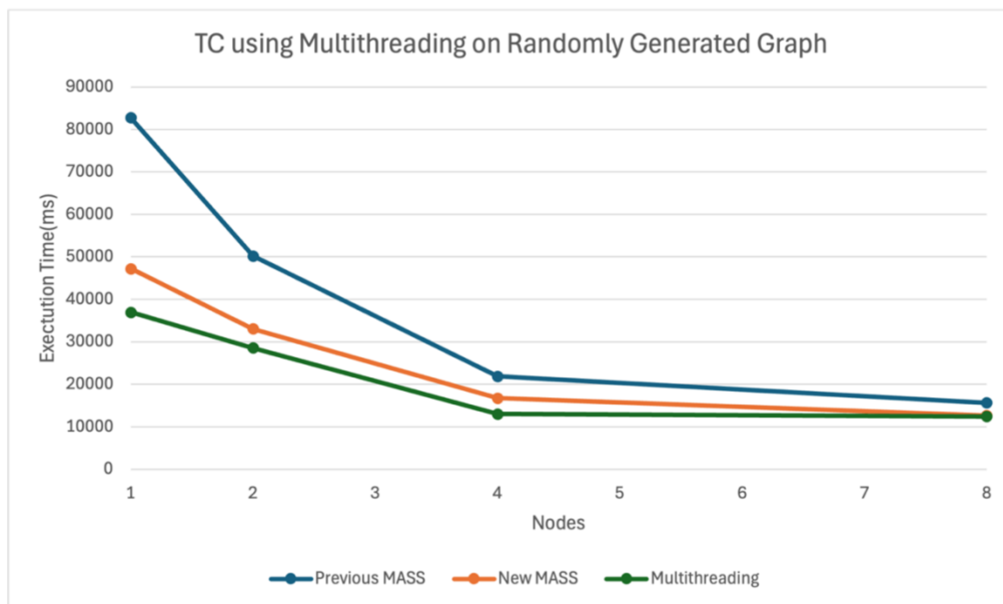

Figure 12 Parallel Performance of TC - Random using Multithreading

Figure 13 compares the memory usage of Previous MASS and New MASS. The memory usage is measured after the application finishes execution. As shown in the figure, New MASS reduces memory usage compared with Previous MASS. This improvement is mainly caused by the new agent allocation logic, which correctly removes agent references from Places after agents migrate to other locations. As a result, unused agent references are no longer retained in the original Places, reducing unnecessary memory consumption.

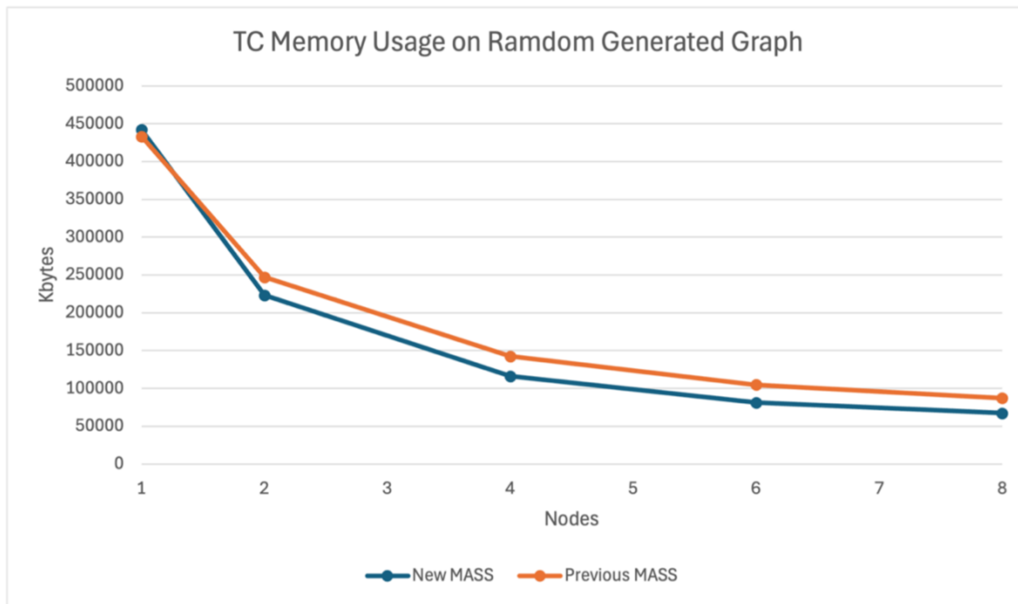

Figure 13 Memory Usage of TC - Random on Different Number of Nodes

Figure 14 evaluates the parallel performance of the multithreaded MASS using the partitioned graph. The results show that the performance improvement from multithreading is affected by the graph structure. The largest improvement is observed when the number of threads is equal to the number of partitions. This may be because this setting better matches thread-level parallelism with the graph partition structure, allowing each thread to process one partition more efficiently. As a result, the system may reduce scheduling overhead and improve cache locality during agent execution.

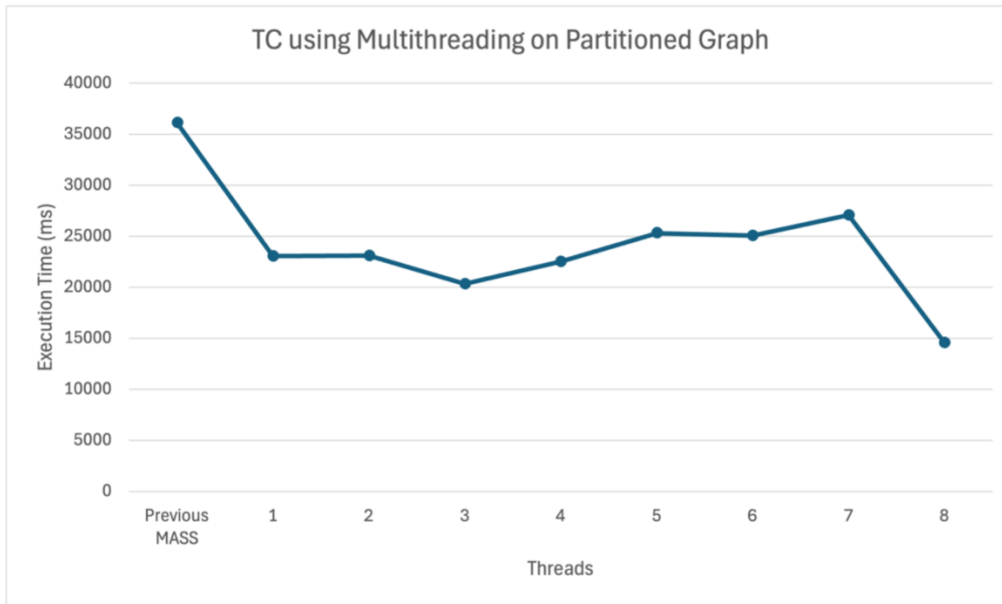

Figure 14 Parallel Performance of TC - Partitioned using Different Number of Threads

Figure 15 shows the impact of agent initialization control on Triangle Counting using randomly generated graph. In this graph, most vertices have multiple edges, so the number of agent initialization operations cannot be significantly reduced through simple initialization conditions. As a result, this feature does not bring clear performance improvement in this case.

In contrast, for Triangle Counting using special graph structures, such as sparse graphs, an initialization condition can be used to initialize agents only on vertices with at least two neighbors. This condition allows the system to skip many redundant initialization operations and achieve performance improvement, as shown in Figure 16.

This comparison shows that the effectiveness of agent initialization control depends on suitable initialization conditions and graph structures.

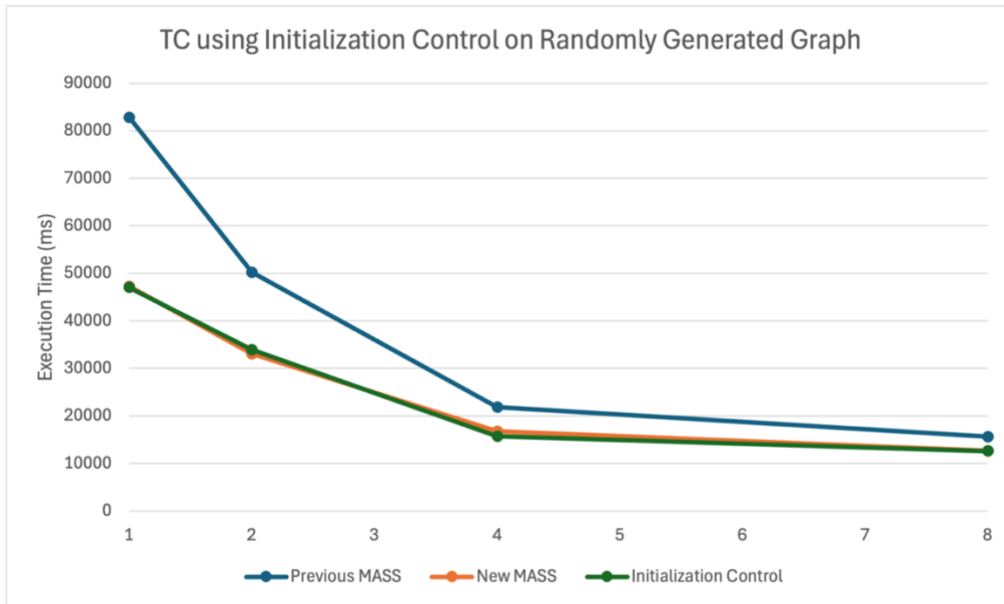

Figure 15 Parallel Performance of TC - Random using Agent Initialization Control

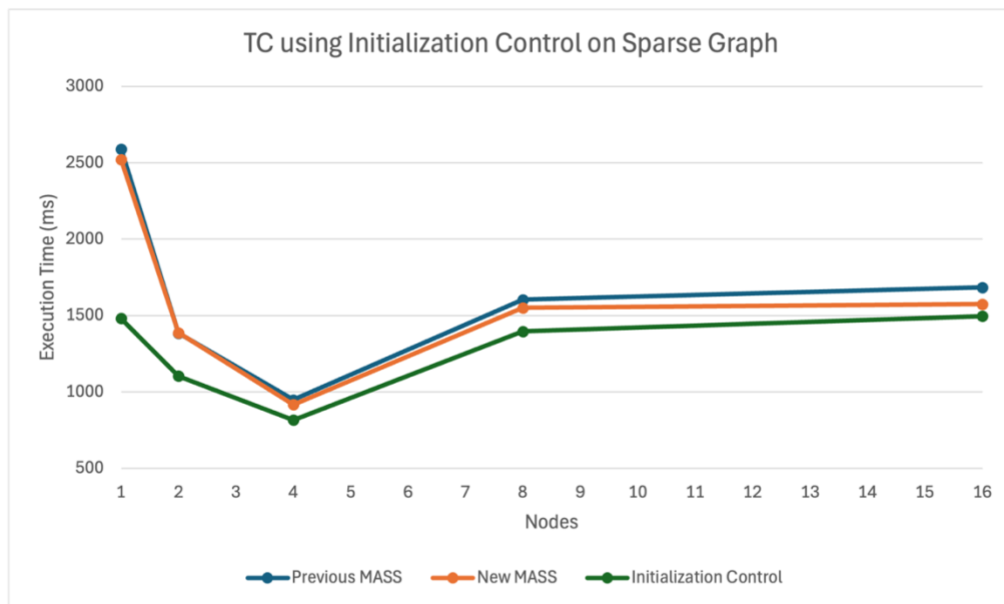

Figure 16 Parallel Performance of TC - Sparse using Agent Initialization Control

Figures 17 and 18 show how the Hazelcast-inspired graph partitioning feature affects the parallel performance of Triangle Counting. When Triangle Counting runs on the randomly generated graph, this feature even decreases performance. However, when it runs on the partitioned graph, the feature significantly improves performance.

This comparison shows that Hazelcast-inspired graph partitioning can improve parallel performance, but its effectiveness depends on the graph structure. It is more suitable for graphs with clear partitioned structures, where related vertices can be placed closer together and unnecessary cross-node communication can be reduced.

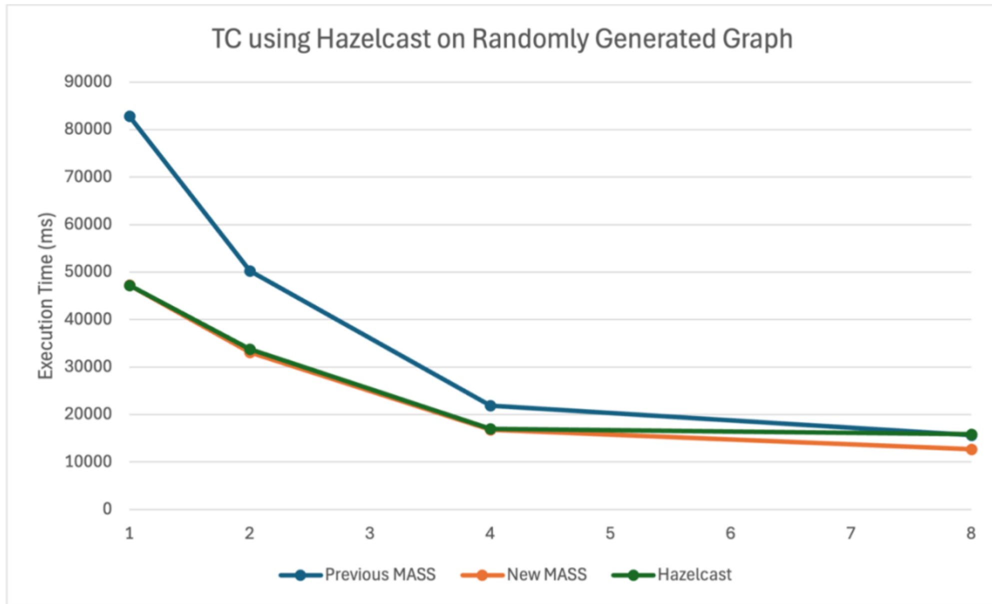

Figure 17 Parallel Performance of TC - Random using Graph Partitioning

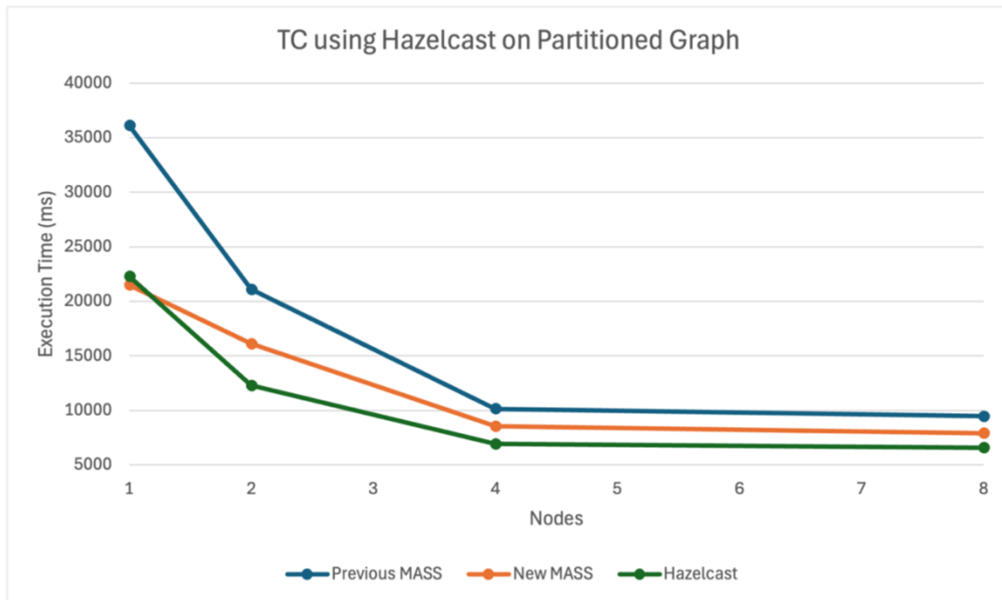

Figure 18 Parallel Performance of TC - Partitioned using Graph Partitioning

5.2.2 *ACO-Based TSP*

This subsection presents all evaluation results for the Ant Colony Optimization-based Traveling Salesman Problem (ACO-based TSP) application.

Figure 19 shows the execution performance of ACO-based TSP under four different configurations: Previous MASS, New MASS, New MASS with asynchronous migration, and New MASS with lambda-based reductive computation. When the number of nodes is small, especially fewer than four nodes, New MASS introduces some additional overhead. This may be caused by the frequent initialization of the local agent bag.

In addition, asynchronous migration does not bring a large performance improvement. It only provides some improvement when the application runs on 12 nodes. This may be caused by two reasons. First, when the number of nodes is small, the master-worker communication overhead is not large enough for asynchronous migration to provide a clear benefit. Second, in ACO-based TSP, the main communication overhead is concentrated in agent migration itself, so asynchronous migration cannot effectively reduce the main communication cost.

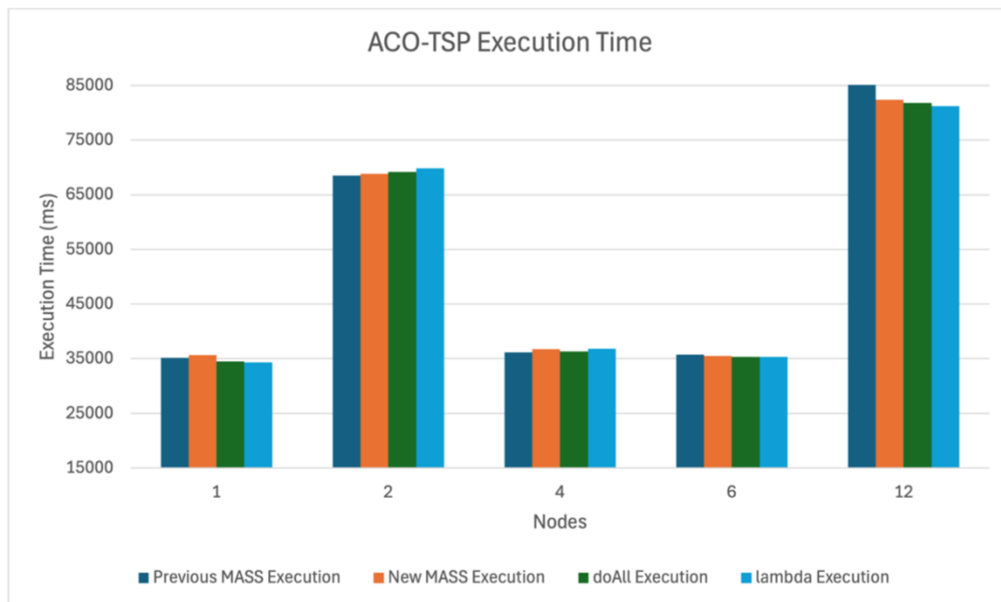

Figure 19 Parallel Performance of ACO-Based TSP using Different Features

In Figure 19, lambda-based reductive computation does not seem to bring a clear improvement in the total execution time. However, Figure 20 directly compares the data collection time under different configurations and shows that this feature reduces the data collection overhead by more than 90%. This result indicates that lambda-based reductive computation can effectively improve data collection performance. The improvement is not obvious in the overall result because data collection accounts for only a small part of the execution time in ACO-based TSP.

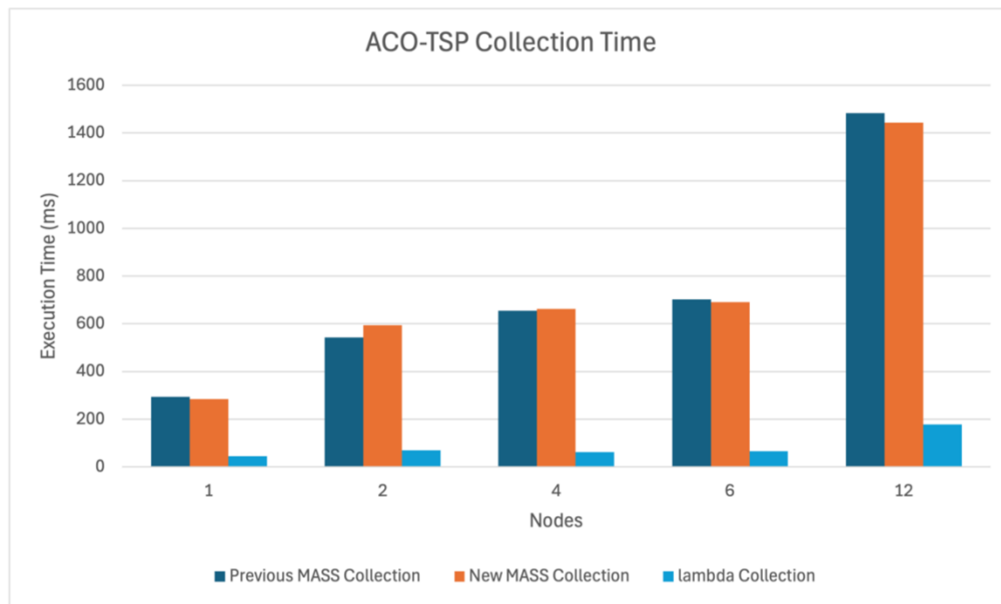

Figure 20 Collection Performance of ACO-Based TSP using Lambda-Based Reduction

5.2.3 *BFS*

This subsection presents all evaluation results for the Breadth-First Search (BFS) application.

Figures 21 and 22 compare the performance of Previous MASS, New MASS, and New MASS with asynchronous migration for BFS using two different graph structures. The results show that asynchronous migration brings only small improvements on the randomly generated graph but provides more noticeable improvements on the sparse graph. This is because the sparse graph creates fewer agents, so the overhead from agent execution and migration is smaller. As a result,

master-worker communication overhead becomes more visible in the total execution time, making the benefit of asynchronous migration more noticeable. Under the same graph structure, the improvement also becomes more visible as the number of nodes increases, because the master-worker communication cost becomes higher with more nodes.

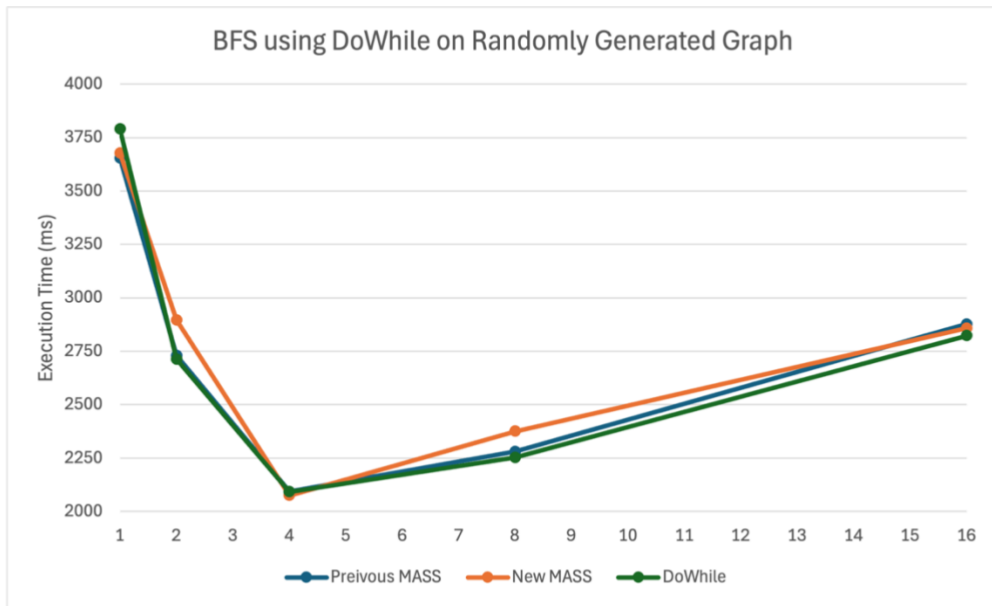

Figure 21 Parallel Performance of BFS - Random using Asynchronous Migration

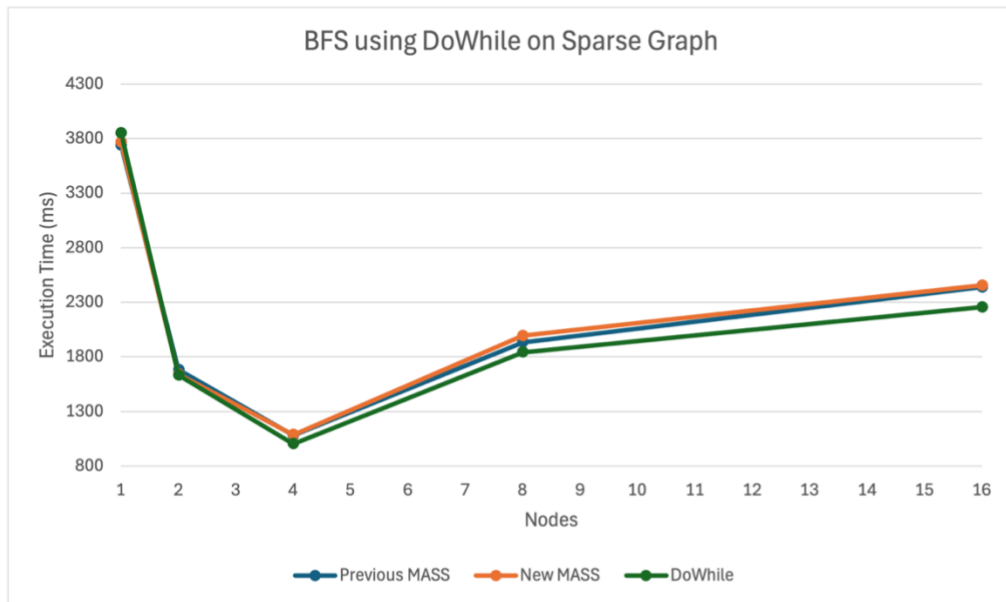

Figure 22 Parallel Performance of BFS - Sparse using Asynchronous Migration

5.2.4 *Wave2D*

This subsection presents all evaluation results for the 2-Dimensional Wave Simulation (Wave2D) application.

Figure 23 compares the parallel performance of *exchangeAll()* and *exchangeBoundary()* between the original version and the asynchronous version. This evaluation uses the Wave2D application with a 500 x 500 grid size and 100 iterations. In this evaluation, the *exchangeAll()* method exchanges data among all Places, while *exchangeBoundary()* only exchanges information for boundary Places. The results show that asynchronous data exchange can bring significant performance improvement for Wave2D, especially on 8 and 16 nodes. This is because Wave2D requires frequent data exchange during execution, which introduces high master-worker communication overhead. The asynchronous data exchange feature can significantly reduce this overhead, leading to better parallel performance.

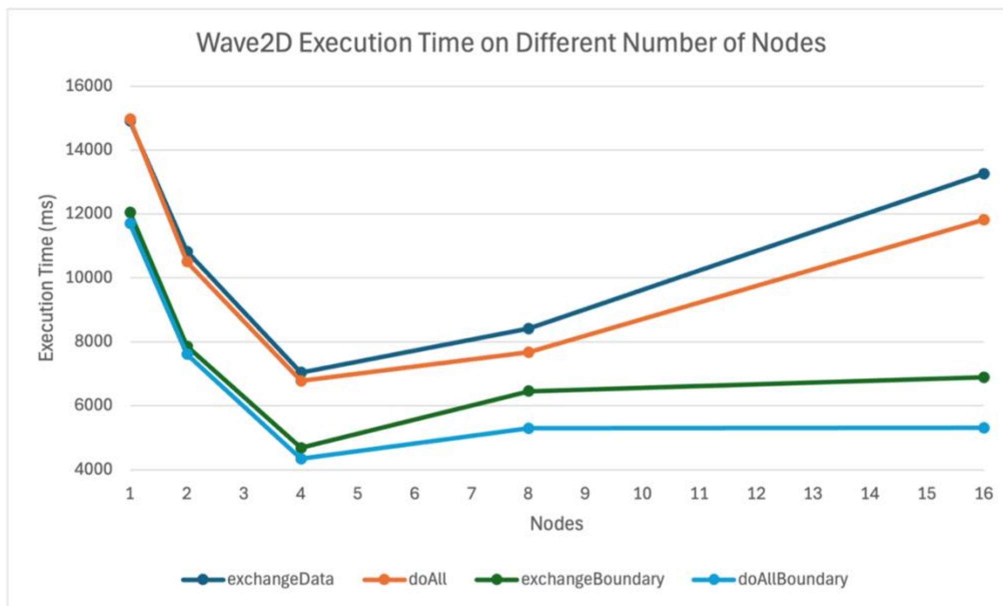

Figure 23 Parallel Performance of Wave2D using Different Features

Figure 24 compares the impact of asynchronous data exchange under different Wave2D settings, including 500 × 500 with 100 iterations, 1000 × 1000 with 100 iterations, and 500 × 500

with 500 iterations. The results show that increasing the grid size does not clearly increase the absolute improvement brought by this feature. However, increasing the number of iterations leads to a larger reduction in execution time, because more iterations introduce more rounds of data exchange and more master-worker communication overhead.

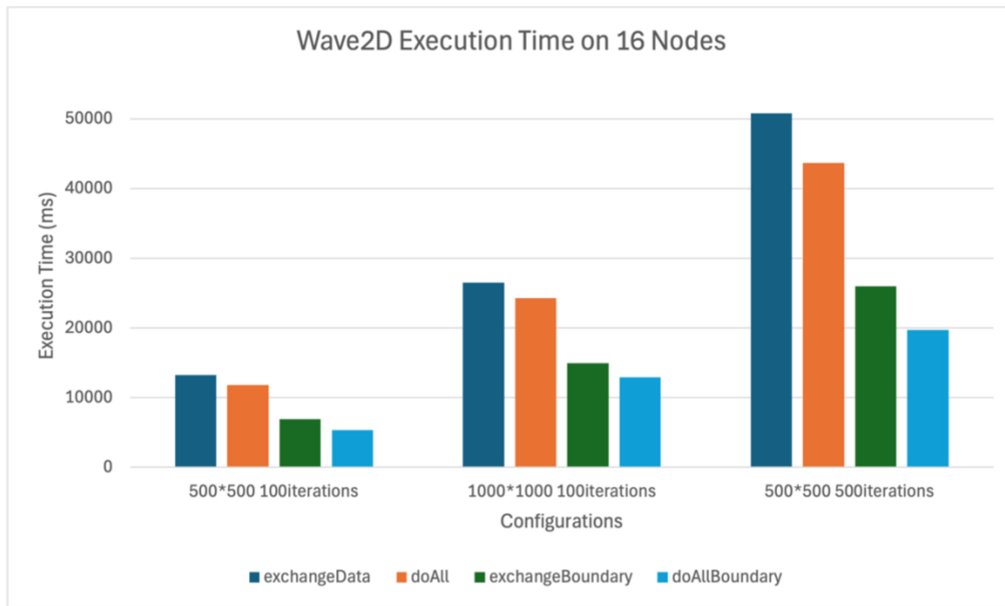

Figure 24 Parallel Performance of Wave2D using Different Features on 16 Nodes

5.2.5 *GraphDB*

This subsection presents the evaluation results for graph database (GraphDB) query execution. The evaluation includes the performance improvement brought by different features using the IMDB dataset, as well as the execution performance comparison between MASS Java, ArangoDB, and Neo4j using the IMDB dataset and the Shipment dataset.

Figure 25 shows that the multithreading feature can improve the performance of GraphDB queries, especially for depth-2 queries. This may be because depth-2 queries spend more time on actual query execution. In contrast, queries with smaller depths spend a larger portion of time on

query parsing and other fixed overheads like initialization, which cannot be reduced by the multithreading feature.

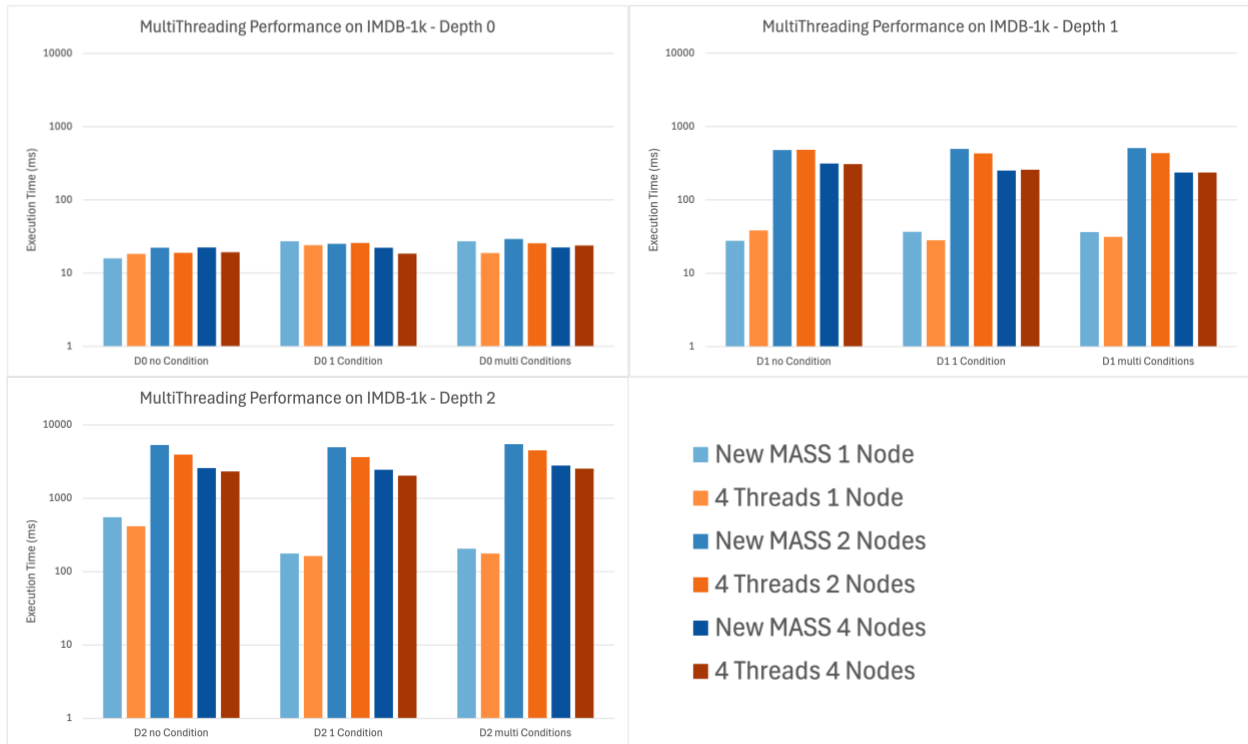

Figure 25 Parallel Performance of IMDB Queries using Multithreading

Figure 26 shows that Hazelcast-inspired graph partitioning brings limited performance improvement for GraphDB queries. On 8 nodes, MASS with this feature performs close to or slightly faster than the baseline version, but the overall improvement is not significant. As discussed earlier, the benefit of this feature strongly depends on the graph structure. Therefore, if the graph cannot be effectively optimized by the partitioning strategy, the performance improvement remains limited.

Figure 26 Parallel Performance of IMDB Queries using Graph Partitioning

As shown in Figure 27, agent initialization control provides a relatively clear performance improvement, especially for depth-0 and depth-1 queries. Without initialization control, many agents are created on vertices that do not satisfy the query conditions, and these agents quickly terminate after execution starts. Agent initialization control avoids these unnecessary initialization operations by creating agents only on vertices that may match the query, thereby enhancing the execution performance of GraphDB queries. The improvement is more visible for depth-0 and depth-1 queries because these queries involve less migration and traversal overhead. For depth-2 queries, other overheads, such as agent migration, become more dominant, so the benefit of reducing initialization overhead becomes less obvious.

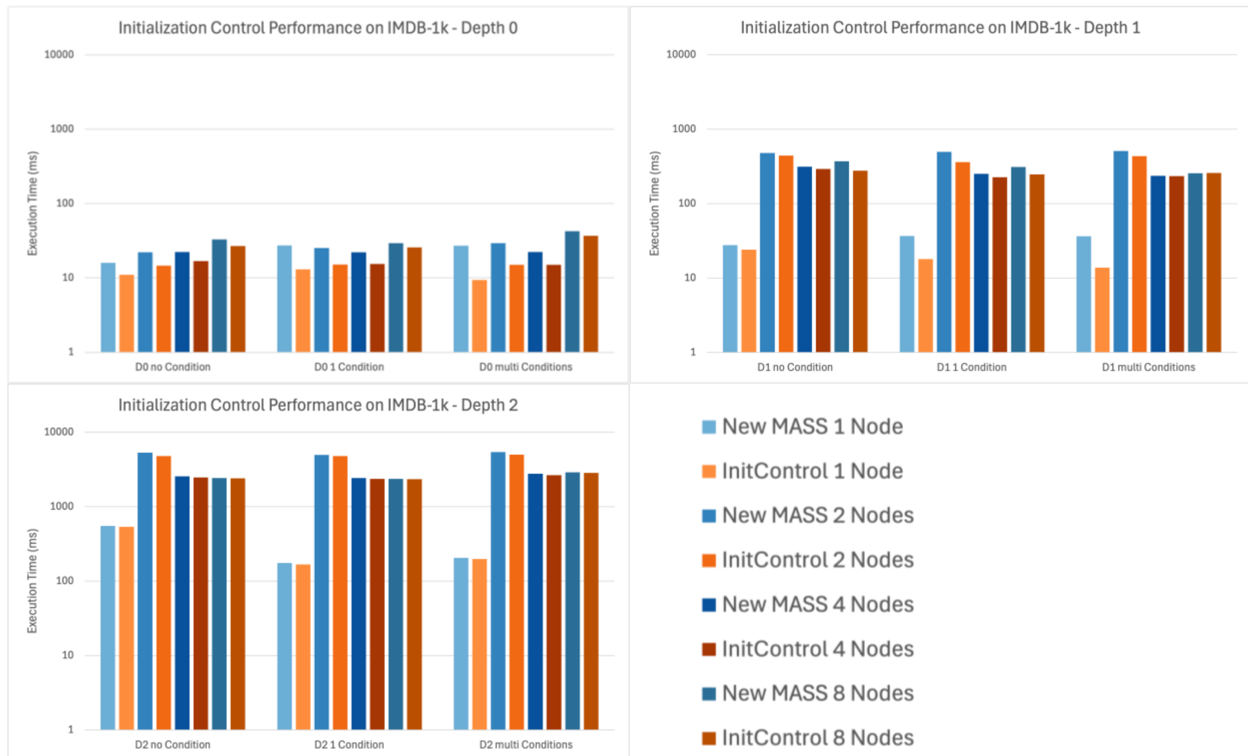

Figure 27 Parallel Performance of IMDB Queries using Agent Initialization Control

As shown in Figure 28, asynchronous migration brings a small performance improvement, and the improvement is slightly more noticeable on 8 nodes. The overall improvement is limited for two possible reasons. First, the current implementation of asynchronous migration is not fully asynchronous and still retains some synchronization steps. These remaining synchronization steps limit reduction of communication overhead. Second, in GraphDB query execution, the synchronization overhead reduced by asynchronous migration may not be the dominant overhead. Other costs, such as query parsing, agent execution, and agent migration, may account for a larger portion of the total execution time.

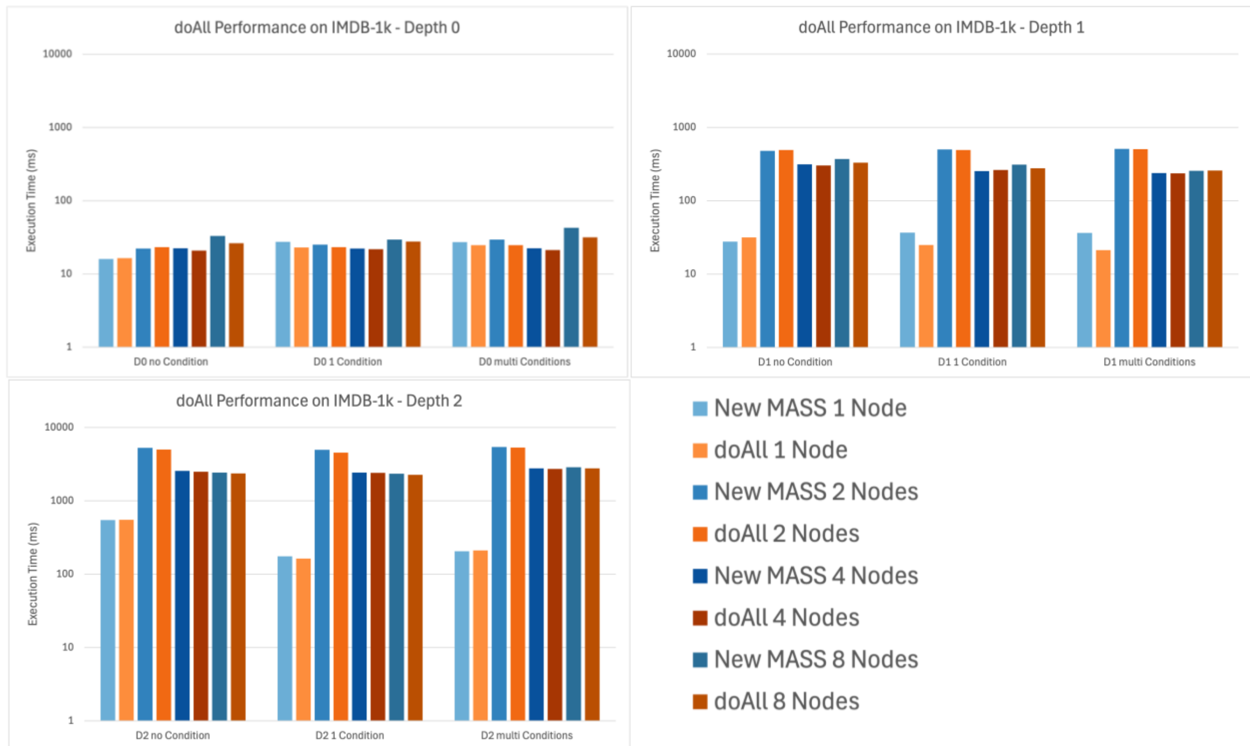

Figure 28 Parallel Performance of IMDB Queries using Asynchronous Migration

This research also compares the query execution performance of Previous MASS, New MASS with proposed enhancements, ArangoDB, and Neo4j in both single-node and multi-node environments. The comparison uses the same queries and datasets to evaluate how MASS Java performs against existing graph database systems.

Figure 29 shows that New MASS with enhancements improves the query performance compared with Previous MASS on the IMDB dataset. However, it still does not outperform Neo4j. Compared with ArangoDB, New MASS achieves better performance on depth-2 queries, suggesting that MASS Java has some advantages for deeper traversal queries but still requires further optimization to compete with Neo4j.

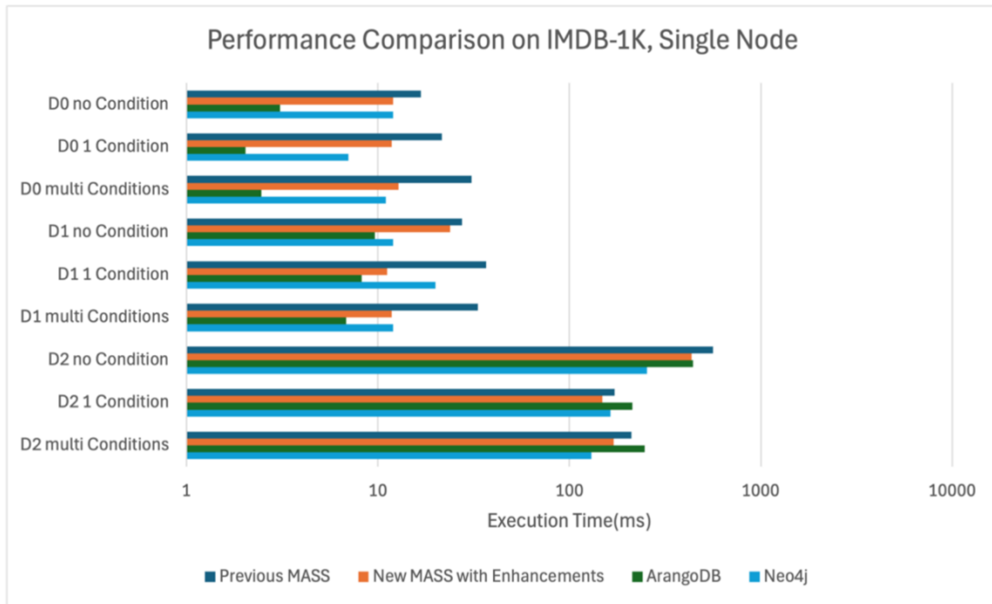

Figure 29 Performance Comparison of IMDB Queries on Single Node

The multi-node results show a different trend. As shown in Figure 30, MASS Java outperforms ArangoDB on both depth-1 and depth-2 queries.

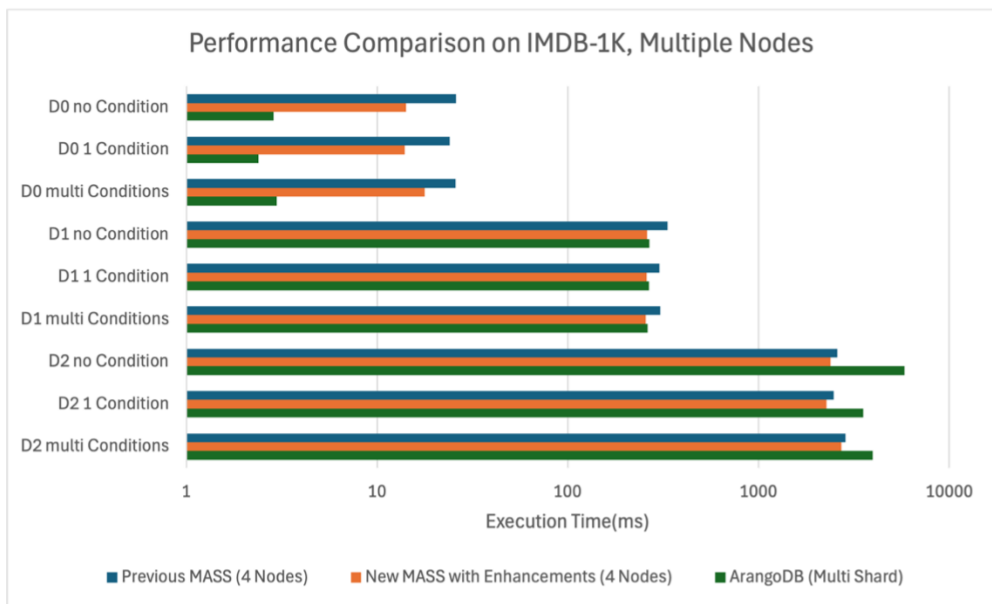

Figure 30 Parallel Performance Comparison of IMDB Queries on Multiple Nodes

Figures 31 and 32 show the query execution performance on the Shipment dataset. In the single-node setting, MASS Java performs worse than both ArangoDB and Neo4j. However, in the multi-node setting, MASS Java outperforms ArangoDB on depth-2 and depth-3 queries. In

addition, New MASS with enhancements is faster than Previous MASS in most cases, indicating that the proposed enhancements are generally effective for GraphDB query execution on this dataset.

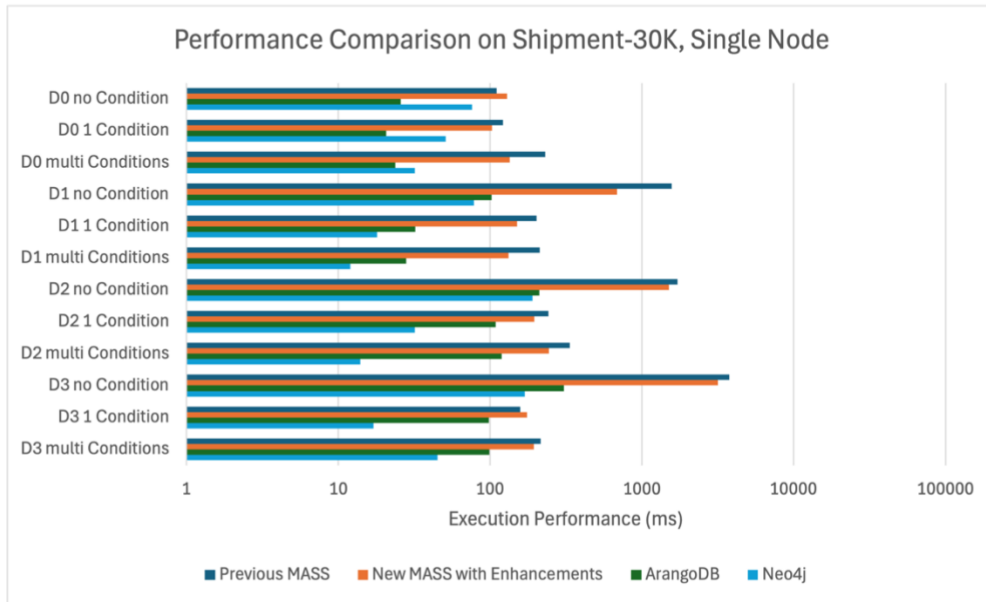

Figure 31 Performance Comparison of Shipment Queries on Single Node

Figure 32 Parallel Performance Comparison of Shipment Queries on Multiple Nodes

5.2.6 *Further Performance Improvement*

This subsection further investigates a major performance bottleneck observed in the GraphDB evaluation results and evaluates a possible optimization method. As shown in Section 5.2.5, the current MASS Java implementation runs much slower in the multi-node setting than in the single-node setting for non-zero-depth queries. This research finds that this issue is strongly related to the distributed HashMap implementation introduced in Chapter 2. To verify this explanation and explore a possible solution, this subsection replaces the distributed HashMap with a replicated HashMap and compares the GraphDB query performance before and after this change.

The distributed HashMap in MASS Java is mainly used to store the mapping from string keys in the graph database to the indices used by MASS Java. For non-zero-depth queries in GraphDB, agents need to migrate from the initial vertices to their neighboring vertices. To perform this migration, the system needs to look up the corresponding vertex index using the neighbor vertex name stored in each relationship list. Therefore, the query execution process needs to call *distributedMap.get()* repeatedly.

In a single query, this lookup operation may be called thousands of times. Since each *distributedMap.get()* may require access to a remote node, these lookups introduce significant communication overhead. This overhead can become large enough to offset the benefit of using multiple nodes, which explains the previous observation.

To explore a possible solution, this research replaces the distributed HashMap with a replicated HashMap. The replicated HashMap stores a full copy of all key-value pairs on every node in the distributed environment. Therefore, the *get()* operation can be performed locally without remote communication, reducing the communication overhead caused by frequent lookups.

Figure 33 compares the GraphDB query performance of MASS Java between the distributed HashMap implementation and the replicated HashMap implementation when running on 4 nodes. The result shows that, for non-zero-depth queries, the replicated HashMap reduces the execution time by more than 90%. This result indicates that the current distributed HashMap implementation introduces a major overhead in GraphDB query execution and should be optimized.

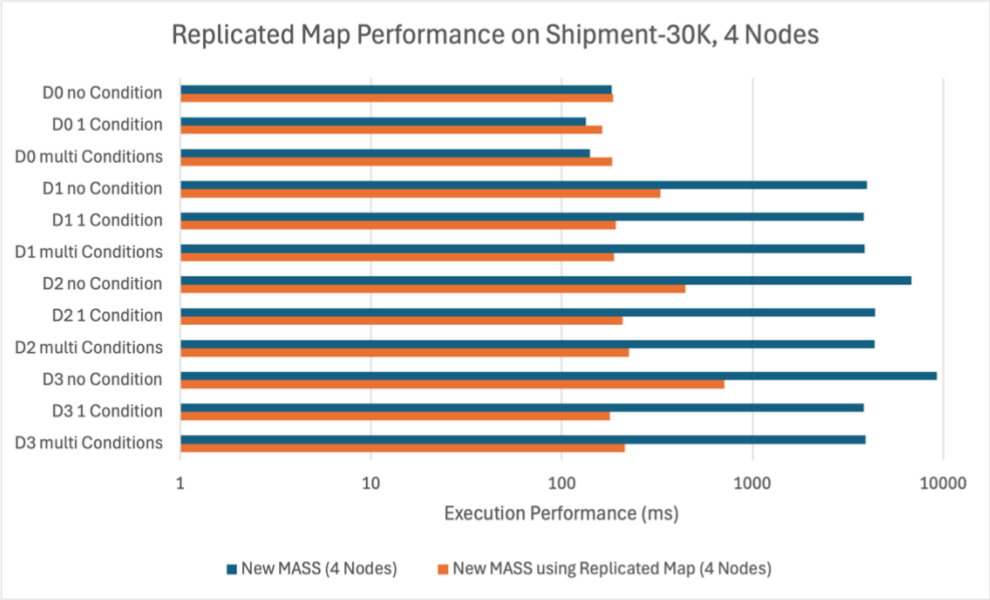

Figure 33 Parallel Performance of Shipment Queries using Replicated Map on 4 Nodes

However, the replicated HashMap also increases memory usage because every node stores a full copy of the mapping. In addition, if the graph database is updated frequently, maintaining consistency among replicated maps may introduce extra synchronization overhead. Therefore, further discussion and work are still needed to reduce overhead caused by the distributed HashMap while controlling memory usage and consistency cost.

Chapter 6. CONCLUSION

This chapter concludes the thesis by summarizing its key contributions, discussing its limitations, and outlining potential directions for future work.

6.1 CONTRIBUTIONS

This research contributes to the improvement and evaluation of MASS Java's parallel performance in the following ways. First, it implements and explores enhancements on top of the previous MASS Java implementation, including agent allocation enhancement, GraphPlaces multithreading, Hazelcast-inspired graph partitioning, agent initialization control, asynchronous migration and data exchange, and lambda-based reductive computation. These methods can improve MASS Java by either increasing single-node parallelism or reducing communication overhead in multi-node environments.

Second, this research evaluates these enhancements using representative MASS Java applications, including Triangle Counting (TC), Breadth-First Search (BFS), Ant Colony Optimization-based Traveling Salesman Problem (ACO-based TSP), and 2-Dimensional Wave Simulation (Wave2D). The evaluation results show that the proposed methods improve the execution performance of MASS Java under specific application scenarios. This research also discusses possible reasons why some methods do not lead to obvious performance improvements in certain cases.

Third, this research benchmarks the MASS graph database using two datasets and compares its performance with mainstream graph database systems, including ArangoDB and Neo4j. The evaluation results show that GraphPlaces multithreading and agent initialization control provide relatively clear performance improvements, while the improvements from asynchronous migration

and Hazelcast-inspired graph partitioning are limited. The comparison results also show that the MASS graph database can outperform ArangoDB in some cases, especially in multi-node environments, although it remains slower than Neo4j in single-node execution.

Finally, this research identifies the communication overhead caused by the distributed HashMap as a major performance limitation in the current MASS Java graph database implementation. The evaluation of a replicated HashMap shows that reducing this overhead can significantly improve non-zero-depth query performance in multi-node environments.

6.2 LIMITATIONS

Despite these contributions, several limitations remain in both the proposed methods and the MASS Java implementation. First, the proposed asynchronous migration method still depends on synchronization at the end of each iteration. As a result, it does not fully eliminate global synchronization and cannot provide a completely asynchronous execution model, leading to limited performance improvements.

Second, the Hazelcast-inspired graph partitioning method relies on a fixed partition size. This approach is not flexible and may provide limited performance improvement when the graph does not have a suitable partitioning pattern. In some cases, it may even introduce additional overhead and reduce performance.

Third, the distributed HashMap remains a major limitation in the current MASS Java graph database implementation. As discussed in Section 5.2.6, the current implementation introduces significant communication overhead during graph database query execution, which can offset the benefits of multi-node parallelization.

6.3 Future Work

Future work can further address the limitations identified in this thesis. First, a more efficient mapping mechanism can be designed and implemented for the MASS Java graph database. Possible directions include improving the replicated HashMap approach, adding caching mechanisms, or optimizing the current distributed HashMap implementation. This could help reduce the communication overhead caused by the current distributed HashMap.

Second, the asynchronous migration feature can be further improved. The current implementation still relies on master-worker synchronization at the end of each iteration. Future work could explore a peer-to-peer synchronization mechanism, similar to the one used in *Places.exchangeAll()*, to reduce the synchronization overhead between the master node and worker nodes. This change may make asynchronous migration more efficient while still preserving the correctness of agent migration across nodes.

Third, future work can implement smarter graph partitioning algorithms. Instead of using a fixed partition size, future methods could partition graphs dynamically based on the graph structure, vertex connectivity, and runtime communication patterns. This may help reduce communication and migration overhead, thereby further improving the parallel performance of MASS Java.

BIBLIOGRAPHY

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” *2nd USENIX Workshop Hot Top. Cloud Comput. HotCloud 10*.

[2] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun ACM*, vol. 51, no. 1, pp. 107–113, 2008, doi: 10.1145/1327452.1327492.

[3] W. Dubitzky, K. Kurowski, and B. Schott, “Repast HPC: A Platform for Large-Scale Agent-Based Modeling,” in *Large-Scale Computing Techniques for Complex System Simulations*, IEEE, 2012, pp. 81–109. doi: 10.1002/9781118130506.ch5.

[4] M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, and C. Greenough, “FLAME: simulating large populations of agents on parallel hardware architectures,” in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, in AAMAS ’10. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2010, pp. 1633–1636. Accessed: Apr. 20, 2026. [Online]. Available: https://dl.acm.org/doi/10.5555/1838206.1838517

[5] T. Chuang and M. Fukuda, “A Parallel Multi-agent Spatial Simulation Environment for Cluster Systems,” in *2013 IEEE 16th International Conference on Computational Science and Engineering*, Sydney, Australia: IEEE, Dec. 2013, pp. 143–150. doi: 10.1109/CSE.2013.32.

[6] J. Gilroy, S. Paronyan, J. Acoltzi, and M. Fukuda, “Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory,” in *2020 IEEE International Conference on Big Data (Big Data)*, Atlanta, GA, USA: IEEE, Dec. 2020, pp. 2957–2966. doi: 10.1109/BigData50022.2020.9378298.

[7] Y. Ma, M. Dea, L. Cao, and M. Fukuda, “Toward Implementing an Agent-based Distributed Graph Database System,” in *2024 IEEE International Conference on Big Data (BigData)*,

Washington, DC, USA: IEEE, Dec. 2024, pp. 3456–3465. doi: 10.1109/BigData62323.2024.10825052.

[8] A. R. Prajapati, “An Enhancement of Distributed Graph Queries in an Agent-Based Graph Database”, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/AatmanPrajapati_thesis.pdf

[9] N. Beraki, “Comparison of Distributed Graph Computing Performance Between Java MASS and Hazelcast,” 2025, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/NoelBeraki_su25.pdf

[10] R. Zimmerman, “MASS Java & GraphX Performance Benchmarking for Graph Computing”, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/RobertZimmerman_au25.pdf

[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “GraphX: Graph Processing in a Distributed Dataflow Framework,” *11th USENIX Symp. Oper. Syst. Des. Implement. OSDI 14*, pp. 599–613, 2014.

[12] “ArangoDB Documentation: About arangodb.” Accessed: May 05, 2026. [Online]. Available: https://docs.arango.ai/arangodb/

[13] “Neo4j Documentation: What is neo4j?,” Neo4j Graph Data Platform. Accessed: May 05, 2026. [Online]. Available: https://neo4j.com/docs/getting-started/whats-neo4j/

[14] A. Ahire, “MASS Java Library towards Its Use for a Graph Database System”, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/AtulAhire_whitepaper.pdf

[15] C. M. Macal and M. J. North, “Agent-based modeling and simulation,” in *Proceedings of the 2009 Winter Simulation Conference (WSC)*, Dec. 2009, pp. 86–98. doi: 10.1109/WSC.2009.5429318.

[16] “Repast Suite Documentation.” Accessed: Apr. 20, 2026. [Online]. Available: https://repast.github.io/

[17] “FLAME Overview.” Accessed: Apr. 20, 2026. [Online]. Available: https://flame.ac.uk/docs/overview.html

[18] S. Coakley, M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough, “Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework,” Jun. 2012. doi: 10.1109/HPCC.2012.79.

[19] A. Moreno, J. J. Rodríguez, D. Beltrán, A. Sikora, J. Jorba, and E. César, “Designing a benchmark for the performance evaluation of agent-based simulation applications on HPC,” *J. Supercomput.*, vol. 75, no. 3, pp. 1524–1550, Mar. 2019, doi: 10.1007/s11227-018-2688-8.

[20] E. Gabriel *et al.*, “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., Berlin, Heidelberg: Springer, 2004, pp. 97–104. doi: 10.1007/978-3-540-30218-6_19.

[21] J. Corbalan, A. Duran, and J. Labarta, “Dynamic load balancing of MPI+OpenMP applications,” in *International Conference on Parallel Processing, 2004. ICPP 2004.*, Aug. 2004, pp. 195–202 vol.1. doi: 10.1109/ICPP.2004.1327921.

[22] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, “GraphLab: A New Framework For Parallel Machine Learning,” Aug. 09, 2014, *arXiv*: arXiv:1408.2041. doi: 10.48550/arXiv.1408.2041.

[23] “Data Partitioning and Replication.” Accessed: Apr. 20, 2026. [Online]. Available: https://docs.hazelcast.com/hazelcast/5.4/architecture/data-partitioning

APPENDIX A LISTINGS

```
/* Abstracted Local Agent Bag Initialization */

initializeAgentQueue(PlacesBase placesBase, int tid) {
    placeRange = getLocalRange(tid)
    for index in placeRange:
        if(places.get(index) != null):
            for(Agent a in places.get(index).getAgents()):
                localAgentQueue.add(a);
}
```

Listing 1 Local Agent Bag Initialization

```
/* Abstracted Local Agent Bag Based Agent Retrival*/

Agent curAgent = MThread.pollNextAgent();
if(curAgent != null):
    //Execute, manage or perform any other operations on this Agent
else:
    //Current Thread has no Agent to operate
    //break or return
```

Listing 2 Local Agent Bag Based Execution

```
/* Abstracted Local Agent Bag Based Agent Removal*/

if(curAgent.isMigrated || curAgent.isKilled):
    PlaceBase oldPlace = curAgent.getPlace()
    oldPlace.getAgents().remove(curAgent)
```

Listing 3 Local Agent Removal

```
/* Abstracted GraphPlaces Multithreading CallAll Logic */

callAll( int functionId, Object argument, int tid ) {
    placeRange = getLocalRange(tid)

    for index in placeRange:
        if(places.get(index) != null):
            places.get(index).callMethod(functionId, argument)
}
```

Listing 4 Multithreaded GraphPlaces callAll()

```
/* Abstracted Mapping Between VertexID and OwnerID*/

getOwnerID(vertexID):
    partitionID = vertexID / PARTITION_SIZE
    ownerID = partitionID % NUM_NODES
    return ownerID
```

Listing 5 Hazelcast-inspired Mapping Between VertexID and OwnerID

```
/* Abstracted per-place initialization */

for each place in places {

    flag = place.callMethod(functionId, callArgument)

    if flag is true:
        create one or more agents at this place
    else:
        skip this place

}
```

Listing 6 Agent Initialization Check on Places

```
/* Abstracted Agent Initialization Logic */

Agents( ..., functionId, callArgument ) {

    //Start initialization on local machine
    super( ..., functionId, callArgument )
    localAgents = new array[sizeOfSystem]

    // Send initialization message to remote machines
    message = createInitializationMessage( ..., functionId, callArgument)
    for each remoteNode in remoteNodes {
        remoteNode.send(message)
    }

    barrierAllSlaves(localAgents)
}
```

Listing 7 Agent Constructor with Initialization Control

```
/* Abstracted Asynchronous Migration */

doWhile(functionId, argument) {
    MASSBase.setParams(functionId, argument);
    sendMessages(DO_WHILE_START, remoteNodes, functionId, argument);

    while(true){
        Agents.callAll(functionId, argument);
        Agents.manageAll();
        BarrierWithRemoteMachines(); //Make sure all Agents have been migrated
        checkAgentNumber();
        if(noAliveAgent)
            break;
        else
            sendMessages(DO_WHILE_CONTINUE);
    }

    sendMessages(DO_WHILE_END);

    BarrierWithRemoteMachines();
}
```

Listing 8 Asynchronous Migration Logic on Master Machine

```
/* Abstracted Asynchronous Migration on Remote Machines */

AgentDoWhileProcessing(messageType) {
    if messageType == AGENTS_DO_WHILE_START:
        setupExecutionContext()

        callAll()
        manageAll()
        count = checkExecutableAgents()

        sendAck(max(count, 0))

    else if messageType == AGENTS_DO_WHILE_CONTINUE:

        callAll()
        manageAll()
        count = checkExecutableAgents()

        sendAck(max(count, 0))

    else if messageType == AGENTS_DO_WHILE_END:

        resumeAllAgents()
        sendAck(localPopulation)
}
```

Listing 9 Asynchronous Migration Logic on Worker Machines

```
/* Abstracted Asynchronous Data Exchange */

doAll(functionId, argument, iterations) {
    MASSBase.setParams(functionId, argument);
    sendMessages(DO_ALL_START, remoteNodes, functionId, argument, iterations);

    for iterations {
        Places.callAll(functionId, argument);
        Places.exchangeBoundary(); //Built-in synchronization with neighbours
    }

    BarrierWithRemoteMachines();
}
```

Listing 10 Asynchronous Data Exchange Logic

```
/* Abstracted reduce workflow */

reduce(mapper, reducer) {

   // distribute reduction request to remote nodes
   for each remoteNode in remoteNodes {
       message = createReduceMessage(..., mapper, reducer);
       remoteNode.send(message)
   }

   // collect partial results from all nodes
   partialResults = new array[sizeOfSystem]
   partialResults[0] = localReduce(mapper, reducer)

   for each remoteNode in remoteNodes {
       response = remoteNode.getMessage()
       partialResults[remoteNode.pid] = response
   }

   // merge partial results using the reducer
   result = null

   for each partial in partialResults {
       if result is null:
           result = partial
       else if partial is not null:
           result = reducer.apply(result, partial)
   }

   return result

}
```

Listing 11 Lambda-based Reduction on the Master Node

```
/* Abstracted node-level reduction logic */

localReduce(mapper, reducer) {

   initializeSharedVariablesForThreads(mapper, reducer)
   setThreadPartialReturns(new array[threadNum])

   MThread.resumeThreads(reduceMode)
   localReduce(mapper, reducer, mainThreadId)
   MThread.waitForAllThreads()

   result = null

   for each partial in threadPartialResults {
       if result is null:
           result = partial
       else if partial is not null:
           result = reducer.apply(result, partial)
   }

   return result

}
```

Listing 12 Lambda-based Reduction at Node Level

```
/* Abstracted thread-level reduction logic */

localReduce(mapper, reducer, tid) {

   MThread.initAgentQueue(tid)
   result = null

   while true {
       agent = MThread.pullNextAgent();
       if agent is null:
           break

       value = mapper.apply(agent)
       if value is null:
           continue

       if result is null:
           result = value
       else:
           result = reducer.apply(result, value)
   }

   threadPartialResults[tid] = result
   synchronizeThreads(tid)

}
```

Listing 13 Lambda-based Reduction at Thread Level