Enhancing Agent Execution Performance in MASS Java Library

Bo Fu

Term Report of Work Done

At 2025 Autumn Quarter

Master of Science in Computer Science & Software Engineering

University of Washington

12/10/2025

Thesis Committee:

Prof. Munehiro Fukuda, Committee Chair

Prof. Min Chen, Committee Member

Prof. Robert Dimpsey, Committee Member

I. Introduction

The Multi-Agent Spatial Simulation (MASS) Java library is a distributed simulation framework that supports agent-based modeling over spatial structures such as arrays and graphs. The framework separates two core components: Places, which represent distributed data structures, and Agents, which are mobile entities that operate across Places. This architecture allows the system to simulate complex, parallel activities across distributed environments.

Recent studies [1][2] have shown that MASS Java has potential as an agent-based graph database system, although its performance remains limited in certain scenarios. These limitations are mainly due to the lack of optimization for graphs on agent migration and execution performance.

To address these limitations, the thesis will introduce enhancements on Agent initialization control, lambda expression support and asynchronous agent migration. These Agent performance improvements will ultimately practicalize agent-based graph computing and DB systems by making them more efficient. And the effectiveness of these improvements will be evaluated with benchmark programs for graph computing and graph databases.

II. Motivation

The original version of the MASS Java library introduced agent-based modeling over distributed environments, enabling Agents to dynamically navigate graphs across multiple processes [3]. Later, support for automated Agent migration over distributed data structures was added, allowing Agents to move between nodes without manual coordination [4]. More recently, Place-level execution has been improved through METIS-based graph partitioning and Aeron-based messaging [5].

Building on these developments, Yuan Ma [1] evaluated MASS Java as a graph database and benchmarked its performance against major graph database systems, while Sumit Hotchandani [2] explored agent-based link prediction using the platform. Although these programs demonstrate the potential of MASS Java as a graph database system, performance remains constrained due to limited optimization of agent execution, particularly in initialization, migration, and reductive computation. The current function invocation model also requires substantial boilerplate to express agent operations, which may reduce overall programmability.

This thesis research focuses on improving the programmability and execution performance of agents in MASS Java, addressing limitations to practicalize agent-based graph computing and database systems. One major limitation is that agents are instantiated on every graph vertex during queries in an agent-based graph DB system, which introduces unnecessary overhead and degrades query performance. To address it, this quarter’s work introduces a new initialization API that allows agents to be deployed exactly on graph vertices that meet their initial queries.

Besides, because the MASS Java library lacked reduction support, developers had to gather all Agents or Places on the main node and perform the computation manually, such as finding the maximum value. This approach introduced significant communication and execution overhead and did not take advantage of the parallel capabilities of remote nodes. To solve this problem, this quarter's work implements reduction methods based on lambda expressions, which can both improve programmability by reducing boilerplate code and enhance execution performance through parallel reduction.

III. Design

During this quarter, the work of this thesis primarily focused on two goals: Agent initialization control and the lambda expression based reduction. The following sections describe the design rationale and how they help improve the execution performance of MASS Java framework.

A. Agent Initialization Control

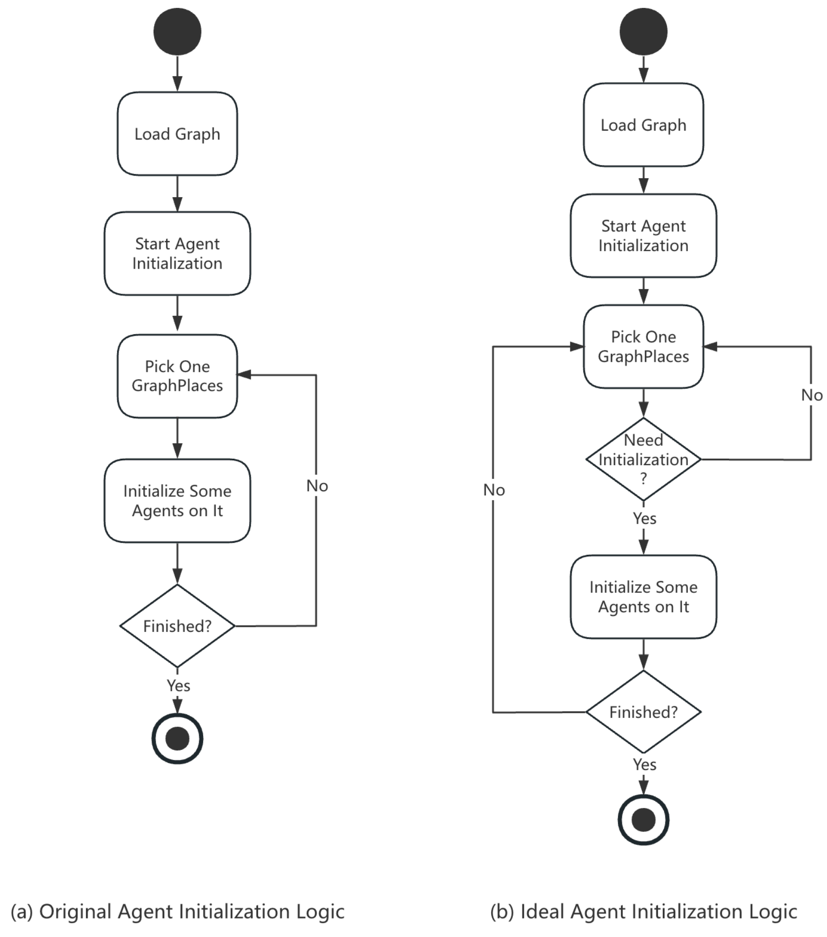
Load Graph
Start Agent Initialization
Pick One GraphPlaces
Initialize Some Agents on It
Finished?
No
Yes
Load Graph
Start Agent Initialization
Pick One GraphPlaces
Need Initialization ?
No
Yes
Initialize Some Agents on It
Finished?
No
Yes
(a) Original Agent Initialization Logic
(b) Ideal Agent Initialization Logic

Figure 1. Agent Initialization Control Diagram

As shown in Figure 1, when Agents are initialized, the MASS Java Library populates them on the vertices of the graph (known as GraphPlaces). In the original implementation, each GraphPlace is assigned at least one Agent, and these Agents then participate in subsequent execution and migration. However, in certain applications like GraphDB query, not all GraphPlaces require Agents. In such cases, introducing a conditional check for each GraphPlace allows the system to determine whether initialization is necessary, thereby reducing both the Initialization cost and the overhead incurred during later execution.

B. Reduction computation based on lambda expressions

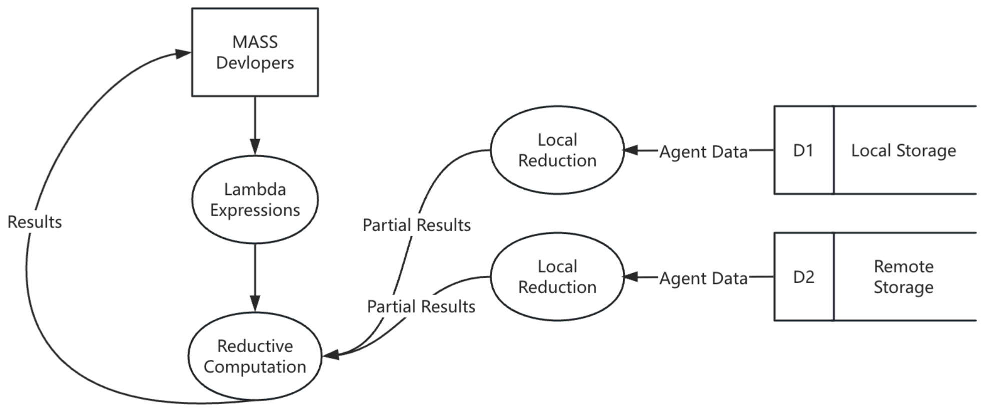

Figure 2 Execution Result Dataflow with Lambda Expressions

As discussed earlier, the original MASS Java library required collecting all Agents or Places on the main node to perform reductions, which introduced communication overhead and prevented parallel computation. A better approach is shown in Figure 2. The reduction logic is expressed as a lambda expression and distributed to remote nodes, where each node performs the computation locally and produces a partial result. These partial results are then aggregated by the main node to obtain the final outcome. For applications that require frequent reductions, such as ACO-based TSP (Ant Colony Optimization based Traveling Salesman Problem), this approach can significantly improve execution performance. This approach also enhances programmability by allowing developers to express reduction operations with lambda expressions rather than manually handling data collection and aggregation.

IV. Implementation

A. Agent Initialization Control

1. Conditional Check in AgentBase.java

To allow application developers to control where agents are initialized, this work introduces a conditional check in the constructor of Agents. To stay consistent with the design of the MASS Java library and allow flexible customization, the implementation uses the existing functionId based mechanism to invoke developer-defined functions. Based on the return value of the function called on each Place, the system determines whether an agent should be initialized at that location.

Listing 1 is an abstracted view of the initialization workflow, showing how the system selects the appropriate initialization path based on the underlying Places type (lines 6-12). The full source code could be found in Appendix.

```
 1  /* Abstracted initialization logic */
 2
 3  AgentsBase( ... , functionId, callArgument ) {
 4
 5      // Identify the type of Places (regular places, graph places, or property graph
            places)
 6      if (currentPlaces is PropertyGraphPlaces) {
 7          initializeForPropertyGraph(...);
 8      } else if (currentPlaces is GraphPlaces) {
 9          initializeForGraph(...);
10      } else {
11          initializeForRegularPlaces(...);
12      }
13
14  }
```

Listing 1 Agent Initialization Based on Place Types

Listing 2 shows the conditional check logic. Each Place invokes a customized method through callMethod() using a functionId (line 5) and obtains its return value. In MASS Java, callMethod() is the mechanism for invoking application developer-defined functions. Application developers override this method in their customized Place class, allowing the framework to dispatch and execute the developer-defined method based on the given functionId. Based on this return value, Agents are populated on that Place if the result is true (lines 7–10).

```
 1  /* Abstracted per-place initialization */
 2
 3  for each place in places {
 4
 5      flag = place.callMethod(functionId, callArgument)
 6
 7      if flag is true:
 8          create one or more agents at this place
 9      else:
10          skip this place
11
12  }
```

Listing 2 Agent Initialization on Each Place

2. *Agent Constructor and Communication in Agents.java*

Listing 3 shows the constructor of Agents, which is directly invoked by application developers. It calls the constructor of AgentBase and then invokes the message-sending function (lines 5-7).

```
/* Abstracted constructor logic */

Agents( ..., functionId, callArgument ) {

   super( ..., functionId, callArgument )
   localAgents = new array[sizeOfSystem]
   initializeOnMaster(functionId, callArgument)

}
```

Listing 3 Agents Constructor

Listing 4 illustrates the message synchronization process during the construction of Agents. As MASS Java operates in a distributed environment where graph vertices are partitioned across multiple machines, the condition used for agent initialization must also be distributed. Therefore, the constructor constructs the initialization message, sends it to all nodes, and synchronizes the execution (lines 6–14), ensuring that the developer-defined conditional check is applied consistently across the entire system.

```
 1  /* Abstracted master-side initialization logic */
 2
 3  initializeOnMaster(functionId, callArgument) {
 4
 5     // prepare initialization message containing functionId and callArgument
 6     message = createInitializationMessage( ..., functionId, callArgument)
 7
 8     // send initialization message to all remote nodes
 9     for each remoteNode in remoteNodes {
10         remoteNode.send(message)
11     }
12
13     // synchronize with all slave nodes
14     barrierAllSlaves(localAgents)
15
16     // record local population and register this Agents instance
17     localAgents[0] = getLocalPopulation()
18     registerAgentsInstance(this)
19
20  }
```

Listing 4 Message Synchronization During Agents Construction

The modifications also include changes to constructors of Agents for PropertyGraphPlaces and Space. However, since the underlying principles are similar, they are not elaborated here, and the source code can be found in the Appendix.

B. Reduction computation based on lambda expressions

To address previous limitations about reduction, the new implementation expresses the reduction logic as a lambda expression. The lambda expression is then distributed to all nodes, allowing each node to execute the reduction locally and produce a partial result. These partial results are finally gathered and aggregated on the main node to obtain the global outcome.

1. *Functional Interface Definition*

The lambda expressions are implemented as two serializable functional interfaces:

- SerializableFunction serves as the mapper, transforming each local element into an intermediate value to be aggregated later.
- SerializableBinaryOperator serves as the reducer, combining two intermediate values into one so that partial results from different nodes can be merged into a final result.

2. *Reduction Method for Main Node in AgentBase.java*

Listing 5 shows how the main node initiates a reduction operation. It first receives the customized lambda expression from the application (line 3), then distributes the corresponding message to all nodes (lines 6-9). Next, it triggers the local reduction on the local machine (lines 12-18). Finally, it collects all partial results from the nodes and aggregates them (lines 23 to 30).

```
 1  /* Abstracted reduce workflow */
 2
 3  reduce(mapper, reducer) {
 4
 5     // distribute reduction request to remote nodes
 6     for each remoteNode in remoteNodes {
 7         message = createReduceMessage(..., mapper, reducer);
 8         remoteNode.send(message)
 9     }
10
11     // collect partial results from all nodes
12     partialResults = new array[sizeOfSystem]
13     partialResults[0] = localReduce(mapper, reducer)
14
15     for each remoteNode in remoteNodes {
16         response = remoteNode.getMessage()
17         partialResults[remoteNode.pid] = response
18     }
19
20     // merge partial results using the reducer
21     result = null
22
23     for each partial in partialResults {
24         if result is null:
25             result = partial
26         else if partial is not null:
27             result = reducer.apply(result, partial)
28     }
29
30     return result
31
32  }
```

Listing 5 Reduce Logic on Main Node

3. Reduction Methods for Every Node in AgentBase.java

Listing 6 illustrates the local reduction process within a single node using multiple threads. Shared variables and thread local storage are first initialized (lines 5-6). All worker threads are then resumed in reduction mode, and the main thread also participates in the computation (lines 8-9). After thread synchronization (line 10), partial results are aggregated using the reducer function to produce the final result (lines 14-19), which is returned to the caller (line 21).

```
 1  /* Abstracted node-level reduction logic */
 2
 3  localReduce(mapper, reducer) {
 4
 5      initializeSharedVariablesForThreads(mapper, reducer)
 6      setThreadPartialReturns(new array[threadNum])
 7
 8      MThread.resumeThreads(reduceMode)
 9      localReduce(mapper, reducer, mainThreadId)
10      MThread.waitForAllThreads()
11
12      result = null
13
14      for each partial in threadPartialResults {
15          if result is null:
16              result = partial
17          else if partial is not null:
18              result = reducer.apply(result, partial)
19      }
20
21      return result
22
23  }
```

Listing 6 Reduce Logic on Every Node

Listing 7 illustrates the abstracted thread-level reduction logic executed by each worker thread. Each thread initializes its local state and iteratively processes Agents from the shared queue (lines 5–11). Then, the mapper function is applied to each Agent and valid results are combined using the reducer function (lines 13–20). After all assigned Agents are processed, the thread's partial result is stored in the shared array (line 23), which will be aggregated later on the main thread.

```
/* 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 7 Reduce Logic on Every Thread

4. *Examples and Simplified Reduction Methods*

This work also added several higher level utility methods to simplify common reduction tasks. While the core interface requires developers to provide both a mapper and a reducer lambda expression, some frequent operations like counting can be expressed more concisely. These utility methods serve both as conveniences and as examples for application developers.

- collectBest selects the best Agent according to a developer defined binary operator, which determines how two candidates are compared during reduction.
- collectTopK retrieves the top K Agents according to a developer provided comparison operator, returning a ranked subset of Agents.
- count performs a distributed counting operation by aggregating the number of Agents across all nodes.

The modifications also include reduction methods for Place-level classes. However, since the underlying principles are similar, they are not elaborated here, and the source code can be found in the Appendix.

V. Expected Results and Evaluation Plan

Due to time constraints, this quarter's work does not include an evaluation of the impact on execution performance. Based on preliminary expectations, the lambda expression based reduction is likely to provide substantial performance improvements, especially for applications that require frequent reductions such as ACO based TSP. The benefit of initialization control depends on the specific application scenario. It may also provide noticeable performance improvements if it can be leveraged by queries in GraphDB.

As shown in Table 1, the evaluation of the work presented in this thesis will be carried out during the Winter and Spring quarters, with the features developed during the independent study also included as part of the comprehensive assessment. The evaluation includes benchmarking and programmability comparisons between the improved and original versions of the MASS Java library, as well as comparisons against other mainstream libraries.

Table 1 Future Evaluation Plan

Quarter	Weeks	Evaluation Plan	Notes
Winter 2026	Week 7-9	Benchmarking and analysing the programmability improvements using existing datasets and applications.	Benchmark result report
Spring 2026	Week 1-2	Find mainstream libraries to compare with and appropriate program	
Spring 2026	Week 3-4	Compare performance with current MASS Java and mainstream libraries.	
Spring 2026	Week 5-6	Finish performance comparison.	Comparison report

VI. Conclusions

This quarter’s work introduces a more flexible and customizable mechanism for Agent initialization control. By allowing developers to specify initialization conditions through customized functions, the system can avoid unnecessary agent creation and reduce both initialization and subsequent execution overhead. In addition, this work extends the MASS Java library with support for lambda expression based reduction, which can significantly improve the efficiency of reduction operations.

Despite these improvements, the current work has certain limitations like the lack of comprehensive benchmarking. Further evaluation will be applied in the next phase to fully assess the performance impact of these enhancements. The evaluation will also include the features implemented during the independent study, such as the multi threaded execution of Agents, the Hazelcast inspired graph partitioning, and the lambda expression based callAll functions.

References

[1] Yuan Ma, Michelle Dea, Lillian Cao, and Munehiro Fukuda, "Toward Implementing an Agent-based Distributed Graph Database System", The 4th Workshop on Knowledge Graphs and Big Data In conjunction with the IEEE Big Data 2024, pages 3456-3465, December 15-18, 2024

[2] S. Hotchandani, “Agent-based Link Prediction in Graph Database,” 2025. Accessed: Jul. 17, 2025. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/SumitHotchandani_wi25.pdf

[3] J. Gilroy, S. Paronyan, J. Acoltzi, and M. Fukuda, “Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory,” in Proc. of the IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 2020, pp. 2957–2966.

[4] V. Mohan, A. Potturi, and M. Fukuda, “Automated Agent Migration over Distributed Data Structures,” in Proc. of the 15th International Conference on Agents and Artificial Intelligence (ICAART), Lisbon, Portugal, 2023, pp. 363–371.

[5] A. Ahire, “MASS JAVA LIBRARY TOWARDS ITS USE FOR A GRAPH DATABASE SYSTEM,” Mar. 2025. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/AtulAhire_wi25.pdf

Appendix

Complete source code of this implementation can be found in the branch:

mass_library_developers/mass_java_core/fuchacha/agent-initialization

mass_library_developers/mass_java_core/fuchacha/lambda