Enhancing Agent Execution Performance in MASS Java Library

Bo Fu

Term Report of Work Done as Independent Study

At 2025 Summer Quarter

Master of Science in Computer Science & Software Engineering

University of Washington

08/22/2025

Supervisor:

Prof. Munehiro Fukuda

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 execution performance. Besides, though previous work has explored the multi-nodes optimizations of MASS Java, the multithreading part has remained unoptimized.

To address these limitations, the independent study introduces enhancements on multithreading logic of GraphPlaces and Agents. Specifically, the study is aimed to implement a better allocation algorithm of Agents, which groups near Agents on the same thread to maintain good CPU affinity and efficient cache usage.

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. However, most of the previous work has focused on optimizations across multiple nodes, while MASS Java still lacks proper optimization for multi-threading, especially in Agents and GraphPlaces. This Independent Study is mainly motivated by the goal of improving overall performance of MASS Java through multi-threading enhancements, and ultimately increasing its usability as a graph database.

These improvements are evaluated using the benchmark program TriangleCounting on datasets ranging from 300 to 3,000 nodes to assess scalability and runtime efficiency.

III. Implementation

A. Places Multithreading Improvements

Previously, when initializing MASS Java with multiple threads and using GraphPlaces, the process would enter a deadlock. The cause of this issue was that the required variables used by

worker threads were not set correctly. In addition, there were also some logical errors that prevented the execution of the process. Here is an improved version of GraphPlaces with corrected multithreading configuration and logic.

1. *Variable Initialization in GraphPlaces.java*

This method is called by the main thread, so it is important to set variables like PlacesBase, FunctionId, which are required by the MThread class. And after setting these variables, the process will allocate GraphPlaces to threads and execute methods on them.

```
@Override
public void callAll(int functionId, Object argument) {
    //Send messages to all secondary processes
    MASS.getRemoteNodes().forEach(node -> node.sendMessage(new Message(
        Message.ACTION_TYPE.GRAPH_PLACES_CALL_ALL_VOID_OBJECT,
        this.getHandle(),
        functionId,
        argument
    )));

    //Setup messages for multithreading
    MASSBase.setCurrentPlacesBase(this);
    MASSBase.setCurrentFunctionId(functionId);
    MASSBase.setCurrentArgument(argument);
    MASSBase.setCurrentMsgType(Message.ACTION_TYPE.GRAPH_PLACES_CALL_ALL_VOID_OBJECT);
    MThread.resumeThreads(MThread.STATUS_TYPE.STATUS_CALLALL);

    //Execute the actual call on main thread
    this.callAll(functionId, argument, 0);
    MThread.barrierThreads( 0 );

    // Synchronize with all secondary processes
    MASSBase.getLogger().debug("Attempting to barrierAllSlaves...");
    MASS.barrierAllSlaves();
    MASSBase.getLogger().debug("barrierAllSlaves completed!");
}
```

2. *Place Allocation and Execution in GraphPlaces.java*

In this method, each thread is assigned a specific range of place indexes, and subsequently invokes the method only on the corresponding places. The allocation method “getLocalRange” is defined by PlaceBase class, which is used in the original 2D version Place.

```
@Override
public void callAll( int functionId, Object argument, int tid ) {
    //check and set local placesSize
    if (this.placesSize == 0)
        this.placesSize = places.size();

    if ( MASSBase.getLogger().isDebugEnabled() )
        MASS.getLogger().debug("Local PlacesSize = " + this.placesSize);

    // TODO: Allocate places in the same partitions divided by METIS to the same thread
    //Use the function from PlaceBase to arrange places to threads
    int[] range = new int[2];
    getLocalRange(range, tid);
```

```
        MASS.getLogger().debug("local tid = " + tid + " range from " + range[0] + " to " +
range[1]);

        // debugging
        if ( MASSBase.getLogger().isDebugEnabled() )
            MASSBase.getLogger().debug( "thread[" + tid + "] callAll functionId = " +
                    functionId + ", range[0] = " + range[0] +
                    " range[1] = " + range[1] );

        //use the same logic as PlaceBase
        if ( range[0] >= 0 && range[1] >= 0 ) {

            for ( int i = range[0]; i <= range[1]; i++ ) {
                if (places.get(i) == null) { continue; }
                places.get(i).callMethod( functionId, argument );
            }

        }
    }
```

The modifications to Places-related classes also include the callAll method with return values in GraphPlaces, as well as changes to PropertyGraphPlaces (a variant of GraphPlaces). However, since the underlying principles are similar, they are not elaborated here, and the source code can be found in the Appendix.

B. Agent Multithreading Improvements

In the previous implementation of callAll and manageAll for Agents, all threads shared a single global Agent bag containing all Agents. Each thread retrieved Agents sequentially from this global bag, which introduced substantial synchronization overhead. Moreover, since Agents were randomly assigned, a single thread could end up processing Agents distributed across many different Places, resulting in poor data locality. A more efficient strategy is to allocate Agents residing in nearby Places to the same thread. Furthermore, to eliminate unnecessary synchronization, each thread should maintain its own local Agent bag.

1. Maintenance of Local Agent Bag in MThread.java

Each thread is required to maintain its own local Agent bag. This involves the initialization of the bag, clearing its contents when necessary, retrieving Agents from it, and supporting additional utility methods that facilitate efficient thread-level management.

```
private static ThreadLocal<Queue<Agent>> agentQueue =
ThreadLocal.withInitial(LinkedList::new);
    public static void initializeAgentQueue(PlacesBase placesBase, int tid) {
        clearAgentQueue();
        Queue<Agent> localAgentQueue = agentQueue.get();
        boolean isGraphPlaces = GraphPlaces.class.isAssignableFrom(placesBase.getClass());
        GraphPlaces graphPlaces = null;
        if(isGraphPlaces) {
            graphPlaces = (GraphPlaces) placesBase;
            if(placesBase.placesSize == 0)
                placesBase.placesSize = graphPlaces.places.size();
        }
        MASSBase.getLogger().debug( "Current placesSize is " + placesBase.placesSize);
```

```
    int[] range = new int[2];
    placesBase.getLocalRange(range, tid);

    if ( range[0] >= 0 && range[1] >= 0 ) {
        for ( int i = range[0]; i <= range[1]; i++ ) {
            if(isGraphPlaces) {
                if (graphPlaces.places.get(i) == null) { continue; }
                for(Agent agent : graphPlaces.places.get(i).getAgents()) {
                    localAgentQueue.add(agent);
                }
            } else {
                if(placesBase.places[i] == null) { continue; }
                for(Agent agent : placesBase.places[i].getAgents()) {
                    localAgentQueue.add(agent);
                }
            }
        }
    }
    barrierThreads(tid);
}

/*
 * clear agent queue
 */
public static void clearAgentQueue( ) {
    agentQueue.get().clear();
}

/*
 * get the full agent queue, should be only used for debugging
 */
public static Queue<Agent> getAgentQueue( ) {
    return agentQueue.get();
}

/*
 * get the agent queue's size
 */
public static int getAgentQueueSize( ){
    return agentQueue.get().size();
}

/*
 * poll a agent from the queue, get null if queue is empty
 * used for callAll, manageAll etc.
 */
public static Agent pollNextAgent( ) {
    return agentQueue.get().poll();
}
```

2. *Agent Index Mapping in AgentList.java*

Since Agents are no longer retrieved directly from the global AgentList, their indexes must be mapped to attributes within the Agent objects to enable correct removal from the AgentList. Moreover, although each thread now operates with a local Agent bag, the global AgentList remains necessary in certain scenarios and therefore its logic is preserved.

```
  public void add(Agent item, int index) {
    int xindex = index / CAPACITY_X;
```

```
    int yindex = index % CAPACITY_X;
    if(array[xindex] == null) {
      array[xindex] = new Agent[CAPACITY_Y];
    }
    item.setAgentListIndex(xindex * capacityY + yindex);
    array[xindex][yindex] = item;
  }
  private void reduceHelper( ) {

      if ( reduceDone )
          return;

      int max = size_unreduced( );
      int currentNull = 0;
      int cur_full = max - 1;
      int xNull, yNull, x_full, y_full;

      while (true ) {

          for ( ; currentNull < max && get( currentNull ) != null; currentNull++ );
          for ( ; cur_full >= 0 && get( cur_full ) == null; cur_full-- );
          if ( currentNull >= cur_full )
              break;

          // swapping
          xNull = currentNull / capacityY;
          yNull = currentNull % capacityY;
          x_full = cur_full / capacityY;
          y_full = cur_full % capacityY;
          array[xNull][yNull] = array[x_full][y_full];
          array[xNull][yNull].setAgentListIndex(xNull * capacityY + yNull);
          array[x_full][y_full] = null;

      }

      xNull = currentNull / capacityY;
      yNull = currentNull % capacityY;

      for ( int i = xNull + 1; i < array.length && array[i] != null; i++ )
          array[i] = null;

      currentX = xNull;
      nextY = yNull;
      reduceDone = true;

  }
```

3. *Agent Retrieval Logic in AgentBase.java*

When invoking an Agent in practice, the retrieval process must operate on the local Agent bag rather than the global bag. This ensures thread-level independence, reduces synchronization overhead, and improves data locality during execution.

```
           Agent tmpAgent = MThread.pollNextAgent();
          if (tmpAgent != null) {
              myIndex = tmpAgent.getAgentListIndex();

              MASS.getLogger().debug( "Thread [" + tid + "]: agent(" + tmpAgent + ")[" +
myIndex + "] was called, agent index is " + tmpAgent.getAgentListIndex() + " place is " +
tmpAgent.getPlace());
```

```
            MASS.getLogger().debug( "fId = " + functionId + " argument " + argument );

            //Use the Agents' callMethod to have it begin running
            MASS.getLogger().debug( "in the other arg: " + argument);
            tmpAgent.callMethod( functionId, argument );

        }

        //Otherwise, we are out of agents and should stop
        //trying to assign any more
        else {
            break;
        }
```

4. *Agent Removal Logic in AgentBase.java*

When an Agent is no longer needed, it must be removed from the Place's AgentList to ensure correctness. This step guarantees that obsolete Agents are not included during the initialization of local Agent bags in subsequent iterations.

```
        } else if (GraphPlaces.class.isAssignableFrom(evaluatedPlaces.getClass())) {
            GraphPlaces graphPlaces = (GraphPlaces) evaluatedPlaces;

            int globalLinearIndex = evaluationAgent.getIndex()[0];
            int nodeId = graphPlaces.getOwnerID(globalLinearIndex);

            //Bo Fu: Agents should be removed from oldPlaces
            //whether it is local migration or remote migration
            Place oldPlace = evaluationAgent.getPlace();

            if (oldPlace.getAgents().remove(evaluationAgent) == false) {
                // should not happen
                String errorMessage = "evaluationAgent {}" +
                        evaluationAgent.getAgentId()
                + " couldn't been found in " +
                "the old place!";

                MASS.getLogger().error(errorMessage);

                // throw it back to our new fatal exception handler
                throw new RuntimeException(errorMessage);
            }
```

Other related implementations include the method for setting indexes in Agent.java, as well as the logical adjustments in different versions of callAll and manageAll within AgentBase.java. Since the underlying principles are similar, they are not elaborated here, and the source code can be found in the Appendix.

IV. Results

A. Multithreading Improvements on Single Node

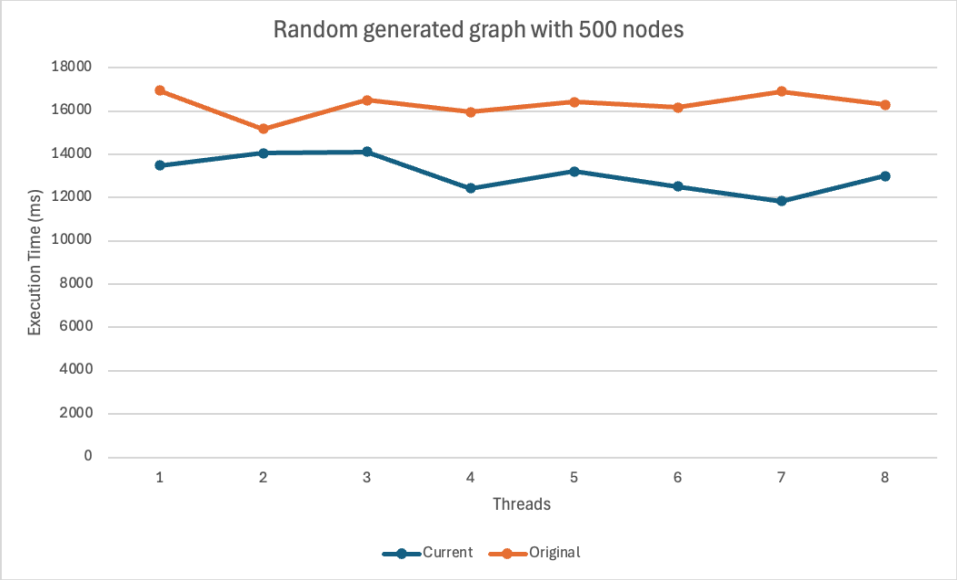

Figure 1. Multithreading Execution Performance on 1 Node With a Randomly Generated 500-node Graph

In Figure 1, "Original" refers to the MASS Java library where only the multithreading logic in GraphPlaces is implemented, while "Current" refers to the version with the new Agent allocation algorithm. The results show that adding multithreading to GraphPlaces alone improves performance by about 10%. After modifying the Agent logic, the system achieves around 25% faster execution in single-thread mode and about 43% faster execution in multithread mode compared to the original single-thread baseline.

It is worth noting that even without multithreading, the new Agent allocation strategy brings significant improvements. This is likely because Agents in the same Place are executed sequentially rather than interleaved across different Places, which makes better use of CPU and cache. In addition, although the cssmpi machines have only four cores, further speedup is still observed beyond four threads. This may be explained by finer-grained load balancing from splitting Agents into smaller portions, as well as operating system scheduling.

B. Multithreading Performance on Graphs with Different Structures

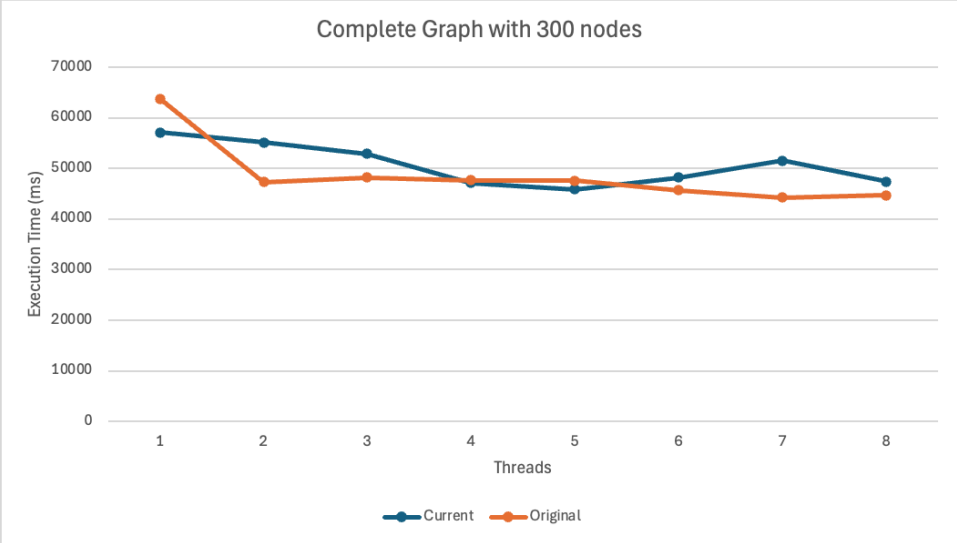

Figure 2. Multithreading Execution Performance on 1 Node With a 300-node Complete Graph

However, the new Agent allocation algorithm is not always more efficient. When running TriangleCounting on a complete graph, the original allocation strategy performs faster. This is because, in TriangleCounting on a complete graph, Agents tend to cluster in Places with smaller indexes. Allocating Agents based on Places in this case leads to workload imbalance across threads, thereby diminishing the performance gains from multithreading.

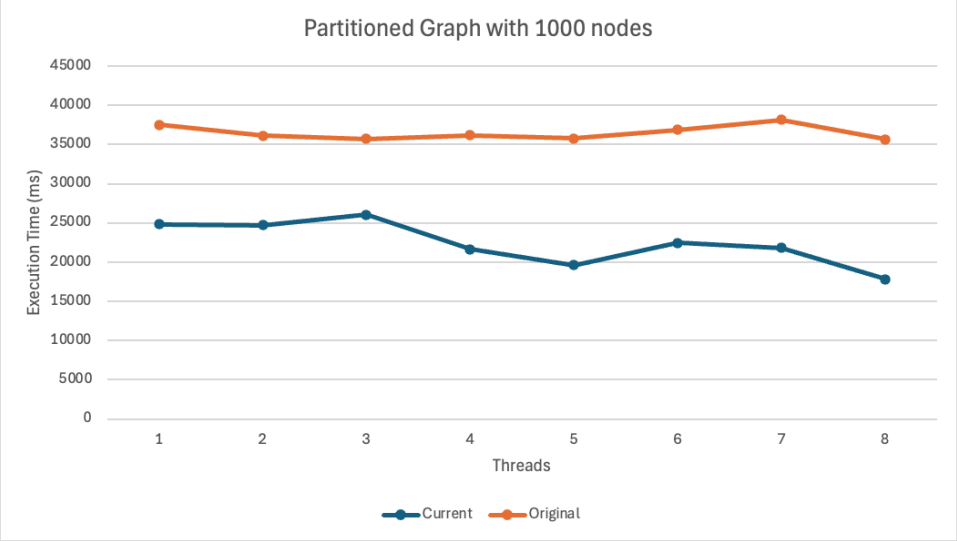

Figure 3. Multithreading Execution Performance on 1 Node With a Partitioned 1000-node Graph

Figure 3 also shows that the new Agent allocation algorithm is highly influenced by the graph structure. Here, a partitioned complete graph is used to ensure balanced workloads across threads, and under this condition, the new scheduling algorithm achieves substantial improvements in multithreaded performance.

C. *Multithreading Improvements on Multiple Nodes*

Table 1. Best Single-Thread and Multi-Thread Performance under Multi-Node With a 3000-node Graph

Nodes	Current 1 thread	Current multithread	Original 1 thread	Original multithread
1	46039	42011	95601	89973
2	35896	32946	57517	56701
4	21218	21087	30176	28817
8	14866	**15285**	19208	18955

Table 2. Best Single-Thread and Multi-Thread Performance under Multi-Node With a 5000-node Graph

Nodes	Current 1 thread	Current multithread	Original 1 thread	Original multithread
4	33641	31588	53306	51840
8	22503	22042	29339	28960

As shown in Tables 1 and 2, the benefit of multithreading becomes smaller and can even cause performance to drop as the number of nodes increases. This is probably because each node has fewer Agents to handle, while most of the overhead comes from communication or other costs, so multithreading brings little improvement.

D. *Reduced memory usage*

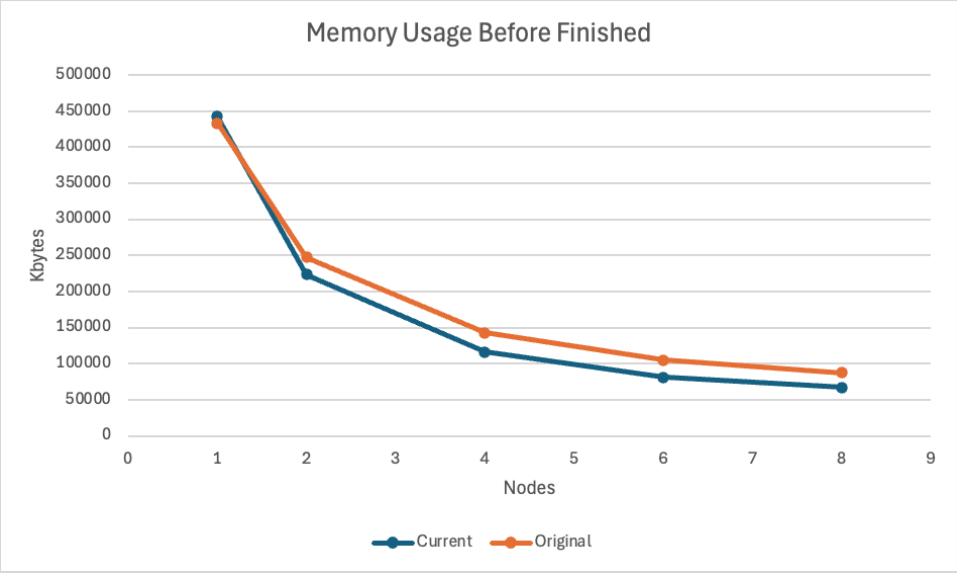

Figure 4. Memory Usage Before MASS.Finish()

Figure 4 shows the memory usage at the end of the MASS program execution, just before termination. The new AgentBase implementation reduces memory usage by 23% compared to the original design. This improvement is due to the removal of references to Agents within the Place objects that were not properly cleared in the previous implementation.

V. Conclusions and Discussions

A. Conclusions

With the implementation of multithreading logic in GraphPlaces, MASS Java can execute with multiple threads reliably, achieving approximately a 10% improvement in performance.

This independent study also introduced a new Agent allocation strategy, which demonstrated further performance gains in both single-threaded and multithreaded executions. In the TriangleCounting application, the revised approach improved single-thread performance by about 25% and multithread performance by about 43%.

Further exploration on different graph structures shows that the efficiency of the new allocation algorithm is strongly influenced by the graph structure. It performs well on partitioned graphs with balanced workloads but shows reduced effectiveness on complete graphs. Moreover, in multi-node environments, the improvement was much less than in single-node execution, and when the number of nodes reached 8, multithreading even led to performance degradation.

Finally, the revised allocation method helped reduce potential memory leaks, suggesting better scalability in terms of memory usage.

B. Limitations and Future Work

This study still has several limitations. As mentioned earlier, the performance of the new Agent allocation algorithm is affected by the structure of the graph, and the multithreading performance shows weaker results in multi-node environments and on small-scale graphs. In addition, the current multithreaded allocation algorithm for GraphPlaces remains relatively simple.

Future improvements could involve integrating Atul's previous METIS-based graph partitioning method to achieve more effective multithreaded allocation of Agents and GraphPlaces, with the goal of balancing workloads across threads. Another possible direction is to further reduce the synchronization overhead in multithreaded execution.

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:

Branch:

mass_library_developers/mass_java_core/fuchacha/agent-multithread

Pull Request:

#57 BoFu-Agent-Multithreading

https://bitbucket.org/mass_library_developers/mass_java_core/pull-requests/57