# An Interactive Environment to Support Agent-based Graph Programming (Draft Paper)

Daniel Blashaw[a] and Munehiro Fukuda[b]
*Division of Computing and Software Systems, University of Washington Bothell, U.S.A.*
{*dblashaw, mfukuda*}*@uw.edu*

Keywords: ABM, visualization, big graphs, parallel computing

Abstract: We apply agent-based modeling (ABM) to distributed graph analysis where a large number of reactive agents roam over a distributed graph to find its structural attributes, (e.g., significant subgraphs including triangles in a social network and network motifs in a biological network). Of importance is providing data scientists with an interactive environment to support agent-based graph programming, which enables interactive verification of agent behaviors, trial-and-error operations, and visualization of graphs and agent activities. This paper presents and evaluates our implementation techniques of these interactive features.

## 1 INTRODUCTION

In contrast to conventional data streaming, we apply agent-based modeling (ABM) to big-data computing (Fukuda et al., 2020). More specifically, instead of streaming data to analyzers such as Spark[1] and Flink[2], we construct a distributed data structure, dispatch reactive agents to it as mobile analyzers, and find its structural attribute in their emergent collective group behavior. For instance, triangles in a given social network is considered as a useful factor to measure the intimacy among the network users and can be counted by walking agents three times over the network (Gordon et al., 2019).

Distributed data structures have been facilitated for years in well-known parallel and distributed systems. GlobalArray constructs multi-dimensional arrays on top of MPI (Nieplocha et al., 2006). Pregel is a large-scale graph library based on inter-vertex message passing (Malewicz et al., 2010) and is used in Spark's data streaming as GraphX. RepastHPC is the parallel version of Repast Simphony[3] that distinguishes spatial and network projections.

Focusing on graph analysis, these systems however have substantial difficulties in implementing our agent-based approach: GlobalArray could represent a graph with an adjacency matrix but does not support element-to-element communication, thus obstructing agent communication nor movement; Pregel and GraphX nail computation in their vertices, immobilizing agents over a graph; and RepastHPC is meant for traditional ABM simulation, not considering parallel file I/Os nor interactive operations on its network projections. Given this background, we facilitated distributed graph construction, agents' graph traversal, and Cytoscape[4]-enabled graph visualization in the MASS (multi-agent spatial simulation) library (Gilroy et al., 2020).

Of importance is providing data scientists with an interactive environment to support agent-based graph programming, which includes interactive verification of agent behaviors, capability of trial-and-error operations, and visualization of graphs and agent activities. This paper presents and evaluates our implementation techniques of these interactive features.

The rest of this paper is organized as follows: Section 2 differentiates our interactive environment from the related work in graph programming; Section 3 gives technical details on the MASS interactive features and their implementation; Section 4 evaluates MASS execution overheads, programmability improvements, and visualization; and Section 5 concludes our discussions as mentioning our future plans.

[a] https://orcid.org/0000-0002-0822-6667
[b] https://orcid.org/0000-0001-7285-2569
[1] http://spark.apache.org/
[2] http://flink.apache.org/
[3] https://repast.github.io/
[4] http://cytoscape.org/

## 2 RELATED WORK

This section compares the MASS library with other related systems from the following four viewpoints: (1) potential of distributed graph analysis with agents, (2) agent tracking over a distributed graph, (3) forward and backward graph analysis, and (4) visualization of graphs and agents.

### 2.1 DISTRIBUTED GRAPH ANALYSIS WITH AGENTS

NetLogo[5] and Repast Simphony are capable of simulating networked agents or agent movements over a network, respectively using network extensions or network projections. However, the biggest challenge results from their single-computing execution that limits graph scalability. In fact, our scalability test shows that Repast Simphony suffers from counting the number of triangles in a graph only with 3,000 vertices (Wenger et al., 2021).

RepastHPC and FLAME[6] are MPI-supported parallel ABM simulators. The former maps Repast Simphony's network projection onto a cluster system where agents can traverse the network, thus moving from one cluster node to another. The latter populates agents statically over a cluster system where networked agents can communicate with their neighbors through broadcast messages. Although these two systems parallelize ABM in graphs, their MPI-based C/C++ implementation does not consider interactive operations that are essential to the speed or the serving layer in big-data computing. Furthermore, they cannot initialize a graph in parallel as rank 0 must read an input file sequentially.

WAVE (Sapthy and Borst, 1996) and UCI Messengers (Bic et al., 1996) are mobile-agent execution platforms, both allowing their agents to construct and to roam over a distributed graph at run-time. Their drawback is the necessity of describing graph construction logics in their agent code, which in turn means that they are incapable of automating graph construction from an input file and are thus unsuited to big-data computing.

### 2.2 AGENT TRACKING

ProvMASS (Davis et al., 2018) provides MASS users with a novel approach for tracking data provenance in a distributed setting. This data provenance includes agent data, simulation space, and cluster node information, and is captured to file at run-time. Although these data provenance features contribute to analyzing agent behavior, they have a significant impact on simulation performance. Specifically, such agent-tracking implementation must be lightweight enough to keep in memory during simulation and to practicalize interactive graph analysis.

Repast Simphony, on the other hand, has a lightweight implementation for tracking agent data, but settings must be pre-configured before running the simulation and recorded agent data can only be written to console or file. This is useful for reviewing agent information but does not facilitate interactive uses nor operations on the agent data in a running simulation.

IBM Aglets (Lange and Oshima, 1998) allows users to communicate with the agent servers named Tahiti. Through Tahiti's GUI, users can create, clone, inspect, dialogue with, retract, and destroy agents. However, since Aglets are intended to work on Internet tasks, they do not distinguish distributed data structures nor duplicate too many instances through the GUI menus.

### 2.3 FORWARD AND BACKWARD GRAPH ANALYSIS

We anticipate that data scientists may want to conduct various analyses on the same graph, (e.g., centrality and clustering analyses on the same biological biological network). These operations need to retract agents or even roll back computation, which are then followed by a new analysis. Some systems indirectly or directly implement such forward and backward computation as follows:

Optimistic synchronization in parallel simulators (Wang and Zhang, 2017) allows each computing node to take repetitive snapshots of on-going computation for the purpose of rolling back to the computation and accepting tardy messages from slower computing nodes. As their checkpointing and rollback operations are system-initiated features, users cannot use them intentionally for their trial-and-error analysis.

The UCI Messengers system implements the optimistic synchronization in the execution platforms so that agents can automatically go back to a network node they previously visited (Fukuda et al., 1998). Needless to say, agent checkpointing and rollback are carried out automatically and thus not user-controllable.

The MASS library freezes agents when they explosively clone themselves in a short time period,

[5]https://ccl.northwestern.edu/netlogo/

[6]http://www.flame.ac.uk/

which prevents physical memory from being exhausted quickly (Mart, 2017). These agents are serialized and stored in either separate memory or disk space until the other agents complete their computation and release their memory space. Again, this is a system-automated but not user-controllable feature.

Looking at single-CPU execution, Repast Simphony requires that all graph and agent information be set prior to execution of the simulation and does not support incremental backtracking or manipulation of a running simulation.

## 2.4 GRAPH VISUALIZATION

Single-CPU ABM simulators furnish non-computing users with a plenty of graph analyzing and visualization features. NetLogo arranges an IDE-based graph visualization with its network extension (Wilensky, 2013). It includes a plenty of primitives for network analysis, (e.g. centrality and clustering analyses) and visualizes the resultant graphs. However, it forces users to pre-configure visualizations and lacks the mid-simulation control features. Repast Simphony is equipped with JUNG (O'Madadhain et al., 2003) as its internal graph tool that uses the Java Swing API to display graphs. This in turn means that data scientists need to embed visualization logics in their graph programming.

Cytoscape is an open-source network visualization tool, originally developed for use in analysis of biomolecular interaction networks, which has grown to be widely used by various disciplines for the following three graph-programming supports: (1) extensive file support for importing graphs into Cytoscape, (2) native functionality for dynamic manipulation of existing graph structures, and (3) the use of the OSGi framework to make its components modular and easily extensible. Needless to say, Cytoscape is not concerned with agent activities on a Cytoscape graph.

We should emphasize that all these graph visualization endeavors are limited to single-CPU execution, thus unable to address the demand for large-scale graph analysis.

In summary of this section, agent-based graph programming needs to address interactive and scalability problems in the following three areas:

1. **Agent tracking:** quickly observing a large number of agents traversing a graph;
2. **Forward and backward computation:** interactively retracting active agents, restoring former graph states, and dispatching new agents; and
3. **Graph visualization:** dynamically modifying graphs through GUI and visualizing agent activities on a graph.

# 3 INTERACTIVE FEATURES AND THEIR IMPLEMENTATION

In the following, we briefly introduce the MASS library and thereafter explain our technical solutions to three interactive graph-programming features: (1) agent tracking, (2) forward and backward computation, and (3) graph visualization.

## 3.1 MASS LIBRARY

The MASS library represents ABM using the two modeling objects: *Places* and *Agents*. *Places* is a computational space implemented with a multi-dimensional array that is partitions into smaller stripes, each mapped to a different cluster node. Each place, an independent array element is referred to with an architecture agnostic index and capable of exchanging data amongst themselves. On the other hand, *Agents* is a collection of reactive agents within the computation. An agent has navigational autonomy of traversing places. Upon a migration, agent data is serialized and passed between cluster nodes via TCP communication.

The MASS library functions using a master-worker pattern to control the computation. User applications interact with the MASS master node that runs their main() function; starts MASS workers with *MASS.init()*; invokes a parallel function call at each place or agent with *Places.callAll(func)* or *Agents.callAll(func)*; exchange data among places in an inter-place RPC form with *Places.exchangeAll(func)*; clones, kills, and moves agents within *Agents.manageAll()*; and terminates the MASS workers with *MASS.finish()*.

To ease graph programming, MASS derives *GraphPlaces* from the *Places* base (Gilroy et al., 2020). Using this class, users can initialize a distributed graph with an input file in XML, HIPPIE, CSV, and text formats. As *GraphPlaces* can grow by adding a new *Places* instance to itself, users can incrementally construct the graph with *addVertex()* and *addEdge()*.

The MASS library interfaces with Cytoscape for GUI-enabled graph construction and visualization as well as with JShell (Oracle, 2017) for interactive agent deployment over the graph.

## 3.2 AGENT TRACKING

We implemented an agent-tracking feature in MASS, based on the following three design strategies:

1. **performance:** minimizing temporal and spatial overheads incurred by additional network com-

munication and memory management, thus enabling quick observation of many agent activities;
2. **consistency:** generating unambiguous and exactly-once outputs of agent propagation history including cloning and migration over a graph; and
3. **usability:** facilitating a straightforward, easy-to-use agent-tracking API and providing users with easily consumable statistics such as finding all agents alive at a particular time or determining the number of visits an agent made.

To pursue the performance consideration, instead of allowing each agent to carry its travel history with it, we added to each place the AgentHistoryManager class that is responsible for managing which agents or classes of agents are being tracked and then recording history each time a tracked agent visits that place. Importantly, this means agent history is stored on the *Places* but not on the *Agents*; this maintains execution performance by ensuring agents remain lightweight for serialization and transfer between computing nodes. The tradeoff is that agent history is distributed amongst the cluster nodes during execution which introduces some complications when extracting the data, especially when child agents are involved.

In many applications, agents will come to decision points at which their instructions indicate they need to travel to multiple places at once. In these instances, the parent agent will move to one place, and then a child agent will be spawned for each of the other available places. At this point, if agents are being tracked by their class name, then places will begin gathering data on the newly spawned child agents. This pattern may continue throughout the computation, causing multiple waves of child agents to spawn at various times in the computation. The issue that arises from this process is that the child agents will have an incomplete history, because they did not exist at the beginning of execution.

To help illustrate this problem, consider the Triangle Counting benchmark application running on a sample graph shown in Figure 1. Triangle Counting is solved in four ABM simulation cycles:

**Time 0:** one agent is spawned on each available place.

**Time 1-2:** each agent propagates itself to all neighboring places with a lessor index value than the current place (which results in spawning children if more than one neighbors fits this criterion).

**Time 3:** all remaining agents attempt to return to their original source. Each agent that can return home at time 3 represents a discovered triangle.

Figure 2 shows the history captured for each agent

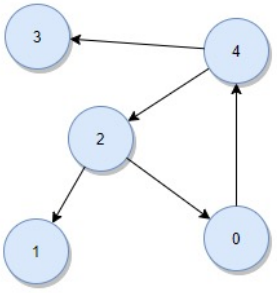


Figure 1: A sample graph

| Agent (Parent) | Time 0 | Time 1 | Time 2 | Time 3 |
|---|---|---|---|---|
| 0 | [0,0] | | | |
| 1 | [0,1] | | | |
| 2 | [0,2] | [1,1] | | |
| 3 | [0,3] | | | |
| 4 | [0,4] | [1,3] | | |
| 5 (4) | ? | [1,2] | [2,1] | |
| 6 (2) | ? | [1,0] | | |
| 7 (5) | ? | ? | [2,0] | [3,4] |

Figure 2: Agent history before parent propagation, where $[t,v]$ means: at time $t$, a given agent visited vertex $v$

as this benchmark application plays out; note that child agents are missing their parent itinerary before they were spawned. This missing piece of data is needed for the user application to correctly determine the path the agents traveled along, i.e., the edges of the triangles.

As shown in Figure 2, agent 7 is the only agent that completed its path at time 3, but its complete path is unknown because agent 7 is a child and did not spawn until time 2. Further, agent 7 is the child of agent 5 and agent 5 is also a child agent of agent 4. So, even if we retrieve information from agent 5, the data will remain incomplete unless we also pull information from agent 4. Thus, there is the need to solve this problem depth-first recursively at the time agent 7's history is extracted to the user program.

Figure 3 illustrates the result of propagating data from parent agents, with the red arrows indicating the flow of information from parent to child. In the case of agent 7, we see that it retrieved results directly from parent agent 5, after agent 5 retrieved its own history from parent agent 4. Now, from agent 7s movement history, we can correctly conclude the triangle found from this simulation is between vertices 4, 2, and 0.

| Agent (Parent) | Time 0 | Time 1 | Time 2 | Time 3 |
|---|---|---|---|---|
| 0 | [0,0] | | | |
| 1 | [0,1] | | | |
| 2 | [0,2] | [1,1] | | |
| 3 | [0,3] | | | |
| 4 | [0,4] | [1,3] | | |
| 5 (4) | [0,4] | [1,2] | [2,1] | |
| 6 (2) | [0,2] | [1,0] | | |
| 7 (5) | [0,4] | [1,2] | [2,0] | [3,4] |

Figure 3: Agent history after parent propagation

Listing 1 shows a code snippet to initiate agent tracking and to collect agent travel history. The *main()* program creates a graph (line 5) and initializes *AgentHistoryManager* at each graph place (line 6). Once an agent class is registered for tracking, the user program may continue execution without worrying about tracking data. Each time the manageAll() function is invoked, the MASS library will keep track of all associated agent movements. Although the code invokes *manageAll()* twice for each loop iteration, one in line 10 for spawning children and the other in line 12 for moving all the agents to neighboring places, *manageAll()* internally counts these two invocations as one logical time event. This gives users a simple view of agent dissemination. Finally, *main()* can retrieve all the agent travel histories through *graph.callAll(AGENT_TRACE_GET)* (line 14). This retrieval process internally consolidates all collected results into a single, cleaned, and sorted *AgentHistoryCollection* object. It is at this step that parent-data propagation occurs.

Listing 1: Using MASS agent tracking

```
1 import MASS.*;
2 public class Analysis {
3   public void main(String[] args) {
4     MASS.init();
5     GraphPlaces graph = new GraphPlaces( ... );
6     graph.callAll(places.
            AGENT_TRACE_REGISTER_CLASS, Crawler
            .class.getName());
7     Agents crawlers = new Agents(``Crawler'', graph);
8     while ( crawlers.hasAgents() ) {
9       crawlers.callAll(ClawerAgent.spawn_);
10      crawlers.manageAll();
11      crawlers.callAll(ClawerAgent.walk_);
12      crawlers.manageAll();
13    }
14    AgentHistoryCollection history = graph.callAll(
            places.AGENT_TRACE_GET);
15    MASS.finish();
16 } }
```

## 3.3 FORWARD AND BACKWARD COMPUTATION

To provide incremental execution and the ability to checkpoint and rollback computation, we have leveraged the MASS library's interface to JShell named InMASS (Alghamdi, 2020). Our design strategies are two-fold:

1. **performance:** minimizing temporal and spatial overheads incurred by checkpointing an on-going computation and retrieving a past computation
2. **code base:** maintaining the original InMASS implementation with less impact on the basic functionality.

In conventional data streaming, Spark addresses forward/backward computation with immutable RDDs that create new versions upon any transformation applied to them, and thus keeps old RDDs retrievable with their references. This strategy takes the same effect as checkpointing and rollback of computation. To avoid generating too many snapshots of RDD, Spark carries out lazy evaluation of RDD transformations until they really need to be evaluated for passing their changes to RDD actions (which produce non-RDD values). In agent-based graph programming, agents travel or propagate over a graph as changing each data item. This in turn means that, if we use the same strategy as Sparks dataset immutability, we need to take a snapshot every time an agent changes each vertex. Furthermore, unlike Sparks RDD, (i.e., a collection of data items), a graph needs more disk space for storing its serialized data upon a checkpointing and more time for de-serializing it upon a rollback. Taking these overheads in consideration, we implemented interactive parallelization in the MASS library as follows:

1. **Maintaining only one snapshot of computation:** MASS users are supposed to commit their operations to an in-memory graph once they have no intention to roll back beyond this checkpoint. This saves the secondary storage space.
2. **Maintaining a history of previous MASS function calls:** The MASS library will keep recording any MASS functions invoked since the last snapshot was taken, so that MASS can rebuild any past graph structure between the snapshot and the latest graph state.
3. **Rolling back computation by re-executing functions in history:** Upon a user-specified rollback, the MASS library will re-apply previous function calls to the snapshot in a chronological order all the way to the rollback point. While this rollback scheme needs a substantial time to rebuild a past graph, the normal computation can run faster without continuously taking a snapshot of ongoing executions onto disk.

At the highest level, InMASS is simply a wrapper class that initializes a JShell window, injects MASS startup code into that JShell instance, and then provides hooks for various MASS execution and shutdown functions. This basic functionality alone enables line-by-line execution when running on a single node and effectively eliminates boilerplate code in user applications. The challenges of InMASS implementation, however, revolve around (1) making JShell

function properly for all cluster nodes in a distributed environment and (2) deciding how to save and reload computation state for checkpoint and rollback functionality.

To address issue 1, we customized a Java class loader named *InMASSLoader* that facilitates distribution of new classes bytecode from the MASS master to worker nodes. Once all computing nodes are aware of the new classes, they use new *MASSObjectInputStream* and *MASSObjectOutputStream* functions to assist in serialization and deserialization of these dynamic classes.

To address issue 2, we had *Agents* and *Places* inherit *AgentsInternal* and *PlacesInternal* serializable classes to facilitate serialization and deserialization of all agents and places data. Then, each MASS worker process gathers all hash tables containing all *Agents* and *Places* instances, and holds them in one single object named *MState*. This is the object to be saved and updated on checkpoint and rollback. (Note that users can choose a checkpoint storage from active memory, temporary disk location, or a specified file in disk.) To facilitate user ability to rollback to states other than the original checkpoint, the MASS master process prepares the *MHistory* object to keep a log of all API calls to *Agents* and *Places* and to store their bytecode to enable re-execution on demand. Consequently, when a user requests rollback to "step 5", for example, the original snapshot will be loaded from *MState*, and then *MHistory* will execute the next five API calls that follow the snapshot.

## 3.4 GRAPH AND AGENT VISUALIZATION

To facilitate visualization and validation of agent activities over a distributed graph, we have extended the existing MASS-Cytoscape integration, based on the following three implementation strategies:

1. **Usability:** allowing users to focus on programming their graph application in MASS, while not configuring or managing the visualization solution. This automation includes transfer of agent data to Cytoscape.
2. **Expandability:** following the OSGI framework to modularize MASS-related plugins, which eases the future expandability of the MASS-Cytoscape system integration.
3. **Scalability:** allowing large-scale graphs to be shown in Cytoscape by extending its capability to retrieve partial graphs from the MASS library, using an n-neighbors approach.

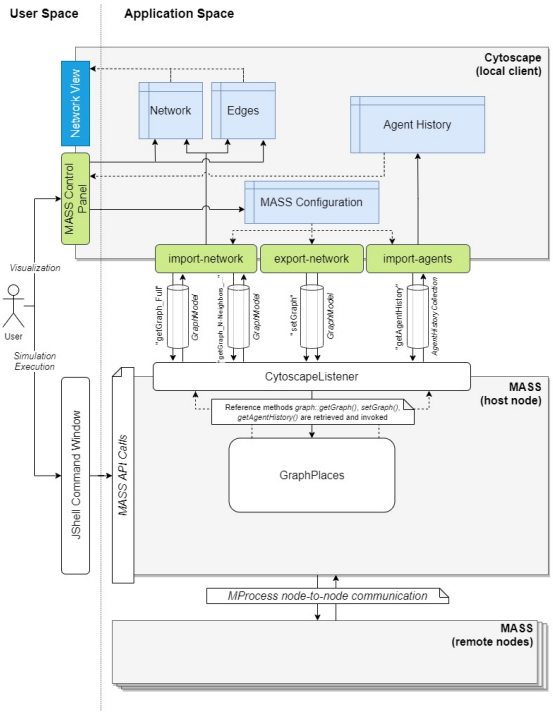


Figure 4: MASS-Cytoscape integration architecture diagram

### 3.4.1 USABILITY ENHANCEMENT

Figure 4 presents an overview of the MASS-Cytoscape architecture. On the left side of the figure is the user space. We have illustrated the users two points of interaction, with the JShell window for running their code in MASS and with the MASS Control Panel for managing their data flow and visualization in Cytoscape.

The MASS Control Panel serves three main functions. First, it provides a single point of interaction for the user by internally managing the data transfer plugins: import-network, export-network, and import-agents. Second, it provides the ability to manipulate the MASS Configuration tables that inform the data transfer plugins of how to find the MASS computation and what data to pull back into Cytoscape. Lastly, it provides the interface and logic for visualizing agent movement through manipulation of the Cytoscape data tables and network view.

In MASS, the CytoscapeListener class must be started by the user application to open a TCP-based communication port for MASS-Cytoscape communication. This listener will then field any requests from Cytoscape by first parsing the request, then obtaining reference to the corresponding *GraphPlaces* method, and finally invoking that method and returning the results to the requesting Cytoscape plugin. Internally,

the *GraphPlaces* methods utilize standard MASS internal APIs, such as *callAll()*, to communicate with the rest of the cluster and set or retrieve the appropriate information.

Visualizations in Cytoscape are all controlled by two factors: the layout and the network view. While we utilized the Circular layout, (i.e., one of Cytoscape's defaults), we customized the network view to change edge thickness and vertex color for visualizing agent travel histories.

### 3.4.2 MODULARITY AND EXPANDABILITY

Cytoscape plugin components must follow OSGi to modularize them in a bundle. Since each bundle is self-contained, plugin developers are responsible to coordinate bundle invocations and to provide bundle-to-bundle data communication. We handle this communication using a shared table named "MASS_Configuration table within the Cytoscape environment. The table includes fields for the MASS hostname and port as well as other fields used for partial graph streaming. The MASS Control Panel writes new values to the table after taking inputs from the UI and the data transfer plugins read in relevant information each time a new data transfer task is created. To tolerate any ordered start-up of bundle, the panel also maintains a reference to each of the data transfer plugins and the ability to test and reacquire the reference.

### 3.4.3 GRAPH SCALABILITY

Visualization of a partial graph is important in situations where the MASS cluster is operating on a graph too large to be stored in a single machine. To allow the visualization environment to support the scale of these graphs, we have implemented an optional N-Neighbors approach to graph retrieval from MASS. This requires the user to provide a centroid node ID as well as determine the degrees of separation (DoS) that should be imported. DoS is interpreted as the number of neighbor rings that we would like to visualize. Shown in Figure 5, if the user selects 1 DoS then the graph retrieval will bring back the centroid node as well as one ring of immediate neighbors. Specifying 2 DoS would bring back all the centroids neighbors as well as the neighbors of those nodes in the first ring, and so on.

To manage this request on the MASS side of the program, the MASS master node first receives the request, saves centroid and DoS information, and then invokes *GraphPlaces*'s *getGraphNNeighbors()* method. This method iteratively queries the remote workers for each DoS requested, passing a list each time to ensure only the required graph vertices are sent back. Importantly, this approach results in additional overheads from iterative network calls but ensures that the data returned to the master node is limited, which is critical when a given graph is too large for a single machine.

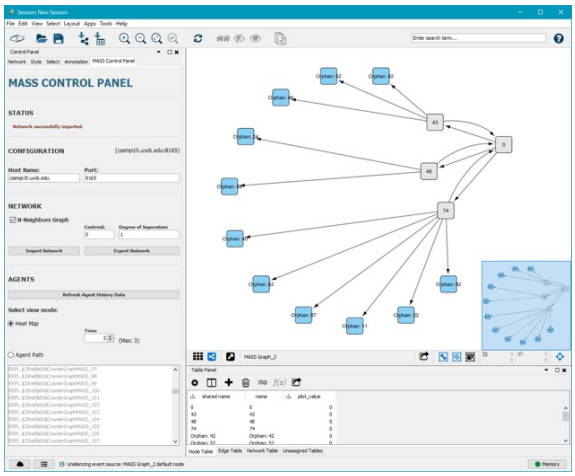

Figure 5: N-Neighbors Graph Retrieval (where Centroid = 0, DoS = 1)

## 4 EVALUATION

We evaluated the MASS interactive environment in the following three criteria: (1) execution performance, (2) ease of programming, and (3) usefulness of visualization. At the end of this section, we also discuss about the current limitations. All the verifications, measurements, and visualizations were performed using a cluster of eight computing nodes, each with an Intel Xeon Gold 6130 CPU at 2.10GHz and 20 gigabytes of system memory.

## 4.1 EXECUTION PERFORMANCE

We first compared the execution performance between the agent-tracking API and the conventional history-passing technique (i.e., maintaining and passing a travel history from a parent to its children directly every time new children was spawned). We utilized the agent-based Triangle Counting benchmark application for this comparison. Table 1 summarizes five different graph sizes between 100 and 1,500 vertices.

As the size of the graph increases, we also observe a corresponding exponential increase in the total number of agents needed to perform the analysis. Due to variability in complexity from one graph to the next, and the management of agent data being the primary

Table 1: Graph sizes used for performance evaluation

| #vertices | max # neighbors | total #agents spawned | #triangles found |
|---|---|---|---|
| 100 | 8 | 373 | 8 |
| 500 | 19 | 7,734 | 145 |
| 750 | 28 | 24,818 | 526 |
| 1,000 | 50 | 143,486 | 4,578 |
| 1,500 | 66 | 342,802 | 9,124 |

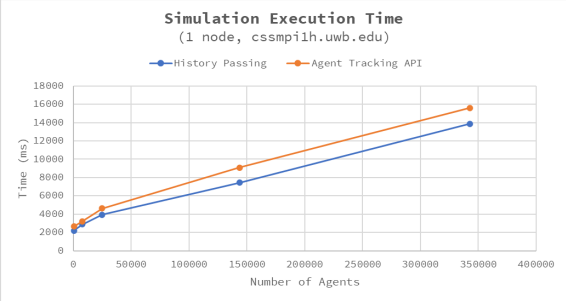


Figure 6: Performance of triangle counting with agent-tracking API versus history-passing technique

point of interest for these features, we will use the total number of agents to provide context to our results.

Figure 6 shows Triangle Counting's execution time as increasing the number of agents. Note that we calculated the average of five measurements. Beyond 140K agents, we observe approximately 1,700ms ( 11-18%) slower performance when using the agent-tracking API. This increase in processing time is due to the added overhead from data capture methods invoked when agents are spawning and moving. This gap widens slightly when running in multi-node configurations due to network latency, but the correlation between the two techniques remains consistent.

Figure 7 compares agent-tracking API and the history-passing technique in their overheads when extracting agent data. We conversely observe a significant gap in performance between the two techniques regarding agent-data extraction time. There are two primary explanations for this disparity in performance:

1. The amount of data being returned to the user is significantly different. Using the agent-tracking API returns all agent data captured in computation while the history-passing technique returns only data for agents that are alive at the time of retrieval. This is the same as the number of triangles found at the end of computation. In terms of the 1,500 vertices trial, this means that the history-passing technique returned data for 9,124 agents while the agent-tracking API approach returned results for 342,802 agents.
2. The agent-tracking API takes additional steps to clean, sort, and propagate the data upon retrieval. The impact of this extra data processing is somewhat limited in Triangle Counting, however, because the three repetitive walks on triangle edges limit parent-child propagation to at most two levels. Longer computation with more parent-child relationships is likely to see further increase in agent-data extraction time.

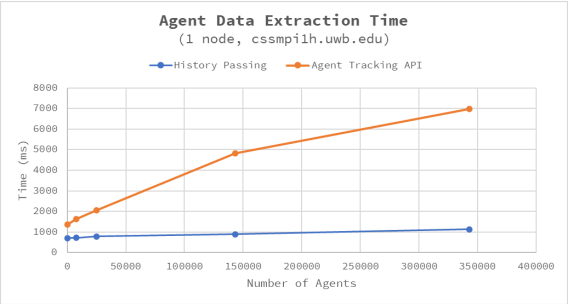


Figure 7: Agent data extraction overheads incurred by agent-tracking API and history-passing technique

## 4.2 PROGRAMMABILITY EVALUATION

Table 2 shows a qualitative comparison of the two techniques. First, the history-passing technique only works for agents that are alive at the time of data extraction; whereas the agent-tracking API manages history for all registered agents and makes that data available at any point in execution. Second, the agent-tracking API is arguably more intuitive for inexperienced MASS users. The history-passing technique requires the user to have additional understanding of the MASS library: to understand that they can pass arguments from the parent to the child agents to maintain a log of visits. In contrast, the agent-tracking API can be initiated and then ignored until needed, allowing the user to instead focus on the logic of their application. Additionally, the agent-tracking API will function the same way in all applications allowing knowledge of its use to be easily transferred to new projects. Finally, the agent-tracking API shows a small reduction in # lines of code (LoC) and, more importantly, a consolidation of those lines to one initialization statement and then a single block of code to retrieve and process the data.

## 4.3 GRAPH AND AGENT VISUALIZATION

To facilitate user exploration and computation understanding, we have implemented two agent visualiza-

Table 2: Programmability comparison between agent history passing and agent-tracking API

| | Agent-history passing | Agent-tracking API |
|---|---|---|
| Completeness | Only alive agents | All tracked agents |
| Ease of use | Less intuitive | More intuitive |
| LoC | 15 | 13 |

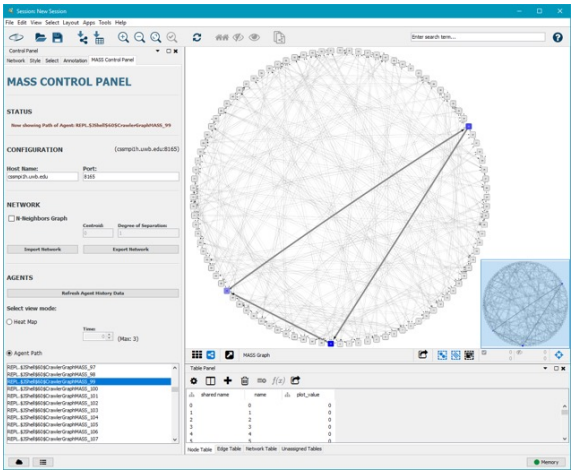


Figure 8: Agent path visualization

tions in Cytoscape: Agent Path and Heat Map. Both visualizations are generated using the MASS Control Panel and each can be freely manipulated in the network view.

Agent Path is shown in Figure 8 and provides the user with the ability to review the complete path of any individual agent. The more recent movements are represented with a darker node and thicker edge. If an agent ever traverses an edge that does not exist in the edge table, then a dashed line is created to signal the issue to the user. This view is particularly useful in computation, such as Triangle Counting, where the pattern of agent movement determines the success of the application. In Figure 8, we see the selected agent was successful in finding a triangle because the agent was able to return to its origin.

The Heat Map visualization, shown in Figure 9, provides the user a representation of all agents active in the simulation at a point in time with darker nodes representing higher concentrations of agents. The user is then able to cycle through the time variable of the computation to observe movement patterns of the entire group. This visualization is best applied to use cases such as in network centrality analyses, in which we seek to observe aggregate movement patterns centralized around particular vertices of interest in graph analysis.

Table 3 provides a comparison of key features between previous versions of MASS Java, this implementation, and competitor software Repast Simphony with regards to agent-based graph programming. Compared features are then sub-divided into execution and visualization categories.

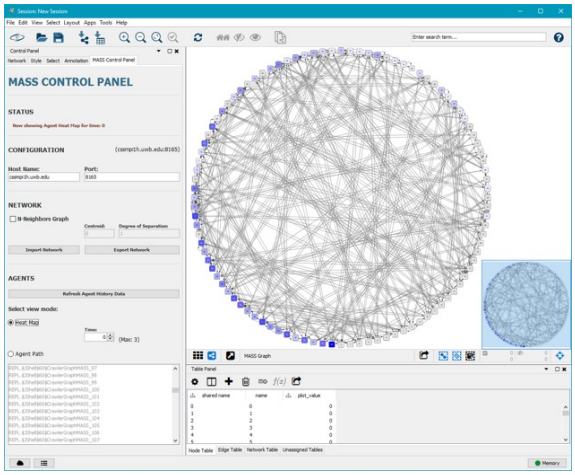


Figure 9: Heat map visualization

Table 3: Visualization and interactivity comparison between MASS and Repast Simphony

| | | MASS (Graphs) | MASS (Graphs) w/ New Features | Repast Simphony |
|---|---|---|---|---|
| *Execution* | Interface | CLI | CLI | GUI |
| | Flow Control | Forward Complete | Forward / Backward Stepping | Forward Stepping |
| *Visualization* | Interface | GUI | GUI | GUI |
| | Objects | Graph Only | Graph + Agents | Graph + Agents |
| | Type | 2D | 2D | 2D / 3D |
| | Synchronicity | Retrieved at end | Asynchronous | Synchronous |
| | Flexibility | Set at start | As needed, selectable | Set at start |

Repast Simphony provides a GUI for both execution and visualization of graph analysis through their integration with the Eclipse IDE. This integration also provides plugin support for 2D/3D visualization of the simulation Context, (which corresponds to MASS *Places* and *Agents*.) The visualizations, known as Projections, (which includes Network Projections) are configured through the IDE before starting the simulation and are strictly synchronized with simulation execution. Repast Simphony does provide statistics and logging features for reviewing historical data, but the simulation itself is limited to only stepping forward through execution or running the simulation at full speed.

MASS with the new interactive support separates the execution and visualization aspects of the system into two windows. Computation is handled through the command-line interface, which also enables forward and backward stepping through the in-

clusion of JShell with checkpointing and rollback features. Visualization is then managed separately through the Cytoscape GUI and MASS-specific extensions. Adding support for agent visualization in this work brings MASS on par with Repast Simphony as far as what objects can be visualized, but our visualizations are still limited to 2D views. Most importantly, the separation of view and execution concerns allows for the visualization to be completed asynchronously and the visualizations to be adjusted as desired through the Cytoscape GUI without the need to pre-configure or restart a computation. This is particularly useful in long-running computations where the need for visualization was not considered ahead of time or when users explore which visualizations may best fit the application.

### 4.4 CURRENT LIMITATIONS

The current implementation has three limitations. We discuss how each of these issues may be mitigated by the user or addressed through future work on the MASS library.

1. JShell provides a CLI which may be unfamiliar or awkward for users just getting started. This difficulty may be particularly acute for users who normally rely heavily on their IDE for autocorrection and suggestion support. Although these features exist in JShell their use is not as smooth as in most IDEs. These challenges will fade over time, however, as users become more familiar with the JShell interface and available features, such as the /open command which allows the user to open and run a pre-written text file. This command is particularly powerful in that it allows the user to continue developing in their chosen IDE and then simply run the /open command on their file when they are ready to test execution.
2. The current agent-tracking API allows for registration and retrieval of an entire agent class. This limitation exists because the history of agent visits is distributed amongst the places and the propagation occurs upon retrieval, once all agent tracks have been coalesced on the master node: if agent tracks are not present on the master process, then they cannot be propagated. Further, the current MASS-Cytoscape integration only allows for retrieval of full agent history. Addressing these limitations may lead to improved performance in instances where only a selection of agents, for example successful agents, is required.
3. Agent visualization in Cytoscape does not allow for much customization by the user and has only been optimized for instances where the user would like to track a single class of agents. Further, the color gradient is based on only fifteen shades of the base color which leads to instances where a vertex may show no agents present, even when they are, because the number of agents on the place is not significant enough with respect to the most populated places.

## 5 CONCLUSIONS

We implemented a set of new tools and functionality for MASS users to enable more rapid development and exploration when building agent-based graph programs. We accomplished this by introducing new APIs for tracking agent data, incorporating an interface for forward and backward computation with JShell, and expanding integration of Cytoscape for visualization of MASS *GraphPlaces* and associated *Agents*. Verification of this work was done by examining each major deliverable using Triangle Counting, which showed that the new functionality provided results consistent with previous methods and did so with minimal impact on execution performance, though greater impact on agent-data extraction times.

Our future work is two-fold in Cytoscape expansion: (1) visualization capabilities of multi-dimensional arrays and binary/quad trees and (2) filtering capabilities of agents to capture for their graph traverse, (e.g., those alive at a particular time or marked as a successful traveler).

## ACKNOWLEDGMENTS

We would like to express our appreciation to Mr. Nasser Alghamdi and Mr. Brian Luger for their In-MASS development and GraphPlaces revising work.

## REFERENCES

Alghamdi, N. (2020). Supporting Interactive Computing Features for MASS Library: Rollback and Monitoring System. Technical report, University of Washington Bothell.

Bic, L. et al. (1996). Distributed Computing Using Autonomous Objects. *IEEE Computer*, 29(8):55–61.

Davis, D. B. et al. (2018). Data provenance for agent-based models in a distributed memory. *Informatics*, 5(2):https://doi.org/10.3390/informatics5020018.

Fukuda, M., Bic, L. F., and Dillencourt, M. B. (1998). Global virtual time support for individual-based sim-

ulations. In *International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA'98*, pages 9–16, Las Vegas, NV.

Fukuda, M., Gordon, C., Mert, U., and Sell, M. (2020). Agent-Based Computational Framework for Distributed Analysis. *IEEE Computer*, 53(3):16–25.

Gilroy, J., Paronyan, S., Acoltz, J., and Fukuda, M. (2020). Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory. In *7th International Workshop on High Performance Big Graph Data Management, Analysis, and Mining (BigGraphs 2020) in conjunction with IEEE Big Data*, pages 2957–2966. IEEE.

Gordon, C., Mert, U., Sell, M., and Fukuda, M. (2019). Implementation techniques to parallelize agent-based graph analysis. In *Int'l Workshops of PAAMS 2019, Highlights of Practical Applications of S urvivable Agents and Multi-Agent Systems*, pages 3–14, Avila, Spain.

Lange, D. B. and Oshima, M. (1998). *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Professional.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA. ACM.

Mart, U. (2017). Multi-Agent Spatial Simulation (MASS) Java Library Performance Improvement for Big Data Analysis. Technical report, University of Washington Bothell.

Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., and Apra, E. (2006). Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, Vol.20(No.2):203–231.

O'Madadhain, J., Fisher, D., White, S., and Boey, Y.-B. (2003). The JUNG (Java Universal Network/Graph) Framework. Technical Report Technical Report UCI-ICS 03-17, School of Inforomation and Computer Science, University of California, Irvine.

Oracle (2017). Java Platform, Standard Edition, Java Shell User's Guide, Release 9. Technical Report E87478-01, Oracle.

Sapthy, P. and Borst, P. (1996). Wave: mobile intelligence in open networks. In *Proc of the First Annual Conference on Emerging Technologies and Applications in Communications*, pages 192–195, Portland, OR.

Wang, X. and Zhang, L. (2017). The research of the rollback mechanism in parallel simulation. In *2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)*, pages 90–95.

Wenger, M., Acoltzi, J., and Fukuda, M. (2021). Comparing thread migration, mobile agents, and abm simulators in distributed data. In *Int'l Workshops of PAAMS 2021, Practical Applications of Agents and Multi-Agent Systems*, page to appear, Salamanca, Spain.

Wilensky, U. (2013). The NetLogo NW Extension for Network Analysis, accessed on: September 3, 2021. [online]. available: http://ccl.northwestern.edu/netlogo/5.0/docs/nw.html.