Automated Agent Migration over Distributed Data Structures

Vishnu Mohan, Anirudh Potturi and Munehiro Fukuda
Division of Computing and Software Systems, University of Washington Bothell, U.S.A.

Keywords: ABM, Agent Migration, Data Structures, Data Analysis, Parallel Computing.

Abstract: In contrast to conventional data streaming, we take an agent-based approach where a large number of reactive agents collaboratively analyze attributes or shapes of distributed data structures that are mapped over a cluster system. Our approach benefits distributed graph database and GIS as agents are dispatched to data of interest and navigate over nearby structured data for further exploration and exploitation. The successful key to this approach is how to code agent propagation, forking, and flocking over data structures. We automated such agent migration in our MASS (multi-agent spatial simulation) library and wrote four benchmark programs with these migration functions. The benchmarks include breadth-first search, triangle counting, range search, and closet pair of points in space. This paper demonstrates improvements of parallel performance with the automated migration and presents our programmability comparison with Repast Simphony.

1 INTRODUCTION

Conventional big-data computing takes so-called data-streaming approach that keeps pipelining datasets to distributed data-analyzing libraries such as MapReduce and Spark. While such datasets are described in a structured file format, (e.g., CSV, XML, or JSON), they pass through distributed memory as text data rather than form distributed data structures (including graphs or geometric data space). Therefore, data streaming has challenges in repetitive data retrievals from and data analysis using distributed graph database or GIS although it is still used to feed initial data to these databases.

In contrast, we apply agent-based modeling (ABM) to analyses of distributed data structures (Fukuda et al., 2020). Our approach constructs graphs or arrays over a cluster system, dispatches a large number of reactive agents to the datasets, and have the agents compute data attributes or shapes. Applying ABM to data science has been used in bio-inspired optimization algorithms such as ant colony optimization (ACO) (Blum, 2005). However, they generally populate only tens of agents over a plain dataset. Our work is differentiated in handling millions of agents over a distributed data structure.

To verify the efficiency of our agent-based data analysis, we previously implemented benchmark programs with our MASS (multi-agent spatial simulation) library (Fukuda et al., 2020). These include breadth-first search, triangle counting, ACO, particle swarm optimization, K-means, and K-nearest neighbors. These benchmark programs revealed that our approach and MASS library need two improvements towards the practicalization (Gordon et al., 2019): spatial description and agent migration support. As we have addressed agent-navigable graph structures including their construction (Gilroy et al., 2020), pipelined computation (Hong and Fukuda, 2022), and visualization (Blashaw and Fukuda, 2022), this paper focuses on agent migration.

Agent migration we consider includes propagation, forking, and flocking over a graph and a 2D contiguous space, both mapped to a cluster system. Having reviewed the above applications, we formulated common patterns of agent migration and automated them as migration functions. Using the new automated migration, we then reimplemented breadth-first search, triangle counting, range search, and closet pair of points for our verification purposes.

This paper presents the following three endeavors of agent migration: (1) an implementation of new migration functions and platforms to support these functions, (2) performance improvements in agent-based graph and geometric computation, and (3) simplification of MASS agent code as well as programmability comparison between MASS and Repast Simphony.
phony (North et al., 2007). The rest of this paper is organized as follows: Section 2 considers conventional ABM systems’ agent migration over distributed datasets and clarifies their challenges; Section 3 designs and implements automated agent migration in MASS; Section 4 demonstrates improvements of migration performance and programmability, using graph and geometric benchmark programs; and Section 5 concludes our discussions.

2 RELATED WORK

When applying ABM concepts to analysis of distributed data structures, we need to consider execution performance and programmability for parallelization. The former has been supported by MPI-based ABM simulators represented by FLAME (Holcombe et al., 2006) and RepastHPC (Collier and North, 2013). Since their utmost goal is ABM parallel performance, their C/C++ coding frameworks do not best support programmability nor data visualization. However, most interests of data scientists are quick coding and easy visualization, for which they prefer interpretive or script languages, (e.g., Java and Python). From this viewpoint, we assume what if these scientists would use NetLogo (NetLogo Models, 1999) or Repast Simphony (North et al., 2007) as interpretive ABM systems when conducting their data analysis. Based on our assumption, we focus on their descriptive features for having agents migrate over data structures and analyze their attributes.

NetLogo (NetLogo Models, 1999) supports agent migration in a 2D continuous space and provides agents with pre-defined behaviors that can be utilized by the model designers. It includes migration functions such as FORWARD, BACKWARD, RIGHT, LEFT, HATCH, DIE, JUMP and MOVE-TO. FORWARD/BACKWARD enables agents to move forward and backward from their current position in the environment. RIGHT/LEFT enables agents to change the direction of movement. DIE removes agents from the environment. HATCH spawns new agents that inherit properties from their parent agent. MOVE-TO moves agents to a given coordinate.

Repast Simphony (North et al., 2007) is another ABM system that provides pre-defined agent migration and behavior. This includes moveByDisplacement, moveByVector, moveTo, VNContains, and MooreContains. MoveByDisplacement moves agents from their current location by a specified distance. MoveByVector moves agents by a given distance along a specified angle. MoveTo moves agents from their current location to a new location. VNContains determines whether or not a particular agent is in the von Neumann neighborhood of a particular source. MooreContains determines whether or not a particular agent is in the Moore neighborhood of a particular source.

MASS has three versions, each in Java, C++, and CUDA. For data sciences, we use MASS Java. The library distinguishes two classes: Places and Agents. The former constructs a multi-dimensional array over a cluster system, whereas the latter populates and walks agents over the array. For graph computing, users can instantiate GraphPlaces that incrementally constructs a distributed graph where agents move along its edges. The library supports basic agent-navigation and life-cycle-management functions: migrate() to move agents to a specified place index; spawn() to create new agents as inheriting their parent properties; and kill() to terminate the calling agents. All Places and Agents computation is performed in parallel through their callAll() from the main() function. In addition, Places facilitates inter-place communication with exchangeAll(), whereas Agents commits all agent creation, termination, and migration at once with manageAll().

All these ABM libraries define basic agent migration methods. However, they do not provide functions that automate agent propagation and migration over a graph or a 2D space. Users have to build custom agent migration functions that would use these basic migrations, which requires their significant programming capabilities to successfully perform data analysis. Inability to automate agent navigational functions gives a big burden to data scientists who hopes to conduct big-data computing with ABM. Knowledge of these limitations motivated us to upgrade MASS Java for supporting intelligent agent migration and propagation out of the box.

3 ABSTRACTION OF AGENT MIGRATION

This section first looks at several applications in graph computing and computational geometry to find common patterns of agent migration. Thereafter, we will explain their implementation and our infrastructural supports for automated migration.

3.1 Application-Based Agent Migration

We consider the following five categories of applications: (1) breadth-first search (BFS); (2) triangle counting and connected components; (3) range
search; (4) the closet pair of points (CPP), Voronoi diagram construction, and K-nearest neighbors (KNN); and (5) Dijkstra’s algorithms and ant colony optimization (ACO).

3.1.1 BFS

This graph algorithm disseminates agents over a graph until all vertices have been visited. Upon visiting a next vertex, an agent needs to check if the current vertex has been visited by another agent. If not, it propagates its child agents to each of all neighboring vertices except where it came from. To mitigate agent creation and termination overheads, a parent agent itself should choose one of the next vertices to visit as shown in Figure 1. We implement this feature in the migratePropagate() function.

3.1.2 Triangle Counting and Connected Components

To count the number of triangles in a graph, agents walk on consecutively connected edges three times. Those who come back to their original vertex can report that they have successfully traveled around a triangle. To avoid double-counting the same triangle, agents are supposed to choose a next vertex with a lower ID than the current vertex for the first two walks (see Figure 2). This is a special form of migratePropagate() which we name migratePropagateDownStream(). For their third walk, agents must fetch an edge that leads back to their birthplace. We implement this migration in migrateOriginalSource(). Connected components have an agent start its walk from each vertex with migrationPropagateDownStream(). As far as an agent successfully migrates to a next vertex, it concurs this vertex with the original vertex ID it is carrying from where it got started. This eventually colors all connected vertices with the largest vertex ID among them.

3.1.3 Range Search

Given a set of data points in a 2D space, range search first creates a k-d tree that recursively halves a space along the x- and y-axes in turn so that left and right or upper and lower spaces have the same number of data points. To search for data points in a range of interest, agents traverse from the tree root down to nodes whose coverage is overlapped with the given range. This computation involves a dynamic tree construction and agent propagation from the tree root. We implement such agent propagation in migratePropagateTree() that allows an agent to migrate along left, right, or both links from the current tree node. As illustrated in Figure 3, a parent agent must choose one of tree links, which saves the number of agents to be spawned.

3.1.4 CPP, Voronoi Diagram, and KNN

These applications simulate a ripple propagation from each data point, which can be mimicked by repetitively cloning agents from their current coordinates to von Neumann or Moore neighborhood in turn, as shown in Figure 4. The very first agent that encounters another data point finds the closet pair of points.
3.2 Mechanisms for Supporting Abstraction

To support the new migration functions we formulated in Section 3.1, we first modified the MASS library’s agent and place management mechanisms, as described below.

3.2.1 Agent Management

Derived from the Agent base class, SmartAgent implements not only all the new migration functions but also makes accessible additional properties: track, prev, and next, each representing a chronological ID list of places an agent navigated through so far, the ID of the previous place it visited, and that of the next place it will move to.

The new agent life-cycle management enhances agent migration with a merger of spawn() and migrate() functions. In contrast to the original MASS library that needs two invocations of Agents.callAll()/Agents.manageAll() pair, the first to commit spawn() and the second to commit migrate(), the new spawn() function can dispatch child agents to new destinations instantly in one invocation of callAll() and manageAll(). This halves thread control and inter-cluster communication overheads as each call handles agent with multi-threading and involves a cluster-wide barrier synchronization.

3.2.2 Spatial Management

SmartPlace is a subclass of the Place base class for providing a user application with additional spatial properties such as a list of neighboring places and their logical distances. These properties are used in the logic to implement Agent.migratePropagate() that clones a calling agent to all neighboring vertices.

Instead of mimicking a binary or k-d tree with GraphPlaces at a user level, we implement BinaryTreePlaces, each including left and right branch references to its child places. This new class eases our Agent.migratePropagateTree() implementation. Furthermore, it allows main() to add a new leaf to a given tree node without any user-level agent deployment that needs to manually perform this leaf addition.

3.3 An Implementation of Migration Functions

Given the new mechanisms to support agent migration, below we explain each migration function.

1. migratePropagate(): has a calling agent declare its migration to the first neighbor that is not the previous vertex (lines 9-16 in Listing 1). IDs of all the other neighbors are packetized in an argument list (lines 17-22). It is passed to spawn() (line 23) that creates the same number of child agents and dispatches each to a different vertex.
Listing 1: Propagating to unvisited neighboring vertices.

```java
public void migratePropagate(int time) {
    int[] nbrs = getPlace().getNeighbors();
    if (getPlace().footprint || nbrs.length == 0) {
        kill(); // already visited or no more edges to traverse
        return;
    }

    getPlace().footprint = true;
    int prev = -1; int next = -1;
    if (nbrs.length == 1) {
        next = nbrs[0];
    } else { // find the first neighbor, (i.e., next) to visit
      for (int i = 1; i < track.length; i++) {
        if (track[i] == prev) {
          neighbors[i] = track[i];
        } else if (neighbors[i] != track[i]) {
          migrate(neighbors[i]); // the parent agent migrates to next
          int[] dests = new int[nbrs.length - 1];
          for (int i = 1; i < nbrs.length; i++) {
            if (nbrs[i] == prev) {
              nbs[i] = nbrs[1];
            } else if (nbrs[i] == next) {
              continue;
            } else {
              dests[i] = nbs[i]; // each child gets a different dest.
            }
          }
          spawn(dests, dest); // children dispatched
        }
    }
}
```

2. migratePropagateDownStream(): needs to add "nbrs[i] > getPlace().ID" to line 19’s if statement in Listing 1.

3. migrateOriginalSource(): scans all the neighboring vertices and identifies the one whose ID is this agent’s track[0] where it needs to go back (lines 4-6 in Listing 2).

4. migratePropagateTree(): checks which of left, right, or both paths a calling agent needs to go (lines 9, 11, and 15 in listing 3). If the desired path exists, the agent can migrate along it (lines 13 and 17). If necessary, the calling agent forks a child to the right direction while taking left as its own direction (line 10).

5. migratePropagateRipple(): receives the current time to decide which of von Neumann or Moore neighborhood a calling agent should take for its 2D propagation. As shown in listing 4, if time is even, the agent propagates its copies to north,

Listing 2: Migrating to the source vertex.

```java
public void migrateOriginalSource() {
    int[] neighbors = getPlace().getNeighbors();
    boolean found = false;
    for (int i = 0; i < neighbors.length; i++) {
        if (neighbors[i] == track[0]) {
            migrate(neighbors[i]); // found the source
            break;
        }
    }
    if (!found) kill(); // no way to go back to the source
}
```

Listing 3: Migrating down to child tree nodes.

```java
public void migratePropagateTree(int path) {
    if (getPlace().footprint == true;
        int left = getPlace().left;
        int right = getPlace().right;
        switch(path) {
          case LeftBranch: // go left and dispatch a child right
            if (left != -1) spawn(1, right);
            case RightBranch:
              if (right != -1) migrate(right);
      return;
      case BothBranch: // go left and dispatch a child right
        if (left != -1 && right != -1) spawn(1, right);
    return;
    }
    kill(); // no way to go
}
```

Listing 4: Propagating over 2D.

```java
public void migratePropagateRipple(int time) {
    if (getPlace().footprint) {
        kill();
        return;
    }
    getPlace().footprint = true;
    int nAgs = (time % 2 == 0) ? 4 : 8; // #neighbors
    Vector<int[]> nbs = new Vector<int[]>(nAgs);
    int cur = getPlace().coordinates;
    if (nAgs == 8) { // agents == 8 in Moore’s
      // NE, SE, SW, and NW propagations
      inbs.get(0)[0] = cur[0]; inbs.get(0)[1] = cur[1] + 1;
      inbs.get(1)[0] = cur[0] + 1; inbs.get(1)[1] = cur[1];
      inbs.get(2)[0] = cur[0]; inbs.get(2)[1] = cur[1] - 1;
      inbs.get(3)[0] = cur[0]; inbs.get(3)[1] = cur[1];
      if (nAgs == 8) { // agents == 8 in Moore’s
        // NE, SE, SW, and NW propagations
        nbs.get(4)[0] = cur[0] + 1; nbs.get(4)[1] = cur[1] + 1;
        nbs.get(5)[0] = cur[0]; nbs.get(5)[1] = cur[1] - 1;
        nbs.get(6)[0] = cur[0]; nbs.get(6)[1] = cur[1];
        nbs.get(7)[0] = cur[0] - 1; nbs.get(7)[1] = cur[1] + 1;
      }
      spawn(nAgs, nbs); // propagate ripples to neighbors
      kill(); // no more ripple at the current coordinates
    }
```

6. migrateMax(): chooses the one with the maximum weight among all edges emanating from the current vertex. Listing 5 shows no special techniques but can save lines of user-level code.
Listing 5: Migrating along the max-weighted edge.

```java
public void migrateMax() {
    int[] neighbors = getPlace().getNeighbors();
    int[] weights = getPlace().getWeights();
    int weight = Integer.MIN_VALUE;
    int maxIndex = 0;
    for (int i = 0; i < weights.length; i++) {
        if (weight < weights[i]) {
            weight = weights[i];
            maxIndex = i;
        }
    }
    migrate(neighbors[maxIndex]);
}
```

Table 1: Benchmark programs and test parameters.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Applications</th>
<th>Test Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>MigratePropagate</td>
<td>BFS</td>
<td>Graphs with 100, 500, 1K, and 2K vertices</td>
</tr>
<tr>
<td>MigratePropagateDownStream</td>
<td>Triangle Counting</td>
<td>Graphs with 1K, 1K, 10K vertices</td>
</tr>
<tr>
<td>MigrateOriginalSource</td>
<td>Triangle Counting</td>
<td>Graphs with 1K, 1K, 10K vertices</td>
</tr>
<tr>
<td>MigratePropagateTree</td>
<td>Range Search</td>
<td>Trees with 100, 100K, and 200K vertices</td>
</tr>
<tr>
<td>MigratePropagateTree</td>
<td>Range Search</td>
<td>Trees with 100, 100K, and 200K vertices</td>
</tr>
<tr>
<td>MigratePropagateTree</td>
<td>Range Search</td>
<td>Trees with 100, 100K, and 200K vertices</td>
</tr>
<tr>
<td>MigratePropagateRipple</td>
<td>CPP</td>
<td>2D continuous space with 64, 100K, and 200K data points</td>
</tr>
</tbody>
</table>

4 EVALUATION

Our performance and programmability evaluation used four benchmark programs: BFS, triangle counting, range search, and CPP. The first two programs are graph computing, whereas the last two are categorized in computational geometry.

For the performance evaluation, we coded these four benchmark programs with MASS Java in two versions: one without using the automated migration functions and the other using these functions. (In the following discussions, we distinguish them as legacy MASS and new MASS, respectively.) Then, we ran them on a cluster of four Linux machines, each with a Xeon Gold 6130 @ 2.10GHz processor and 16GB memory. The measurements were repeated five times for each of spatial parameters as listed in Table 1.

For the programmability evaluation, we first compared legacy and new MASS versions to demonstrate substantial improvements when using the automated migration. We then coded the same benchmark programs with Repast Simphony, which allowed us to compare MASS and Repast Simphony in their programmability.

4.1 Execution Performance

Figures 6 through to 10 demonstrate how the new MASS version brings performance impacts onto BFS, triangle counting, k-d tree construction followed by range search, and CPP. Below we analyze parallel performance of each benchmark program.

4.1.1 BFS

As shown in Figure 6, the execution time for migratePropagate() is lower than that taken by the legacy migration. Their performance difference diverges as the number of vertices in a graph increases. The performance improvement is primarily due to the usage of the new manageAll() that spawns and moves child agents as a part of a single execution. The more graph vertices there are, the more agents need to be spawned for migration, which is efficiently supported by the new manageAll().

4.1.2 Triangle Counting

Figure 7 demonstrates the performance improvements by migratePropagateDownStream() and migrateOriginalSource(). Since these two automated functions are rather restrictive to spawning agents than migratePropagate(), they did not reduce execution time so drastically.

4.1.3 K-D Tree Construction and Range Search

The legacy MASS is not scalable enough to construct a k-d tree covering 100K or more data points. On the other hand, as mentioned in Section 3.2.2, the new MASS increased the tree construction scalability so as to handle beyond 100K data points. Figure 8 shows...
almost a linear increase of tree-construction time with the new MASS version. Using $migratePropagateTree()$, the new MASS clearly demonstrates the $O(n \log n)$ performance of range search from 100 to 200K data points (see Figure 9).

### 4.1.4 CPP

Figure 10 compares the legacy and new MASS versions for their CPP execution performance. Both versions show little difference in their execution time. This is because a ripple propagation does not increase agent population so drastically as observed in BFS. Furthermore, the more data points there are, the higher the chances an agent encounters the nearest data point within less simulation cycles.

In summary, the automated agent migration mainly improves the execution performance and spatial scalability of graph propagation and tree traverse by agents.

### 4.2 Programmability Evaluation

Tables 2, 3, and 4 respectively compare the programmability between legacy and new MASS versions, between new MASS and Repast Simphony, and across ABM libraries.

#### 4.2.1 Improvements Within MASS

We counted each benchmark program’s lines of code (LoC) by removing all blanks and comments from the source code. Then, we measured how much reduction was made for each benchmark program from legacy to new MASS versions. Table 2 shows that BFS and triangle counting achieved the highest percentage of lines removed (83% and 60% respectively) while CPP had the lowest reduction, (i.e., 22%). This is because $migratePropagate()$, $migratePropagateDownStream()$, and $migrateOriginalSource()$ were able to fully abstract the agent navigation and propagation from BFS and triangle-counting logic.

While CPP utilized the $SpaceAgent$ class that automated propagation to the von Neumann or Moore neighborhood, it still needed user-level logic to remove redundant agents that were duplicated from the previous ripple propagation. For instance, agents dispatched from west to east and from south to north may collide each other at the same coordinates, in which case one of them must be removed. At present, these redundancy check and agent removal operations are not supported by the new migration functions.
Table 2: LoC reduction with automated agent migration.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Applications</th>
<th>LoC Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MigratePropagate</td>
<td>BFS</td>
<td>83%</td>
</tr>
<tr>
<td>MigratePropagateDownStream</td>
<td>Triangle Counting</td>
<td>60%</td>
</tr>
<tr>
<td>MigrateOriginalSource</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MigratePropagateTree (BothBranch)</td>
<td>Range Search</td>
<td>44%</td>
</tr>
<tr>
<td>MigratePropagateTree (LeftBranch)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MigratePropagateRipple</td>
<td>CPP</td>
<td>22%</td>
</tr>
</tbody>
</table>

4.2.2 MASS vs Repast Simphony

New MASS no longer requires benchmark programs to write fine-grain instructions to support automatic agent migration, tree traversal, and 2D propagation. Instead, they are built upon these new features. On the other hand, for Repast Simphony, we had to develop the AgentManager class from scratch as it was essential to all its benchmark programs but was not readily available. As the name suggests, AgentManager is responsible for basic agent-management operations including agent creation, termination, and migration control. Implementing the same auto-navigation as MASS Java in Repast Simphony for each benchmark program requires slight modifications to this base code.

To evaluate the programmability of these applications, LoC is a measure used to identify how much programming must be done to implement an application. Cyclomatic complexity is another metric used to determine the complexity of a program based on the number of logical paths in it. The LoC and Cyclomatic complexity are good metrics to explain better how much complex code one needs to develop for an application. Agent LoC represents LoC used to define the patterns of agent behavior and their tasks. Space LoC is code used to define the space within which the problem is distributed and built on.

As detailed in Table 3, the overall LoC in Repast is higher than MASS, (i.e., 1545 versus 1056 in total) because of the need for code defining the structural elements. Agent LoC is higher in Repast, (i.e., 553 versus 274) because of the mandatory use of AgentManager, much of which contributed to the Agent LoC. An agent reacts to a change in its environment. Hence, Space LoC is higher, too in Repast, (i.e., 378 versus 186) because this section is used to develop the space and acts as a point of invocation of method calls to AgentManager. This process is simple yet seeks fine-grained instructions, as mentioned above. By choice of design, CPP in MASS used an additional class, taking advantage of the strengths of object-oriented programming in storing the CPP results; unlike in Repast, a simple array of points was used. Repast’s CPP saw reduced code (i.e., 314 versus 362) because of the simplified approach to recording the results. The Cyclomatic complexity in MASS is increased through iterations and conditionals. It ranges from 2.25 to 3.944 as compared to Repast’s Cyclomatic complexity in 1.785 through to 2.6. The decision-making process contributes to this by calling the appropriate agent migration methods followed by other utility methods. The repeated use of conditionals can be closely associated with MASS’ object-oriented approach and switch statements in calling appropriate base methods. Lastly, in BFS with Repast, the modified AgentManager increased Agent LoC. Space LoC was majorly increased because, unlike other implementations, the generation of vertices and their neighbors was in-built into the application.

4.2.3 Availability of Automated Agent Migration Across Products

We also compared the availability of automated agent migration methods with Netlogo and Repast Simphony. As summarized in Table 4, with the introduction of the new automated agent navigation, MASS now has the greatest number of advanced agent migration methods. Repast Simphony supports certain agent navigational methods such as Shortestpath, MoveAgentByDisplacement and MoveAgentByVector that are currently not supported in MASS. However, SmartAgent in MASS can be easily extended to incorporate these agent navigational patterns.

Based on our evaluation in this section, we can conclude that the new automated agent migration along with the improvements in the agent/spatial management has brought significant performance improvements to MASS while executing the benchmark programs. The new automated agent migration has also made MASS easier to use and has improved...
Table 4: Availability of automated agent migration across products.

<table>
<thead>
<tr>
<th>Methods</th>
<th>NetLogo</th>
<th>Repast</th>
<th>MASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MigratePropagate</td>
<td>No</td>
<td>Yes²</td>
<td>Yes</td>
</tr>
<tr>
<td>MigratePropagateDownStream</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MigrateOriginalSource</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MigratePropagateTree</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MigratePropagateRipple</td>
<td>Yes²</td>
<td>(Yes)³</td>
<td>Yes</td>
</tr>
<tr>
<td>MigrateMin</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MigrateMax</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MigrateRandom</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ShortestPath</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MoveAgentByDisplacement</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MoveAgentByVector</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

¹ Repast Simphony can perform breadth first search.
² NetLogo supports Voronoi diagram and K-Nearest Neighbor.
³ Repast Simphony can check if an agent is present in the von Neumann or Moore neighborhood.

5 CONCLUSIONS

Focusing on agent-based graph computing and computational geometry, we found common agent migration patterns from several benchmark programs and formulated them as high-level migration functions. Using these migration functions, we re-coded the same MASS benchmark programs. Our performance and programmability measurements demonstrated that the automated agent migration not only accelerated the execution but also improved the programmability as compared to Repast Simphony.

To further extend the work we have completed, we see the following three opportunities. The first plan intends to introduce additional agent navigation functions including:

- **PropagateRippleWithBouncing**: to support the calculation of Euclidean shortest path between two points over contiguous space by propagating a ripple from the source point and bouncing off opaque obstacles until the ripple detects the destination point.
- **MigrateLowestCoordinatePoint and MigrateUnboundedRegion**: to construct a convex hull by moving an agent to the starting coordinate point and thereafter by walking the agent to the Voronoi site present in the unbounded Voronoi region.
- **MoveAgentByDisplacement and MoveAgentByVector**: to move an agent from its current location by a given distance over a continuous space or to move an agent by the distance along a specified angle.

Our second plan is to evaluate MigrateMin, MigrateMax, and MigrateRandom with additional benchmark programs such as Dijkstra’s algorithm. Finally, our third plan is re-implementations of more benchmark programs including Voronoi diagram and convex hulls in MASS, using the automated agent navigation methods.

REFERENCES


