Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory

Justin Gilroy, Satine Paronyan, Jonathan Acoltzi, and Munehiro Fukuda
Computing and Software Systems
University of Washington Bothell
Bothell, WA 98011
{jegilroy, sparon, jacoltzi, mfukuda}@uw.edu

***Abstract*—Some graph analyses, such as social network and biological network, need large-scale graph construction and maintenance over distributed memory space. Distributed data-streaming tools, including MapReduce and Spark, restrict some computational freedom of incremental graph modification and run-time graph visualization. Instead, we take an agent-based approach. We construct a graph from a scientific dataset in CSV, tab, and XML formats; dispatch many reactive agents on it; and analyze the graph in the form of their collective group behavior: propagation, flocking, and collision. The key to success is how to automate the run-time construction and visualization of agent-navigable graphs mapped over distributed memory. We implemented this distributed graph-computing support in the multi-agent spatial simulation (MASS) library, coupled with the Cytoscape graph visualization software. This paper presents the MASS implementation techniques and demonstrates its execution performance in comparison to MapReduce and Spark, using two benchmark programs: (1) an incremental construction of a complete graph and (2) a KD tree construction.**

***Index Terms*—multi-agent systems, agent-based modeling, data analysis, data visualization, parallel programming**

I. Introduction

In response to most demands for big-data computing, particularly in business, the major software tools such as MapReduce [1], Spark [2], and Storm [3] have focused on streaming unstructured text data to their multithreaded analyzing units: *map/reduce* functions in MapReduce, *transformations/actions* in Spark, and *spouts/bolts* objects in Storm. Their simple programming frameworks have also attracted physical scientists in need of their structured data analyses. The key to success is obviously how to fill the gap between structured data analyses and data-streaming tools. To address this issue, SciHadoop [4] serializes and partitions a NetCDF [5] array-oriented scientific dataset, (e.g., climate data) into smaller chunks that are then fed to MapReduce. GraphX [6] facilitates a graph description as Spark's resilient distributed dataset (RDD) to be loaded over a Spark cluster.

However, this approach to interfacing to the conventional tools still suffers from the following two challenges: (1) a dataset cannot remain in memory for incremental modifications and (2) specific data items and their relationship cannot be visualized at run time.

Contrary to this data-streaming approach, we proposed an agent-based framework for distributed data analysis in [7], where analyzing units (i.e., agents) navigate over and even modify a structured dataset in memory at run time, for the purpose of repetitive discovery of the structural attributes, such as the shortest path [8] and the number of triangles [9] in a given graph. We named this approach *agent-based data discovery*. Using the multi-agent spatial simulation (MASS) library [10], we demonstrated its programming and performance advantages over the data-streaming approach. First, agent-based programs can be coded intuitively as typical collective group behaviors in agent-based modeling (ABM): agent propagation, flocking, and collision. Second, some agent-based algorithms can empirically run faster than or are competitive with MapReduce and Spark. For example, in the closet pair problem [11] and connected component search [12], each is implemented as agent collision detection and propagation.

While distributed arrays such as GlobalArray [13] are available and applicable for agents to navigate over, there are few software tools to automate the construction of agent-navigable, distributed graphs except using parallel ABM simulators (e.g., RepastHPC [14]). Furthermore, the visualization of a distributed graph and agents traversing on it still relies on each user's capability. These drawbacks are confirmed by Gordon et. al. [9] to give quite negative impacts to agents' programmability and their total execution time.

Given this background, this research focuses on agent-navigable graph construction and maintenance over a cluster system and realizes the following three graph features in the MASS library: (1) graph construction from different file formats such as CSV, HIPPIE tabs [15], and MATSim [16] XML, (2) both interactive and pre-coded graph modifications over distributed memory, and (3) graph visualization by interfacing the MASS library to Cytoscape [17].

The rest of this paper is organized as follows: section II looks at the current trends in graph processing and proposes an agent-based approach; section III gives the details of our implementation of an agent-navigable graph structure and its visualization; section IV compares our implementation with MapReduce and Spark; and section V summarizes our achievements.

II. Related Work

We first look at several achievements for applying ABM to data sciences beyond its original pursuit of agent-based microsimulation. Second, we examine graph construction and

maintenance strategies taken by data-streaming tools and thread migration. Third, we summarize some graph visualization tools. Finally, we identify the current challenges in graph maintenance over a commodity cluster system.

A. ABM as Data Analyzing Framework

Since highlighted by Swarm [18], ABM has been used to run environmental and social simulation for observing an emergent collective group behavior among many simulation entities called agents. Scientists in operations research are applying the same concept to scheduling problems, engineering optimization, and data sciences. This approach is called biologically inspired algorithms. The particle swarm optimization (PSO) [19] and the grasshopper optimization algorithm (GOA) [20] use agents flocking and repelling features to explore and exploit optimized solutions such as cluster centroid identification. Ant colonial optimization (ACO) [21] implements a pheromone-attracted foraging capability in agents in order to solve the traveling salesman problem (TSP) and optimizes network-routing optimization in a reasonable time range. However, for big-data computing, most work has put their paramount focus on orienting new parallelization techniques to MapReduce [22], [23] and Spark [24], [25], thus handling single-use text inputs and leaving the data distribution to these underlying infrastructures.

To apply ABM to repetitive analyses of structured datasets in memory, we would end up using parallel ABM simulators such as FLAME [26] and RepastHPC [14]. The former simulates an interaction among communicating agents: each static to a given MPI rank, maintaining the entire simulation environment, and behaving as a finite-state machine. The latter creates a lattice, a network, or a contiguous space projection; maps it over an MPI cluster; and observes agent movements in the projection. While FLAME cannot globally map a dataset to a cluster system nor move agents over the data, RepastHPC may have potential to distribute a dataset as a projection and let agents navigate it over for data discovery.

Unfortunately, RepastHPC has the following four difficulties when being used for data sciences rather than ABM simulation: (1) the computation cannot stop under a certain condition instead of using a given simulation tick; (2) all I/O operations are performed sequentially by the master computing node (i.e., rank 0); (3) a projection is not modifiable during simulation; and (4) no visualization or state-checking tools are available to keep track of agent movements or to observe the on-going state of a projection (i.e., a dataset).

B. Graph Processing with Big Data Tools

From the inception of MapReduce [1], its users have developed techniques to extend their text-processing algorithms to graph problems. The typical approach introduced in [27], [28] solves a graph problem by repetitively exchanging information among neighboring graph vertices until all vertices eventually fall into a stable state that emits no more new information. This vertex-oriented approach can be implemented by invoking *map()* to diffuse each vertex's status change to neighboring vertices and then calling *reduce()* to collect such status changes from the neighbors. Therefore, map/reduce invocations must be repeated until a given network reaches a stable state. Tez [29] addressed this demand for facilitating repetitive MapReduce.

Spark [2] can take the same vertex-oriented approach as MapReduce, using its transformations similar to *map/reduce* such as *flatMapToPair()* and *reduceByKey()*. GraphX [6] alleviates a steep learning curve required for users to handle graphs with Spark, by defining a graph in VertexRDD and EdgeRDD and facilitating popular graph functions. For user-customized graph problems, GraphX implements the Pregel [30] API, which automates repetitive message passing from a vertex to its neighbors, using its *sendMsg()* and *mergeMsg()* functions.

Other MapReduce and Spark users focused on graph edges rather than vertices, and developed their own graph algorithms by recursively narrowing down candidate edges as normal text lines [31]. For instance, triangle counting can be implemented by examining all the edge connectivities, narrowing them down to triads, identifying those that create a triangle, and removing duplicates.

However, all the above approaches are expensive and time-consuming. Basically, users are responsible for rerunning MapReduce with a revised input file, rebuilding a different version of RDD, and extracting graph states to observe by invoking additional *reduce()* functions or Spark actions. One solution is IncRDD [32], a Spark RDD extension that relieves users' burdens of new RDD construction as well as speeds up such automated RDD modifications using Cuckoo Hashing [33]. Yet, users cannot directly observe which vertices and/or edges have been affected.

Looking at thread migration, EMU [34] constructs a large scale graph on its distributed memory and orchestrates thread migration over the graph. It emulates a graph with compressed sparse row representation (CSR) [35] and dispatches new threads to remote array elements in Cilk Plus [36]. EMU takes the closest approach to agent-navigable distributed graph construction that we are aiming for. On the other hand, it differs from the MASS library in: (1) the nature of CSR-based graph emulation does not support incremental graph modification; (2) thread migration in Cilk only spawns child threads remotely but does not move active threads; and more importantly (3) EMU needs its custom hardware.

C. Graph Visualization

ABM visualization has been made available in various single-process simulators such as MASON [37] and Repast Simphony [38] but not so much in parallel ABM systems. MASON's GUI is a Java-based tool that runs separately from the simulation platform, capable of drawing a network in a 2D or a 3D space with generic visualization functions and thus applicable to any other ABM simulations. However, its general aspect of visualization gives users a steep-learning curve to master its GUI. On the other hand, Repast Simphony combines the GUI into its IDE. Its graph model is based on a

projection of the data space, and thus the GUI-based graph construction starts with the "Projection" menu bar. A user clicks the "Scenario Tree" menu, chooses a network from "Displays", and sets up how to generate the outputs from the "Data Sets" menu. Despite its convenience, the Repast Simphony's GUI is not portable to any other ABM simulators.

Pavlopoulos et.al. [39] empirically compared the four major graph visualization tools: Cytoscape [17], Tulip [40], Gephi [41], and Pajek [42]. They reported that Pajek demonstrates the best scalability and the fastest visualization speed while leaving some space to enhance manual vertex/edge editing and plug-ins from other graph applications. It is capable of handling 100 million graph vertices that can even be extended to two billion with Pajek64-XXL. Needless to say, its capability is maximized with the underlying memory space. Looking at Stanford Large Network Dataset Collection [43], the large biological network [44] consists of only 8+ million vertices but occupies 69GB in its full file size. This, in turn, indicates that not only the visualization tools' capability but also available memory space (preferably supported by distributed memory) is of importance to maintain and visualize large graphs.

D. Technical Challenges

The above survey in Sections II-A through to II-C implies agent-based graph analysis needs the following three features all to be blended into one implementation:

1) **Graph construction and maintenance over distributed memory:** scales up the graph size and allows agents to use more memory space for data discovery, conducted through their group behavior;
2) **Incremental graph updates:** allows users to repeat trial-and-error computation on a given graph in an interactive fashion; and
3) **Interface to visualization tools:** sends graph data to a visualizer for zooming in and out of the graph as well as receives graph updates from the visualizer for incrementally constructing a graph over distributed memory.

We have implemented these three features in the MASS library and have measured its execution performance. The following two sections describe our achievements.

III. Implementation

A. MASS Library

MASS [10] approaches ABM with two classes: *Agents* and *Places*. The former represents a collection of mobile objects in a simulation, each named *agent*. The latter represents a multi-dimensional array space of entities, each named *place*. MASS users initialize places with their dataset and populate agents that can autonomously traverse places as the logical space. Parallelization in MASS is facilitated by distributing a different subspace of Places across cluster nodes and further splitting a node's portion of the subspace between multiple threads on each node. This distribution of the data-set across multiple computing nodes gives MASS the ability to simulate large logical spaces beyond what can be represented on a single node. Agents can then seamlessly migrate between places regardless of the node or thread they are associated with.

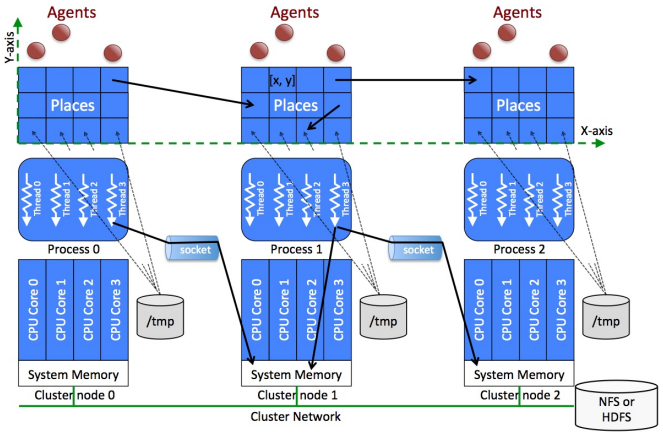


Fig. 1. MASS library architecture

Figure 1 shows the distribution of places across a cluster of three computing nodes, each with four threads of execution to manipulate the places and agents in parallel. Communication between each node is carried out through TCP sockets. A collection of agents are associated with places. Parallel computation onto places and agents are enabled with *Places.callAll(func)* to call a given function of all places, *Agents.callAll(func)* to call a given function of all agents, *Places.exchangeAll(func)* to exchange data among neighboring places through a function call, and *Agents.manageAll()* to commit agent creation, termination, and migration that have been scheduled in the last *callAll()*. For better programmability, we facilitated *Agents.doWhile()* and *doUntil()* that eliminate repetitive notations of *Agents.callAll()* and *manageAll()*.

MASS provides users with parallel file I/O features [8] that access NetCDF and text files in parallel. Each place can open an identical file and read/write only its corresponding file data using the following built-in functions: *Place.open(filename)*, *read(fd)*, *write(fd, bytes)*, and *close(fd)*, where *fd* is a file descriptor. A large dataset is partitioned and distributed to each cluster node's */tmp* directory, in support with either MASS parallel I/O tools or HDFS.

Previously in MASS, users were responsible for emulating a graph with Places. One emulation was to implement an adjacency matrix with a 2D array of places [45], while the other was to define a 1D array of places, each representing a graph vertex and maintaining an adjacency list, namely a list of logically adjacent place indices [8]. In the former approach, an agent scanned the current row of the matrix and cloned itself to other rows enumerated as the logically adjacent vertices. In the latter approach, an agent traversed a graph by moving from one place to another as referring to each adjacency list.

In summary, MASS smoothly transitioned from ABM simulation to multi-dimensional dataset analysis but left the graph-to-array mapping to the users of the library.

B. Graph Programming with MASS

To promote agent-based graph analyses with MASS, we extended the Places class to *GraphPlaces*. Listing 1 gives an example code snippet that propagates agents over a biological network to search for network motifs. The GraphPlaces constructor reads a given input data, (e.g., "input.txt" in Listing 1) and builds a graph as a 1D distributed array of places, each with an adjacency list (lines 5-6). It currently distinguishes CSV, HIPPIE tab, MATSim XML, and key/value formats. The key/value format (indicated as KEYVALUE) uses a vertex ID as a key and lists all the adjacent vertices as the corresponding value. This file format can be pre-partitioned and placed at each computing node's /tmp directory so that the MASS parallel I/O reads graph data in parallel. Once a GraphPlaces object is mapped over a cluster system, the main program can add an additional vertex to the graph (line 7) and establish a new edge to the vertex (lines 8-9). All vertices can invoke a given function in parallel through *callAll()* (line 10) and exchange data with their neighbors through *exchangeAll()* (line 11).

Assuming Listing 1 defines 100 vertices in "input.txt" and adds one more vertex to the graph, the code populates 101 agents, each starting from a different vertex (line 12). Then, the main program repeats the logic of motif search agents until no agents remain in the graph (line 13). The actual agent logic is implemented in an independent class: for example in *Motif* (lines 16-19). The *@onArrival* annotation invokes *Motif.walk()* every time an agent migrates to a new vertex [46].

Listing 1. MASS graph creation

```
1 import MASS.*;
2 public class GraphCreation {
3   public void main(String[] args) {
4     MASS.init( );
5     GraphPlaces graph = new GraphPlaces(1, "Bionet",
6               "input.txt", KEYVALUE, PARALLEL_LIST);
7     graph.addVertex(new Integer(100));
8     graph.addEdge(new Integer(99), new Integer(100),
9                   new Integer(0));
10    graph.callAll(updateState);
11    graph.exchangeAll(forwardMsg);
12    Agents searcher = new Agents(2, "Motif", graph, 101);
13    crawler.doWhile(()->crawlers.hasAgents());
14    MASS.finish();
15 } }
16 public class Motif extends Agent {
17   @onCreation public void init( ) { ...; }
18   @onArrival public void walk( ) { ...; }
19 }
```

C. Technical Challenges of GraphPlaces Implementation

The non-deterministic nature of graph construction and maintenance causes the following two technical challenges:

1) The run-time growth of graph size through vertex/edge additions and deletions
2) The unsorted definition of vertices and edges in an input file

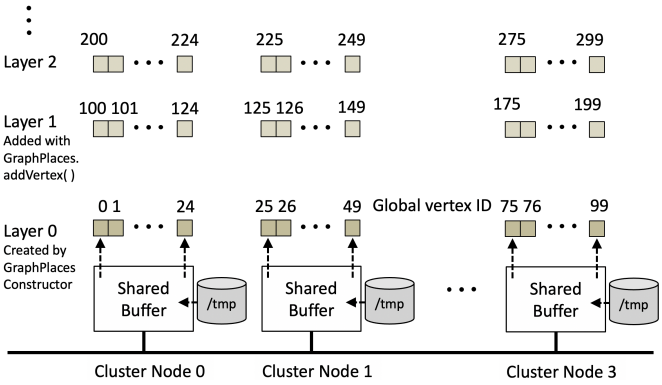


Fig. 2. GraphPlaces Implementation

1) Solution to Incremental Graph Growth: We implement GraphPlaces as a collection of *Places* objects where the first object is internally created with the GraphPlaces constructor, whereas the following Place objects are automatically added to GraphPlaces as needed through *addVertex()* calls. Figure 2 illustrates the runtime additions of Places objects in GraphPlaces. The figure breaks down a cluster of four computing nodes. Layer 0 is the very first 1D Places array. Layers starting from 1 are run-time additions to GraphPlaces. Each graph layer imitates the MASS layer's size and allocates a fraction of the size to each computing node in the cluster. In other words, if we map 100 vertices to a cluster system containing four computing nodes, each node would be allocated one quarter of the size (i.e., 100 places) as their portion of the data set. In this scenario, each computing node would be allocated 25 places. Therefore, node 0 would be allocated indices 0-24, node 1 indices 25-49, node 2 indices 50-74, and node 3 indices 75-99. Maintaining this 100-place scenario, the place indices 0-99 represent layer 0. For additional vertex insertions, we repeat this association in a round-robin fashion where the next allocation of 25 places, in this particular scenario, would be associated to computing node 0 at layer 1.

Based on Table I, this implementation maps a 0-indexed global vertex ID in GraphPlaces (i.e., $globalVID$) to the corresponding triplet ($computeID$, $layerID$, $localVID$), each respectively referring to the cluster-computing node ID, the layer ID, and a local vertex ID within the layer. Using this addressing algorithm, we can store a single integer in the cluster-wide distributed map to allow mapping an arbitrary vertex ID to a unique place in MASS that is directly addressable. The layering allows the use to expand the graph well beyond the initial size.

2) Solution to Unsorted Vertex/Edge Definition in an Input File: The KEYVALUE file format as shown in Listing 1 is our proof-of-concept file definition. Each line of an input file begins with a unique vertex ID, followed by a list of adjacent vertex IDs. It assumes that no identical vertex ID will appear twice or more times to define additional neighbors. This format makes it easy to sort and map vertices to a

TABLE I
PARAMETERS TO IDENTIFY A GIVEN VERTEX IN GRAPHPLACES

Parameters	Remarks
globalVID	0-indexed global vertex identifier in GraphPlaces
clusterSZ	# computing nodes in a given cluster system
initSZ	The size of layer 0 of a GraphPlaces object (i.e., # vertices initialized by the constructor)
stripeSZ	Layer 0's fraction allocated to each computing node (i.e., $initSZ \div clusterSZ$)
computeNID	Computing node's identifier in the cluster (i.e., $(globalID \bmod initSize) \div stripeSZ$)
layerID	0-indexed layer identifier (i.e., $globalID \div initSZ$)
localVID	0-indexed vertex identifier within a given layer (i.e., $globalID \bmod stripeSZ$)

cluster system. Similarly, the CSV format defines an adjacency matrix, which can be read in parallel with slight modifications of the KEYVALUE format. In MATSim XML, vertex IDs are provided as 1-indexed sequential integers first, and thereafter all their edges are listed. Therefore, the MATSim XML format allows easy manipulation of the vertex IDs with a simple offset of 1.

The HIPPIE format, in contrast, contains two possible attributes for each vertex: a string and a non-sequential integer. The integer attribute represents an incomplete range of integers (sampled from 1 up to 100,820,829 seen in the hippie_current.txt dataset [47]). This format revealed a significant deficiency in the existing pattern of mapping vertices to ordinals compared to the other file formats.

The issue at hand is that the MASS library does not have a hashing method that allows mapping a given vertex ID, string, or integer consistently across a cluster system. This problem is compounded by our goal of supporting dynamic modification of the graph; therefore, a pre-calculated complete hash was not an option. A solution that became clear to us was to implement a distributed map in MASS to allow consistent hashing across the cluster system or to utilize an existing library with such distributed map capability included. We elected to use Hazelcast [48] for this project. The mapping solution we came up with maps an object key to an integer that represents the global index for the place associated with a given key. This allows users to reference the vertices in their data as expected: the logical name of the vertex. MASS can then use the logical name of the vertex as a key into the distributed map to retrieve the global index of the place that represents the vertex.

D. MASS and Cytoscape Integration

The MASS library uses Cytoscape as a GUI tool for interactive graph construction, editing, and visualization. The two main reasons are: Cytoscape's popularity among scientific users and its well-defined plug-ins. Integrating MASS with Cytoscape requires two key interactions between the two systems through Cytoscape plug-ins:

1) *Import-network plug-in:* pulling an in-memory graph from MASS and re-building it inside of Cytoscape into a native CyNetwork with a combination of CyNode and CyEdge entities.
2) *Export-network plug-in:* serializing the currently selected Cytoscape CyNetwork into the MASS GraphPlaces for sending to MASS.

To facilitate communication between Cytoscape and MASS, a user must run *CytoscapeListener* in the MASS main program as shown in Listing 2 on line 7. Then, Cytoscape plug-ins establish a TCP socket to communicate with MASS. Since a user interacts with Cytoscape on the graph visualization, Cytoscape invokes both import- and export-network plug-ins. They respectively send the following commands to MASS, followed by a transfer of a lightweight graph representation with only essential information such as the vertex IDs and their neighbors:

1) *getGraph:* initiates retrieving the in-memory graph from MASS that sends the serialized message to Cytoscape.
2) *setGraph:* sends the current CyNetwork to MASS that rebuilds the corresponding GraphPlaces object inside of it.

These two commands are interpreted by CytoscapeListener automatically and are applied to a given graph in parallel of the main program. With its command interpretation in background, a user can repeatedly observe the progress of graph computation (line 8 in Listing 2). As we are planning to merge GraphPlaces and CytoscapeListener into the interactive version of MASS [49] running on top of JShell [50], a user will be able to checkpoint the on-going graph computation on JShell and to view the latest status through Cytoscape.

Listing 2. MASS main to interact with Cytoscape

```
1 import MASS.*;
2 public class GraphVisualization {
3   public void main(String[] args) {
4     MASS.init( );
5     GraphPlaces graph
6                   = new GraphPlaces(1, "EmptyGraph", 100);
7     MASSListener listener = new CytoscapeListener(graph);
8     ...; // Graph computation
9     listener.finish();
10    MASS.finish();
11 } }
```

To implement the above Cytoscape-MASS interaction, we need to address two technical challenges: (1) graph model conversion from MASS to Cytoscape for visualizing a graph and (2) Cytoscape-initiated on-the-fly interaction with MASS, (i.e., runtime getGraph/setGraph invocations against MASS). The following describes our solutions to these two challenges:

1) Graph model conversion from MASS to Cytoscape: The Cytoscape graph model does not use an adjacency matrix nor adjacency lists. Instead, it uses two separate tables: one for vertices and the other for edges. While each table row corresponds to a different entity - either a vertex or an edge - the table columns encode the vertex or edge attributes. The plug-in developers are given the ability to add custom columns to the tables but are limited to primitive types and lists of

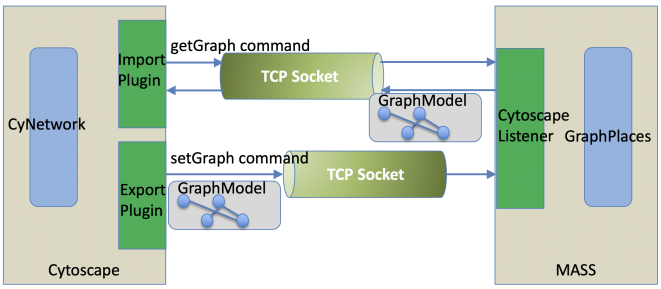


Fig. 3. Communication between MASS and Cytoscape

primitive types. To convert MASS GraphPlaces into Cytoscape graph model, the default columns on both the vertex (node in Cytoscape) and edge tables are adequate. Another challenge we faced is the styling of vertices as they appear in Cytoscape. The default visual representation of vertices is not entirely representative of a unique vertex in a network, so a simple styling transform is applied to incoming graph data to assign names and interactions to the vertices.

2) Cytoscape-Initiated Interaction with MASS: Cytoscape needs to invoke getGraph and setGraph commands in the MASS cluster and to elicit responses. For this purpose, we elected to implement a simple remote-procedure call (RPC) interface in CytoscapeListener. This implementation allows Cytoscape to send a command to MASS as a simple string, which prompts CytoscapeListener to parse the command and to respond accordingly. Figure 3 illustrates RPC calls from Cytoscape to CytoscapeListener in MASS.

These RPC calls are actually made from Cytoscape's import/export plug-ins that are implemented in support with Open Service Gateway Initiative (OSGI) [51]'s modularized Java components. A Cytoscape plug-in is principally built out of three required classes: an activator, a task factory, and a task. Our import-network and export-network plug-ins are implemented as follows:

- **Activator:** works as an entry point of a plug-in, assigns services to a given factory, and associates the factory a menu option.
  - *Import-network plug-in:* creates $MASS \rightarrow Import\ Network$ menu option.
  - *Export-network plug-in:* creates $MASS \rightarrow Export\ Network$ menu option.
- **Factory:** upon a call from a user's menu selection, creates an instance of *Task* class to process a request.
  - *Import-network plug-in:* passes required Cytoscape APIs to an import-network task
  - *Export-network plug-in:* passes required Cytoscape APIs to an export-network task
- **Task:** runs an actual plug-in logic.
  - *Import-network plug-in:* sends getGraph to MASS and de-serializes a graph message into CyNetwork.
  - *Export-network plug-in:* sends setGraph to MASS, serializes CyNetwork, and forwards it to MASS.

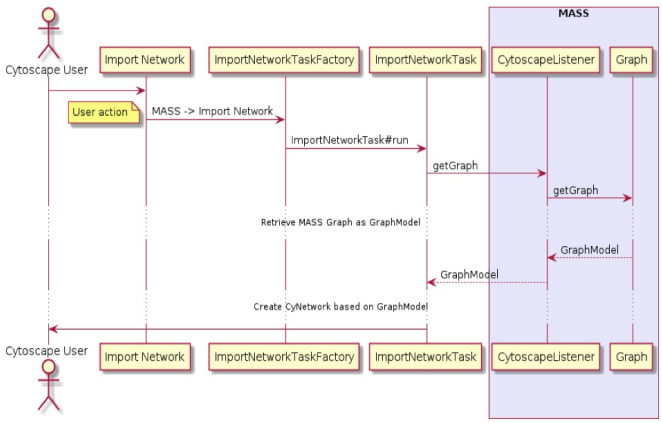


Fig. 4. An interaction diagram for importing a MASS graph into Cytoscape

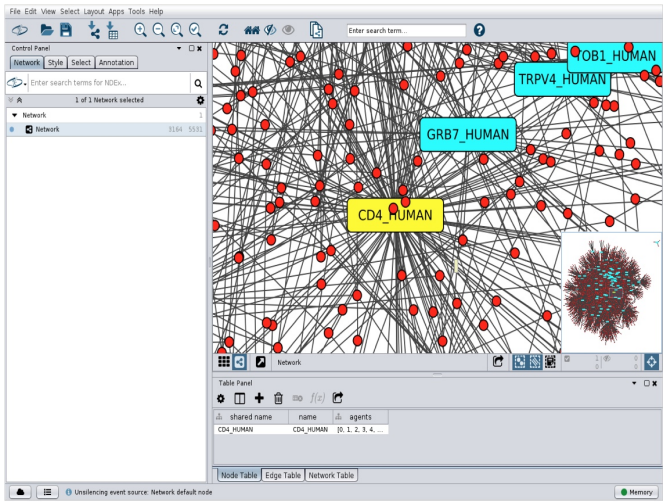


Fig. 5. HIPPIE current dataset imported from MASS into Cytoscape

Figure 4 is an interaction diagram for a Cytoscape user to visualize a MASS GraphPlaces object in Cytoscape. As shown in Figure 5, a snapshot of Cytoscape's GUI, by selecting the "Import Network" menu option, Cytoscape's import-network plug-in kicks off the ImportNetworkTask module that sends a getGraph request to MASS CytoscapeListener to retrieve its in-memory graph. The plug-in then de-serializes the MASS response into a Cytoscape representation of the graph: a CyNetwork. To achieve a simple and consistent visualization of a graph, a few minor styles were applied to the Cytoscape nodes created (see Figure 5: vertices with neighbors in blue rectangles, those with no neighbors in red circles, and a user-clicked vertex in a yellow rectangle).

IV. Evaluation

We have implemented, verified the functionality of, and evaluated the execution performance of all the MASS library's graph features, including its interface to Cytoscape. The functional tests covered all file formats of input data in CSV, HIPPIE, MATSim XML, and KEYVALUE. Figure 5

shows a scenario of loading the hippie_current.txt dataset [47], (a biological network) into the MASS library's GraphPlaces, activating Cytoscape's import-network plug-in, and visualizing the graph on the display. Further functional tests have been conducted with our benchmark test set from [7].

For performance measurements, we developed two programs: (1) incremental construction of a complete graph and (2) KD tree construction used for range search of 2D data points. All the experiments were conducted over a cluster of 14 computing nodes made available by the University of Washington Bothell. Four are physical machines, each with 8-core 2.33GHz CPU (Intel Xeon E5410) with 16GB memory, whereas the other ten are virtual machines, each with 4-core 2.10GHz (Intel Xeon Gold 6130) with 16, 18, or 20GB memory.

A. Incremental Graph Construction

This performance comparison increases the size of a complete graph every 100 vertices. We consider two time-measurement options: (1) time elapsed to add 100 vertices and their associated edges to the graph and (2) time elapsed not only to add vertices/edges but also to retrieve the entire graph back to the main function. The former corresponds to batched computation, whereas the latter assumes interactive computation where users repeat checking their graph modification.

We coded this incremental graph construction in MASS, IncRDD [32], and Spark/GraphX, and compared their execution time. Listings 3, 4, and 5 summarize their code, respectively. While MASS and IncRDD can add vertices and edges to their original graph (lines 2-5 in Listing 3 and lines 2-4 in Listing 4 respectively), GraphX needs to construct a new RDD every time it increases the graph size (lines 1-7 in Listing 5). In these code snippets, blue lines are inserted for time-measurement option 2 that retrieves the constructed graph back to the main program. Contrary to MASS and IncRDD that retrieve an entire graph in one line, GraphX must retrieve vertices and edges into their respective RDD.

Listing 3. MASS incremental graph construction

```
1 GraphPlaces graph = new GraphPlaces(0, "TestGraph", 100);
2 for (int i = 0; i < count; i++) { // count = 100
3   graph.addVertex(start + i);
4   for (int n = 0; n < size + count; n++)
5     if (n != start + i) graph.addEdge(start + i, n);
6 }
7 GraphModel allInMain = graph.getGraph();
```

Listing 4. IncRDD incremental graph construction

```
1 IncRDD graphRDD = generateGraph(sc, model);
2 List<Object> neighbors
3   = new ArrayList<>(current_size + count - 1); // count = 100
4 graphRDD = verticesIncRDD.add((int)id, neighbors);
5 scala.Tuple2[] verticesEdges = (scala.Tuple2[])graphRDD.collect();
```

Listing 5. Spark/GraphX incremental graph construction

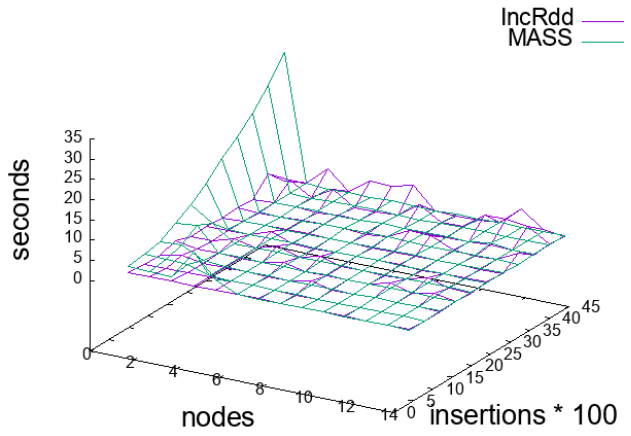
```
1 List<Edge<String>> edges = new ArrayList<>();
2 List<Tuple2<Object, String>> vertices = ...;
3 JavaRDD<Tuple2<Object, String>> verticesRDD
4                                   = sc.parallelize(vertices);
5 JavaRDD<Edge<String>> edgesRDD = sc.parallelize(edges);
6 Graph<String, String> graphRDD
7   = Graph.apply(verticesRDD.rdd(), edgesRDD.rdd(), ...)
8 vertices = ((scala.Tuple2[]) graph.vertices().collect());
9 edges = (org.apache.spark.graphx.Edge []) graph.edges().collect();
```

1) Performance of Incremental Graph Construction Only: Figures 6(a) and 6(b) compare MASS versus IncRDD and MASS versus GraphX performance, respectively, in incremental graph construction by repetitively adding 100 vertices up to 4,500. MASS halves its execution time when increasing the number of computing nodes from one to two, but beyond two nodes, its time reduction slows down. This is because, while an entire graph is distributed over the cluster system, MASS uses Hazelcast's distributed hash to have all computing nodes examine the message to check if they should add a new vertex or edge to their own sub-graph, (which is a typical approach as shown by Iwabuchi et. al. [52]). Conversely, both IncRDD and GraphX fluctuate their execution time mostly slower than MASS, due to the nature of Spark's data partition and shuffle operations. Figure 6(c) focuses on one-by-one vertex additions up to 4,500 vertices on top of each software tool when increasing the number of cluster nodes from 1 to 14. IncRDD outperforms MASS with one and two computing nodes but performs approximately 25% slower or more beyond two computing nodes. GraphX performs 50%+ slower than MASS with multiple computing nodes. Figure 6(c) also confirms Cuckoo Hashing allows IncRDD to perform faster than GraphX.

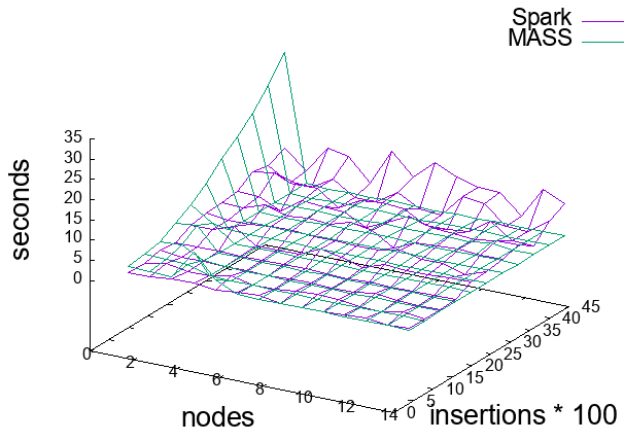
2) Performance of Incremental Graph Construction and Retrieval Back to the Main Program: Figures 7(a) and 7(b) compare MASS versus IncRDD and MASS versus GraphX performance, respectively, in both graph construction and retrieval by repetitively adding 100 vertices. Since MASS incurs overheads largely for de-serializing GraphPlaces vertex/edge information into a user-defined array at the main program, it cannot demonstrate better parallel performance. Similarly IncRDD does not show apparent parallelization effects. It increases its execution time even more than proportional to the graph size. In contrast, because GraphX performs *collect()* at each partition and its main program only gathers them up, it can improve its parallel performance. In fact, as shown in Figure 7(c), GraphX outperforms MASS beyond four cluster nodes when incrementally adding a new vertex to a graph and retrieving all its vertices/edges back to the main program.

B. Agent-Based KD Tree Construction

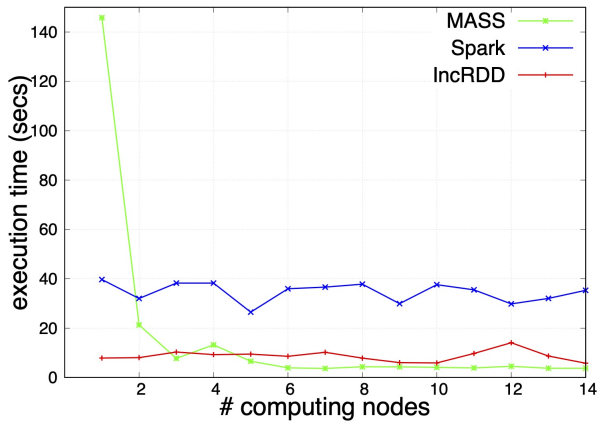
The range-searching problem constructs a a multidimensional binary tree (KD tree) and has each query traverse only the branches in its range. We focus on a 2D problem, so that each range query includes four values: x- minimum, maximum and y- minimum, maximum coordinates in a plane. KD tree for 2D points is a modified 2D binary search tree (BST), which alternates x- and y- coordinates as a key for inserting elements. The alternating sequence starts with the x-coordinate. The



(a) MASS versus IncRDD in repetitive 100-vertex additions

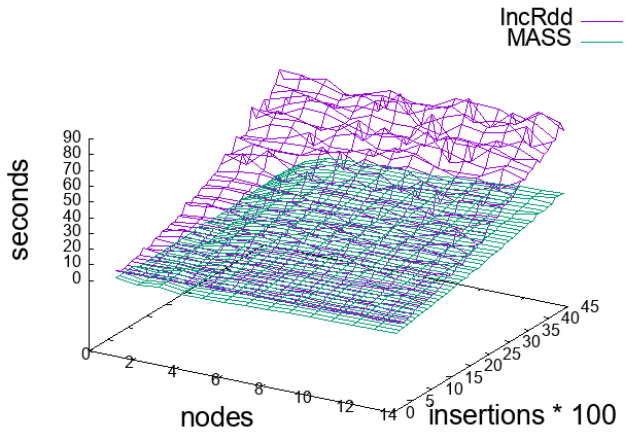


(b) MASS versus Spark/GraphX in repetitive 100-vertex additions

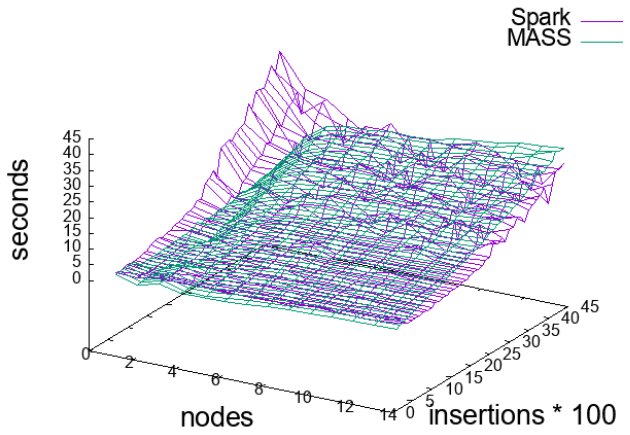


(c) A comparison of MASS, IncRDD, and Spark/GraphX in one-by-one vertex additions up to 4,500 vertices

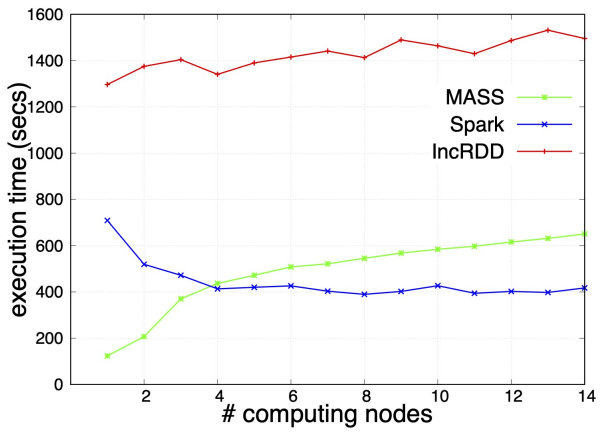
Fig. 6. Incremental graph construction only



(a) MASS versus IncRDD in repetitive 100-vertex additions



(b) MASS versus Spark/GraphX in repetitive 100-vertex additions



(c) A comparison of MASS, IncRDD, and Spark/GraphX in one-by-one vertex additions up to 4,500 vertices

Fig. 7. Incremental graph construction and retrieval

construction of KD tree consists of recursively partitioning the plane into two half-planes, where the point positioned at the bisector line is the next point to be inserted into the tree with respect to x and y dimensions. Each bisector line is determined after sorting the points by x or y coordinate depending on the next dimension of the KD tree level. The bisector line is determined by dividing the number of points by two. We chose to use a balanced BST for the KD tree due to sorting, bisector lines, and x and y dimension alternations.

Listings 6 and 7 show abstract code of MASS-based and MapReduce/Spark-based KD tree construction and one-time range search respectively. MASS mimics a tree with GraphPlaces (lines 6-11 in Listing 6) and thereafter disseminates agents along the tree branches in a given range (lines 21-25). On the other hand, MapReduce and Spark partition a collection of points into small slices (line 4 in Listing 7). For each slice, they build a KD tree (line 8) and perform range search on it (lines 9-11).

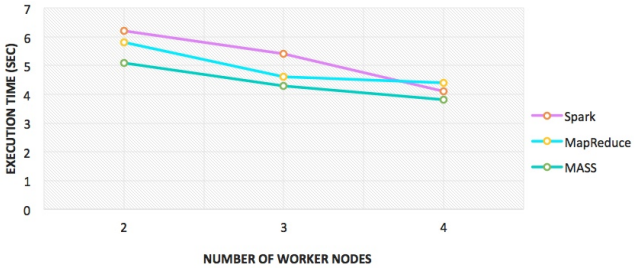
Listing 6. KD tree construction (MASS abstract code)

```
1 public class KDTreeMASS {
2   public void main(String[] args) {
3     Read input points (x, y) from an input file.
4     Create and sort a list of points by x coordinate.
5     MASS.init( );
6     GraphPlaces kdtree
7         = new GraphPlaces(1, "KDTree", #points);
8     for ( each : points ) {
9       kdtree.addVertex( each );
10      kdtree.addEdge( );
11    }
12    Agents searcher = new Agents(2, "Searcher", kdtree, 1);
13    crawler.doWhile(()->crawlers.hasAgents()); //one-time search
14    MASS.finish();
15 } }
16 public class Searcher extends Agent {
17   @onCreation public void init( destination ) {
18       migrate( destination ) // if NULL, it's the root.
19   }
20   @onArrival public void walk( ) {
21     if ( the current vertex is in the range ) {
22       add it to my_range_list;
23       if (left_subtree) spawn( left_subtree );
24       if (right_subtree) migrate( right_subtree );
25       else return my_range_list;
26 } } }
```

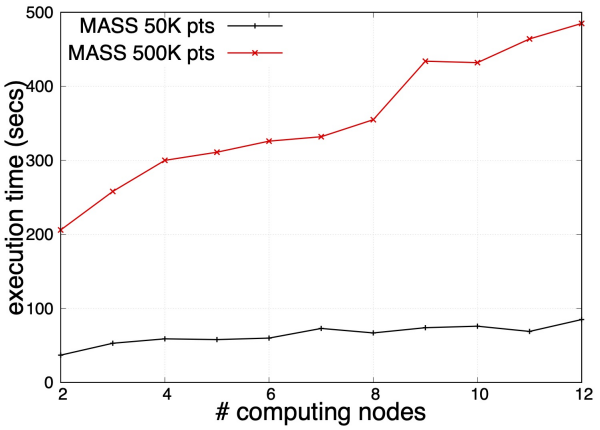
Listing 7. KD tree construction (MapReduce and Spark abstract code)

```
1 public class KDTreeSpark {
2   public void main(String[] args) {
3     Read input points (x, y) from an input file.
4     Partition points into small slices.
5     for (each : partitions) {
6       Sort points by x coordinate.
7       Remove duplicate points if any.
8       Build KD tree of points.
9       Perform range search on KD tree. // one-time search
10      Store the discovered points in a local list.
11      return the local list.
12    }
13    Collect all search result points from each partition.
14 } }
```

Figure 8(a) compares MASS, MapReduce, and Spark when constructing a KD tree with 10K data points. MASS outperforms MapReduce and Spark. This is because the MASS



(a) MASS versus MapReduce/Spark using 10K data points



(b) MASS performance using 50K and 500K data points

Fig. 8. KD tree construction

library's graph construction heavily relies on Hazelcast's distributed hash and multicast whose overheads are negligible with a small number of computing nodes. On the other hand, MapReduce and Spark still needs to repeat point sorting and duplicate removals within each partition. Figure 8(b) measures the MASS library's performance to handle 50K and 500K points over 2 to 12 computing nodes. The larger number of points, the more apparent MASS slows down in proportional to the number of computing nodes. This shows the necessity of switching our vertex/edge distribution strategy from the TCP-based to the UDP-based Hazelcast multicast or to our own multicast implementation.

V. Conclusion

To pursue our research focused on agent-based data discovery of structured datasets, we implemented a framework to construct, visualize, and further edit a graph on top of the MASS library. The performance comparison with GraphX, IncRDD, Spark, and MapReduce demonstrated the efficiency of MASS graph features to incrementally construct a graph in memory but pointed out the necessity of revising the current TCP-based vertex/edge distribution. As a practical graph application, we will re-implement an agent-based biological network motif search [45]. We are also extending the MASS library to cover trees, quad-trees, and 2D/3D contiguous spaces

for spatial and geometric data analysis. Finally, note that the MASS library and all its benchmark test programs are accessible through the MASS library's website [10].

REFERENCES

[1] MapReduce, "Accessed on: August 10, 2020. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/."

[2] Spark, "Accessed on: August 10, 2020. [Online]. Available: http://spark.apache.org/."

[3] Storm, "http://storm.apache.org/."

[4] J. Buck *et al.*, "SciHadoop: Array-based Query Processing in Hadoop," in *Proceedings of SC'2011*, 2011, doi:10.1145/2063384.2063473.

[5] Unidata — NetCDF, "Accessed on: August 10, 2020. [online]. available: https://www.unidata.ucar.edu/software/netcdf/."

[6] Spark GraphX, "Accessed on: August 10, 2020. [Online]. Available: https://spark.apache.org/graphx/."

[7] M. Fukuda, C. Gordon, U. Mert, and M. Sell, "Agent-Based Computational Framework for Distributed Analysis," *IEEE Computer*, vol. 53, no. 3, pp. 16–25, 2020.

[8] Y. Shih *et al.*, "Translation of String-and-Pin-based Shortest Path Search into Data-Scalable Agent-based Computational Models," in *Proceedings of Winter Simulation Conference*, Gothenburg, Sweden, December 2018, pp. 881–892.

[9] C. Gordon *et al.*, "Implementation techniques to parallelize agent-based graph analysis," in *Int'l Workshops of PAAMS 2019, Highlights of Practical Applications of Survivable Agents and Multi-Agent Systems*, Avila, Spain, June 2019, pp. 3–14.

[10] MASS: A Parallel Library for Multi-Agent Spatial Simulation, "Accessed on: August 10, 2020. [online]. available: http://depts.washington.edu/dslab/mass/."

[11] S. Gokulramkumar, "Agent Based Parallelization of Computationl Geometry Algorithms," Master's thesis, University of Washington Bothell, June 2020.

[12] C. Liu, "Development of Application Programs Oriented to Agent-Based Data Analysis," University of Washington Bothell, Tech. Rep., March 2020.

[13] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *International Journal of High Performance Computing Applications*, vol. Vol.20, no. No.2, pp. 203–231, 2006.

[14] Repast HPC, "Accessed on: August 10, 2020. [Online]. Available: https://repast.github.io/repast_hpc.html."

[15] HIPPIE, "Accessed on: August 10, 2020. [online]. available: http://cbdm-01.zdv.uni-mainz.de/∼mschaefer/hippie/index.php."

[16] MATSIM Multi-Agent Transport Simulation, "https://www.matsim.org/."

[17] Cytoscape: An Open Source Platform for Complex Network Analysis and Visualization, "Accessed on: August 10, 2020. [online]. available: http://cytoscape.org."

[18] N. Minar and other, "The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations," Santa Fe Institute, Tech. Rep. SFI WORKING PAPER: 1996-06-042, 1996.

[19] J. Kennedy *et al.*, "Particle swarm optimization," in *Proceedings of the IEEE International Conference on Neural Networks IV*, 1995, pp. 1942–1948.

[20] S. Saremi, S. Mirjalili, and A. Lewis, "Grasshopper optimization algorithm: Theory and application," *Advances in Engineering Software*, vol. 105, pp. 30–47, 2017.

[21] C. Blum, "Ant colony optimization: introduction and recent trends," *Physics of Life Reviews*, vol. 2, no. 4, pp. 353–373, 2005.

[22] A. W. McNabb, C. K. Monson, and K. D. Seppi, "Parallel PSO using MapReduce," in *Proc. of 2007 IEEE Congress on Evolutionary Computation*, Singapore, September 2007, pp. 7–14.

[23] A. Siemiński and M. Kopel, "Comparing efficiency of aco parallel implementations," *Journal of Intelligent and Fuzzy Systems*, vol. 32, no. 2, pp. 1377–1388, 2017.

[24] M. Jayakumar and R. Tyagi, "Particle swarm optimization using spark framework," University of Bonn, Tech. Rep., July 2019.

[25] Y. Karouani and Z. Elhoussaine, "Efficient spark-based framework for solving the traveling salesman problem using a distributed swarm intelligence method," in *2018 Int'l Conf on Intelligent Systems and Computer Vision (ISCV)*, Fez, Morocco, April 2018, pp. 1–6.

[26] FLAME, "Accessed on: August 10, 2020. [online]. available: http://www.flame.ac.uk."

[27] J. Lin *et al.*, *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[28] C. E. Tsourakakis, "Data Mining with MAPREDUCE: Graph and Tensor Algorithms with Applications," Accessed on: August 10, 2020. [Online]. Available: https://www.ml.cmu.edu/research/dap-papers/tsourakakisdap.pdf, 2010.

[29] Tez, "Accessed on: August 10, 2020. [Online]. Available: https://tez.apache.org."

[30] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proc. of SIGMOD'10*. Indianapolis, IN: ACM, June 2010, pp. 135–145.

[31] M. Parsian, *Data Algorithms: Recipes for Scaling Up with Hadoop and Spark*. O'Reilly, 2015.

[32] P. D. Prakash, "IncRDD: Incremental Updates for RDD in Apache Spark," Master's thesis, The University of Texas at Dallas, May 2017.

[33] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, p. 122–144, May 2004.

[34] M. E. Belviranli, S. Lee, and J. S. Vetter, "Designing Algorithms for the EMU Migrating-threads-based Architecture," in *Proceedings of the 22nd IEEE High Performance extreme Computing Conference (HPEC)*, Waltham, MA, Septmeber 2018, pp. 1–7, doi: 10.1109/H-PEC.2018.8 547 571.

[35] U. Borštnik, J. VandeVondeleb, V. Webera, and J. Huttera, "Sparse matrix multiplication: The distributed block-compressed sparse row library," *Parallel Computing*, vol. 40, no. 5-6, p. 47–58, 2014.

[36] Intel Cilk Plus, "Accessed on: August 10, 2020. [online]. available: https://www.cilkplus.org."

[37] S. Luke, C. Cioffi-revilla, L. Panait, and K. Sullivan, "MASON, A Multiagent Simulation Environment," *Simulation*, vol. 81, no. 7, pp. 517–527, 2005.

[38] Repast Simphony, "Accessed on: August 10, 2020. [online]. available: https://repast.github.io/repast_simphony.html."

[39] G. A. Pavlopoulos, D. Paez-Espino, N. C. Kyrpides, and I. Iliopoulos, "Empirical Comparison of Visualization Tools for Larger-Scale Network Analysis," *Advances in Bioinformatics*, vol. 2017, p. https://doi.org/10.1155/2017/1278932, July 2017.

[40] Data Visualization Software — Tulip, "Accessed on: August 10, 2020. [online]. available: https://tulip.labri.fr/tulipdrupal/."

[41] Gephi - The Open Graph Viz Platform, "Accessed on: August 10, 2020. [online]. available: http://gephi.org."

[42] Pajek/PajekXXL/Pajek3XL, "Accessed on: August 10, 2020. [online]. available: http://mrvar.fdv.uni-lj.si/pajek/."

[43] J. Leskovec, "Stanford Large Network Dataset Collection," Accessed on: August 10, 2020. [Online]. Available: https://snap.stanford.edu/biodata/datasets/10028/10028-PP-Miner.html.

[44] STRING: functional protein association networks, "Accessed on: August 10, 2020. [online]. available: https://string-db.org/."

[45] A. Andersen, W. Kim, and M. Fukuda, "Mass-based nemoprofile construction for an efficient network motif search," in *IEEE International Conference on Big Data and Cloud Computing in Bioinformatics - BDCloud 2016*, Atlanta, GA, October 2016, pp. 601–606.

[46] M. Sell and M. Fukuda, "Agent programmability enhancement for rambling over a scientific dataset," in *Int'l Conference, PAAMS 2020, Advances in Practical Applications of Agents, Multi-Agent Systems, and Thrustworthiness*, L'Aquila, Italy, October 2020, pp. 251–263.

[47] HIPPIE current TAB, "Accessed on: May 20, 2020. [online]. available: http://cbdm-01.zdv.uni-mainz.de/∼mschaefer/hippie/hippie_current.txt."

[48] Hazelcast, "Accessed on: May 14, 2020. [Online]. Available: https://hazelcast.com/."

[49] N. Alghamdi, "Supporting Interactive Computing Features for MASS Library: Rollback and Monitoring System," University of Washington Bothell, MS Capstone Final Report, 2020.

[50] Oracle, *Java Platform, Standard Edition Java Shell User's Guide. Oracle*, Accessed on: August 11, 2020. [Online]. Avaiable: https://docs.oracle.com/javase/10/jshell, 2018.

[51] OSGi Alliance, "Accessed on: August 10, 2020. [Online]. Available: osgi.org."

[52] K. Iwabuchi, S. Sallinen, R. Pearce, B. V. Essen, M. Gokhale, and S. Matsuoka, "Towards a distributed large-scale dynamic graph data store," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops - IPDPSW*, Chicago, IL, May 2017, pp. 892–901.