Agent-based Graph Applications in MASS Java and Comparison with Spark

Yutian Cui (Caroline Tsui)

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2022

University of Washington

Abstract

Agent-based Graph Applications in MASS Java and Comparison with Spark

Yutian Cui (Caroline Tsui)

Chair of the Supervisory Committee:

Prof. Munehiro Fukuda

Computing & Software Systems

Graph theory is constantly evolving as it is applied to mathematics, science, and technology, and it has active applications in communication networks, computer science, and geographic information system. Research on the implementation and optimization of graph theory is of great significance to various fields. However, due to the increasing size of databases today, the volume of datasets in industry has reached the level of petabytes (1,024 terabytes) and even exabytes (1,024 petabytes), which can generate graphs contains millions of vertices and edges. Analyzing and processing massive graphs has become a principal task in numerous fields. It is very challenging to process and compute rapidly growing, huge datasets in a reasonable amount of time with limited computing and memory resources. In order to meet the need for improved performance, some solutions, parallel computing frameworks, have emerged in the industry one after another. Nevertheless, it is unclear that how these parallel framework differ in term of performance and programmability for graph theory research or graph application development. The goal of this project is to compare the performance and programmability of two parallel libraries, MASS Java developed by the DSLab at the University of Washington Bothell, and Spark developed by the AMPLab at the University of California Berkeley, for graph programming. In order to balance performance and programmability, we used MASS Java and Spark Java to design and develop Graph Bridge, Minimum Spanning Tree, and Strongly Connected Components respectively, for a total of six graph applications. After more than three times testing each application and comparing their performance, the execution results show that: 1) For the Graph Bridge application, when running small datasets, the MASS Java version performs better when tested with one and two computing nodes. When using more than four computing nodes, the performance of the MASS Java version drops precipitously, and the performance of the Spark version is slightly better than MASS. When running large datasets, the execution performance of the MASS Java version is significantly improved with the increase of computing nodes, while the Spark version cannot complete all tests due to high memory overhead; 2) For the Minimum Spanning Tree application, the execution time of the MASS Java version and the Spark version increases with the number of computing nodes. The execution time of the MASS version increases significantly, while that of the Spark version increases slightly. 3) For the Strongly Connected Components application, the MASS Java version consistently performs better than the Spark version. After calculating Lines of Code, Boilerplate Code Ratio, and Cyclomatic Count per application, and measuring and evaluating the programmability, the conclusions show that: the programmability of the MASS Java version has generally better than the Spark version. Different from the Spark version, MASS Java supports communication between vertices, which greatly

increases the flexibility of algorithm implementation and reduces the complexity of code. In addition, from the perspective of development engineers or users, MASS Java is more intuitive in implementation and more suitable for developing graph applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACKNOWLEDGEMENTS

Words cannot express my gratitude to my professor and chair of my committee, Dr. Munehiro Fukuda, for his invaluable patience and feedback. I also could not have undertaken this journey without my defense committee members, Dr. Dimpsey and Dr. Asuncion, who generously provided knowledge and expertise.

I am also grateful to my team members in Distributed System Lab (DSLab), especially Yan Hong and Brian Luger, for their help, late-night feedback, and support.

Lastly, I would be remiss in not mentioning my family, especially my parents and Grayson Li. Their belief in me has kept my spirits and motivation high during this process. I would also like to thank my cat Dobby for all the entertainment and emotional support.

Chapter 1. INTRODUCTION

In mathematics, graph theory [1] provides an abstract model of entities and their relationships in the form of graphs. In computer science, graph is a data structure consisting of a set of vertices and a set of edges connecting pairs of vertices. Graph is usually used to describe a specific relationship between certain entities, with vertices representing the entity, and edges representing the relationship between the corresponding two entities.

Many real-world situations, such as social networks, biological networks, neural relationships, can conveniently be described by means of a graph consisting of a set of vertices together with edges joining certain pairs of these points. For example, the vertices could represent people, with edges joining pairs of friends; or vertices might be communication centers, with edges representing communication links. Graph computing is to model data (anything in the objective world and the relationship between entities) in the form of a graph, and calculate and analyze the dataset. Therefore, the research on graph theory and graph computing has strong practical significance.

This chapter 1) briefly introduces the two parallelization frameworks, MASS Java and Spark; 2) discusses the motivation for studying graph applications and choosing these two parallel programming libraries; and 3) presents the goals of this capstone project.

1.1 PARALLEL FRAMEWORKS

This section briefly introduces two parallelization frameworks, MASS Java and Spark.

1.1.1 *MASS*

Multi-Agent Spatial Simulation (hereinafter refers to as “MASS”) [2] is an agent-based parallel-computing library for multi-agent and spatial simulation over a cluster of computing nodes. MASS started as a research project at the of Distributed Systems Laboratory (hereinafter refers to as “DSLab”) [3] led by Professor Munehiro Fukuda of University of Washington Bothell in 2010, currently available in several languages, including Java [4], C++ [5], and CUDA [6]. Over the years, many of the ideas behind the system were presented in various journals and research papers over the years. MASS provides an intuitive programming framework for big data processing that can simulate many real-world problems, such as bioinformatics, social networks, geographic information system and climate analysis [7]. MASS follows the master-worker architecture, worker nodes spawn processes, and the processes communicate with each other through TCP connections. `Places` and `Agents` are two crucial components to the MASS library. MASS also supports other data structures, including arrays, binary trees, quadtrees, continuous spaces, which can better meet the needs of users in various situations.

1.1.2 *Spark*

Apache Spark [8] is a parallelization framework developed by Scala language, which provides a unified computing engine for big data analysis. Spark follows the master-worker architecture with two main processes, a driver and executors, and a cluster manager — the driver is responsible for distributing Spark applications into actual Spark jobs that run on worker nodes; the cluster manager is responsible for allocating and releasing resources for spark jobs. Spark provides a fault-tolerant abstraction termed as Resilient Distributed Datasets (hereinafter refers to as “RDD”) [9], which is the immutable collection of elements partitioned over the distributed computing nodes.

1.2 MOTIVATION

Graph is an abstraction of the relationship in nature, which can be applied to simulate many real-world problems, such as social networks, biological networks, and neural relationships, and then get the best solution through a series of algorithms and computations. Therefore, the study of graph theory has a strong practical significance, which is the motivation of this project to choose parallelization graph applications.

With the rapid development of the Internet and mobile Internet [10], the volume of datasets in real academia and industry is often huge, reaching the level of petabytes (1,024 terabytes) or level of exabytes (1,024 petabytes) of data, including billions of or even trillions of records from millions of people, and these data can generate graphs contains millions of vertices and edges. Analyzing and processing massive graphs has become a principal task in different fields. When existing sequential tools encounter huge datasets, high-performance computations cannot be completed with a single computing node [11]. It is very challenging to process and compute rapidly growing, huge datasets in a reasonable amount of time with limited computing and memory resources. In order to meet the need of improving performance, parallelization and the use of distributed memory are common techniques for solving huge graph problems. So far, there have been some parallel frameworks that can be used for distributed computing, such as MASS, Spark, Pregel, Repast Simphony, GraphLab, JGraphT and ringo.

Based on the purpose of discussing and comparing the performance and programmability of different parallelization frameworks, we have chosen two distinct parallelization structures. And most important, Spark is currently used and dominates in the industry. So Spark is a good baseline for comparing MASS.

For the purpose of discussing and comparing the performance and programmability of different parallelization frameworks, we have chosen two distinct parallelization structures, MASS Java and Spark Java. Spark is not an Agent-based framework, and Spark is currently used and dominant in the industry. Therefore, Spark is a good baseline for comparing MASS. Spark provides in-memory computing by iterating `RDD` [9] and reducing data movement, while the Agent-based framework MASS provides direct communication between vertices by `Agents` migrating. The obvious feature difference between these two parallel frameworks prompted us to use the MASS and Spark frameworks to design and implement three graph applications in this capstone project, to find out the advantages and disadvantages of the Agent-based MASS and data processing Spark in graph programming, and evaluate the performance and programmability of application.

1.3 PROJECT GOALS

Due to the motivations mentioned above, the project goals for this capstone work are as follows:

- Design and implement Graph Bridge application using MASS Java and Spark, and evaluate the performance and programmability between MASS Java and Spark.
- Design and implement Minimum Spanning Tree application using MASS Java and Spark, and evaluate the performance and programmability between MASS Java and Spark .
- Design and implement Strongly Connected Components application using MASS Java and Spark, and evaluate the performance and programmability between MASS Java and Spark .

The rest of this white paper is organized as follows: Chapter 2 discusses the background of parallel libraries MASS, Spark and sequential approaches of graph applications. Chapter 3 reviews related works of parallel graphic programs. Chapter 4 describes the design and implementation details of parallelization using MASS Java and Spark. Chapter 5 evaluates the performance and programmability of the graph applications. Finally, Chapter 6 summarizes the limitations and future improvements of this white paper.

Chapter 2. BACKGROUND

This chapter aims to provide an in-depth overview of the MASS Java library and Spark from a graph programming perspective. And introduces the graph classification, the research implications of three graph applications, and the solution of the sequence tool.

2.1 PARALLEL LIBRARIES

2.1.1 *Graph in MASS Java*

In the previous chapter 1.1.1, we briefly introduced the MASS Java framework, and we learned that the MASS library has two crucial components: `Places` and `Agents`.

`Places` is a distributed data structure (e.g. graph) of `Place` objects (e.g. vertex) that are dynamically allocated over a cluster of computing nodes. `Places` are managed by a set of global indexes, and each `Place` element can be identified by an index. Data can be saved in a specific `Place` and exchanged in different `Place`. Once `Place` is initialized and generated, its position is fixed. It will exist on the computing node at the time of distribution until the end of the program execution.

`Agents` are a group of execution instances that can be migrated. `Agents` have several common APIs: 1) They can `migrate()` to any place: each `Agent` object can specify the index of the next Place to indicate where to migrate next; 2) `spawn()` another `Agent`; and 3) `kill()` itself. `Agents` can interact with the `Place` where they reside and manipulate data stored there. On each compute node, multiple threads check in to each `Agent` one by one and process each `Agent`’s requests. `Agents` can move between `Places`. `Agents` can carry their own data when migrating to different Places, or they can transfer the data to the Place, so that other Agents can access the data when they arrive at the Place.

DSLab has successfully added distributed graph data structure to the MASS Java library [14]. It was originally designed and implemented by former lab member Justin Edward Gilroy in 2020 [15] and refactored by current lab member Brain Luger in 2021 [13, 16]. These preliminary works have greatly improved the efficiency of our development of graph applications.

The distributed graph data structure has two crucial components: GraphPlaces and GraphAgent. As shown in the Table 2.1 below, GraphPlaces extends from Places and consists of VertexPlace, which extends from Place. GraphAgent extends from the Agent.

Table 2.1 Relationships (containment, inheritance) between MASS Java classes

MASS Basic Library		MASS Graphic Library
Places	⇦	GraphPlaces
Contains ⇩	*Extends*	*Contains* ⇩
Place	⇦	VertexPlace
Agents		
Contains ⇩	*Extends*	
Agent	⇦	GraphAgent

2.1.1.1 GraphPlaces and VertexPlace

In the MASS Java library, the distributed graph data structure is called GraphPlaces. GraphPlaces is an extension of the existing Places with the ability to support simulations. Each GraphPlaces is composed of VertexPlaces.

As shown in Figure 2.1 below, the vertices in the graph are nearly evenly distributed on a cluster of computing nodes. When we add vertices to the graph, we follow a loop so that the vertices are balanced throughout the cluster. Each vertex is represented by a `VertexPlace` object. A vertex contains a list of outgoing edges and data about the vertex itself.

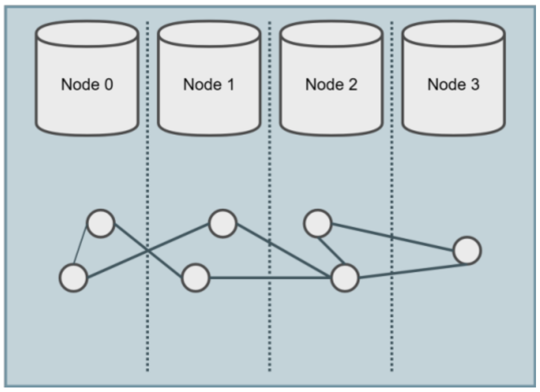


Figure 2.1 Visual representation of a distributed graph [13]

As shown in Figure 2.2 below, on each compute node, vertices are stored in a single vector. A single vector can grow dynamically with the size of the graph. After the graph is constructed, agents can migrate from one vertex to another through edges.

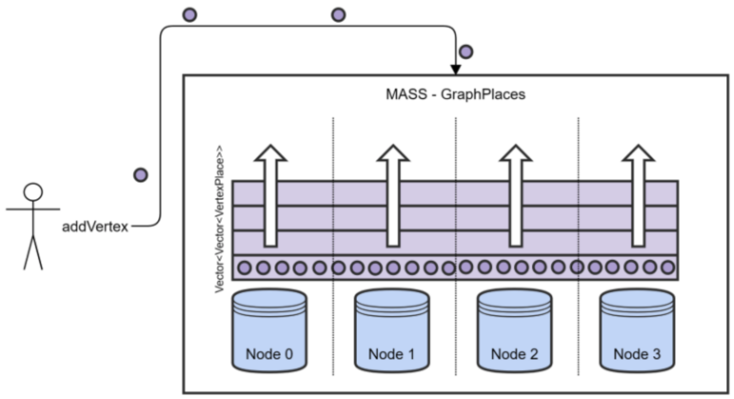


Figure 2.2 Vertex container and distribution [13]

2.1.1.2 `GraphAgent`

The `GraphAgent` class extends from `Agent` and overrides the `map` function with a function that maps the initial population of agents onto the graph. This feature works well in both single-computing node and multi-computing node deployments, ensuring an even distribution of agents in both cases. Therefore, `GraphAgent` also implements the agent's proper work on the graph data structure, such as `spawn()`, `migration()`, `kill()`.

2.1.2 *Spark, GraphX and Pregel*

In the previous chapter 1.1.2, we briefly introduced the Spark framework. We learned that Spark is a data processing parallel analysis and computing engine.

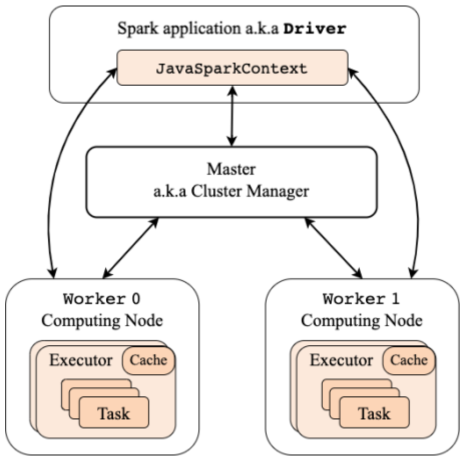


Figure 2.3 Spark Architecture [17]

As shown in Figure 2.3 above, Spark follows the master-worker architecture with two main processes: 1) a driver and executors, and 2) a cluster manager. The driver is responsible for distributing Spark applications into actual Spark jobs that run on worker computing nodes. The cluster manager is responsible for allocating and releasing resources for spark jobs. From the perspective of computing, the code lines in the operators are executed on the *executor* side, and other code lines are executed on the *driver* side.

Spark provides a fault-tolerant abstraction termed as RDD [8], which is the immutable collection of elements partitioned over the distributed computing nodes and the smallest computing unit of Spark. Each RDD stores running data and encapsulated immutable computing logic. At runtime, each RDD divides into several partitions and processes on different computing nodes.

Spark provides three operators — transformation operator, action operator, and control operator, which can be executed in parallel on RDD partitions. Transformation operators, such as `map()`, and `filter()`, construct new RDDs by transforming parent RDDs. Action operators, such as `collect()` and `count()`, calculate the RDD and return the result to the driver. Spark evaluates transformations lazily by delaying execution until the action operator is requested. Transformations are further classified into narrow and wide based on the dependencies involved in RDD creation. Objects residing in a single partition only depend on objects residing in a single partition of the parent RDD, with narrow dependencies, such as `map()`, `filter()`. Instead, objects depend on objects residing in multiple partitions of the parent RDD, with wide-ranging dependencies, such as `reduceByKey()`. Thus, extensive dependencies require expensive shuffle operators involving reallocation of data among worker nodes.

Spark GraphX [18] extends the concept of RDD to support several new graph RDD, such as `GraphRDD`, `VertexRDD`, `EdgeRDD`, and provides many popular graph-related operators, such as `mapVertices()`, `mapEdges()`, `joinVertices()` and `groupEdges()`. Since vertices should be handled in the context of their neighbors, GraphX also introduces the concept of `triples` and edges that can join vertex structures. In addition, GraphX implements the Pregel API [19], which uses its `sendMsg()` and `mergeMsg()` functions to automatically pass duplicate messages from one vertex to its neighbors.

Because it is completely different from Agent-based Model MASS and Spark is a popular parallelization framework today, we chose Spark as the comparison framework of this capstone project in order to get a more objective understanding the position of MASS's performance and programmability in the domain of parallelization framework.

2.2 Graph Applications

Graphs in this context consist of vertices connected by edges and can be classified as directed graph, undirected graph; weighted graph, and unweighted graph.

- Directed graph is a graph consisting of a set of vertices connected by directed edges.
- Undirected graph is a connected set of vertices where all edges are bidirectional.
- Weighted graph is a graph in which each branch is assigned a numerical weight. Thus, a weighted graph is a special type of labeled graph where the labels are numbers (often considered positive).
- Unweighted graph is a graph with no edge weights.

Among them, 1) Graph Bridge is suitable for undirected graphs; 2) Minimum Spanning Tree is suitable for undirected weighted graphs; and 3) Strongly Connected Components is suitable for directed graphs.

2.2.1 *Graph Bridge*

In graph theory, a bridge is a special edge of a graph whose removal increases the number of connected components of the graph. This graph application aims to find all bridges in an undirected graph. Graph bridges are used in social network analysis, and bridge recognition can help identify the strength or weakness of a relationship between two users or between two organizations.

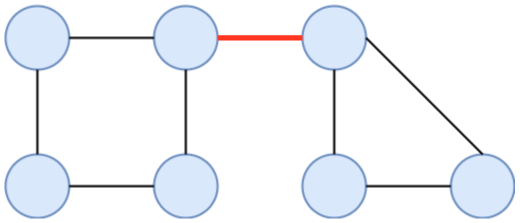


Figure 2.4 Graph with 7 vertices and 1 bridge (highlighted in red)

In 1974, Robert Tarjan described the first algorithm for finding bridges in a graph. Tarjan's graph bridge algorithm [20] performs the following steps:

Listing 2.1 Code snippet of Tarjan’s Graph Bridge (Sequential)

```
public class GraphBridge {
    int[] disc;
    int[] low;
    int[] parent;
    LinkedList<Integer>[] adjLists;
    int timer = 0;
    Set<Pair<Integer, Integer>> bridges = new HashSet<>();

    void findBridge(LinkedList<Integer>[] adjLists) {
          for (int i = 0; i < adjLists.length; i++) {
              if(disc[i] == -1) {
                  dfs(adjLists, i);
              }
          }
      }

      void dfs(LinkedList<Integer>[] adjLists, int root) {
          // base case
          disc[root] = timer;
          low[root] = timer;
          timer++;

          // recursive rule
          for (Integer nei : adjLists[root]) {
              if (disc[nei] == -1) {
                  parent[nei] = root;
                  dfs(adjLists, nei);
                  low[root] = Math.min(low[root], low[nei]);
                  if (disc[root] < low[nei])
                      bridges.add(new Pair<>(root, nei));
              } else if (nei != parent[root]) {
                  low[root] = Math.min(low[root], disc[nei]);
              }
34.           }
35.       }
36.       ...
37.  }
```

Tarjan's algorithm is a common method among existing sequential tools. Listing 2.1 shows the depth-first search traversal from the root vertex to each vertex, and updates the `discovery` value, `low` value and `parent` value (hereinafter referred to as three values) of each vertex, and finds all bridges in the graph by judging of conditional statement line 29 in the process of backtracking. Now we assume a graph has `v` vertices and `e` edges. First, we start from the root vertex and run depth-first search traversal on each of its neighbors. When passing through each vertex, update the three values on the vertex in turn. When the last vertex is reached, the backtracking stage starts, which updates the `low` value on the entire path and judges each edge. If the `low` value of the current vertex is found to be greater than `parent` vertex, then the edge connecting the parent vertex is a new bridge of the graph.

2.2.2 *Minimum Spanning Tree*

In graph theory, a tree is a way of connecting all vertices together. There is one and only one path from any vertex to any other vertex of the tree, so a graph can generate more than one tree. The edges of the original graph of the minimum spanning tree are weighted. The minimum spanning tree is a spanning tree with the smallest total weight in an undirected weighted graph. Depending on graph, there may be more than one minimum spanning tree – more than one tree is a minimum spanning tree when the sum of all edge weights is the same. There is one and only one minimum spanning tree if the sum of all edge weights is different. This application is intended for use in network design areas such as roads, telephone, electrical and cable-laying.

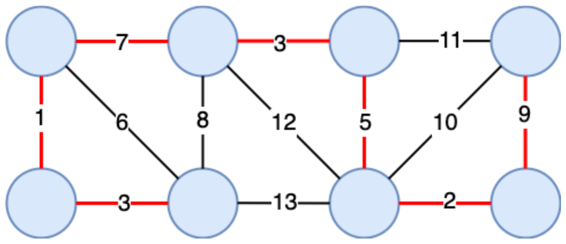


Figure 2.5 Minimum spanning tree of a graph (highlighted in red)

Czech scientist Otakar Borůvka developed the first known algorithm for finding a minimum spanning tree [21] in 1926, he wanted to solve the problem of finding an efficient coverage of Moravia with electricity. The other two sequential algorithms commonly used now are Prim algorithm and Kruskal algorithm. All three are greedy algorithms. Kruskal's algorithm performs the following steps:

Listing 2.2 Code snippet of Kruskal’s Minimum Spanning Tree (Sequential)

```
public class MinimumSpanningTree {
    int lines;
    int[][] graph;
    boolean[] connected;
    List<Edge> result = new ArrayList<>();

    private static long mst(List<String> result) {
        int size = graph.length;
        long sum = 0;
         long visitedNode = 1;
        PriorityQueue<Edge> minHeap = new PriorityQueue<>();
        connected[0] = true;

        for (int i = 1; i < size; i++) {
            if (graph[0][i] != 0)
                minHeap.offer(new Edge(0, i, graph[0][i]));
        }
        while (visitedNode != size && !minHeap.isEmpty()) {
            Edge edge = minHeap.poll();
            if (!connected[edge.to]) {
                for (int i = 0; i < size; i++) {
                    if (!connected[i] && graph[edge.to][i] != 0)
                        minHeap.offer(new Edge(edge.to, i,
graph[edge.to][i]));
24.                }
25.                connected[edge.to] = true;
26.                result.add(edge);
27.                 sum += edge.cost;
28.                 visitedNode++;
29.             }
30.         }
31.         if (visitedNode != size) return -1;
32.         return sum;
33.     }
34.
35.     static class Edge implements Comparable<Edge> {
36.         int from, to, cost;
37.     }
38.     ...
39. }
```

Kruskal's algorithm is a common method among existing sequential tools. Listing 2.2 shows a graph, by sorting all its edges and sequentially taking the edge with the minimum cost to judge the conditional statement line 22, if a cycle is not generated, then add it to the current minimum spanning tree set. Now we assume a graph has `v` vertices and `e` edges. First, we sort the `e` edges from small to large. Among the `e` edges, we pop out each minimum edge in turn. If the two vertices of the edge are found to be located on two trees, then merge the two trees into one tree; if the two vertices of the tree are located in the same tree, then ignore this edge and continue running. After all the edges have been visited, if all vertices can be found on this spanning tree, then this is the minimum spanning tree we are looking for, otherwise, there is no minimum spanning tree.

2.2.3 *Strongly Connected Components*

In the mathematical theory of directed graphs, a graph is said to be strongly connected if every vertex is reachable from every other vertex. Strongly connected components of an arbitrarily directed graph form a partition into subgraphs that are themselves strongly connected.

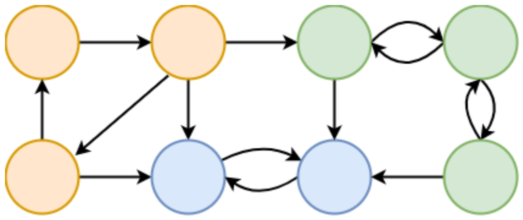


Figure 2.6 Graph with strongly connected components (highlighted in different colors)

Tarjan's depth-first search mentioned earlier also extends Tarjan’s strongly connected component algorithm [22]. Tarjan’s strongly connected components algorithm performs the following steps:

Listing 2.3 Code snippet of Tarjan’s SCC (Sequential)

```
public class StronglyConnectedComponents {
    int disc;
    int[] discoveries;
    int[] low;
    Deque<Integer> stack;
    boolean[] onStack;

    private static void SCC() {
        ...add
         for (int i = 0; i < graph.size(); i++) {
             if (discoveries[i] == -1) {
                 dfs(i);
             }
         }
     }

     private static void dfs(int cur) {
         // base case
         stack.offerFirst(cur);
         onStack[cur] = true;
         discoveries[cur] = low[cur] = disc;
         disc++;

         // recursive rule
         for (int nei : graph.get(cur)) {
             if (discoveries[nei] == -1) {
                 dfs(nei);
             }
             if (onStack[nei]) {
                 low[cur] = Math.min(low[cur], low[nei]);
```

```
            }
        }

        if (discoveries[cur] == low[cur]) {
            List<Integer> curRes = new ArrayList<>();
            for (int node = stack.peekFirst(); ; node = stack.peekFirst()) {
                curRes.add(stack.pollFirst());
                onStack[node] = false;
                if (node == cur) {
                    break;
                }
            }
            sccResult.add(curRes);
            countSCC++;
        }
    }
}
```

Tarjan’s algorithm is a common method among existing sequential tools. Listing 2.3 shows running depth-first search traverse from the root vertex to each vertex, updating the `low` value of each vertex, and finding all the components based on the same index value. Now we assume a graph has `v` vertices and `e` edges. The `index` value is the counter of components. S is the vertices stack, which starts empty and stores a history of vertices that have been explored but not yet added to a strongly connected component. The current stack of nodes pops up only when the complete strongly connected component is found. The outermost loop searches for each node that has not yet been visited, ensuring that nodes that cannot be reached from the first node are still eventually visited. The function `strongconnect` performs a single depth-first search on the graph, finds all successors to node `v`, and reports all strongly connected components of the subgraph. When each node completes recursion, if its `low` is still set to its `index`, then it is the root vertex of the strongly connected component, consisting of all the vertices above it on the stack. The algorithm pops the stack up and includes the current vertex, rendering all of these vertices as a strongly connected component.

Chapter 3. RELATED WORKS

This chapter aims to briefly introduce the graph applications and techniques that exist today for graph programming, and differentiate those works from our capstone project.

3.1 GRAPHLAB

GraphLab [23] is a graph-based high-performance distributed computing framework written in C++, started in 2009 by Professor Carlos Guestrin of Carnegie Mellon University, and is now an open-source project under the Apache License.

GraphLab is a vertex-centric system, unlike MASS, which is also known as "think-as-a-vertex". Each vertex computes its new state based on its current state and the messages received from its neighbors. Each vertex then sends its new state to its neighbors using message passing. The synchronous version follows the Batch Synchronous Parallel (BSP) model, which performs parallel computations in iterative steps and synchronizes between machines at the end of each step. This means that recipients in the same iteration do not have access to messages sent in one iteration; receiver vertices receive their messages in subsequent iterations. Computation stops when all vertices converge to a fixed point or after a predefined number of iterations. GraphLab has three functions in analyzing graphs: Gather, Apply and Scatter (GAS). The GAS model allows each vertex to collect data from its neighbors, apply computational functions to itself, and then disperse relevant information to some neighbors if necessary. GraphLab uses vertex-cut (i.e. edge-disjoint) partitions, not edge-cuts. This duplicates vertices and helps to better distribute the work of very high degree vertices. These vertices exist in social networks and network graphs because they follow a power law distribution. The replication factor of a vertex is the number of machines that replicate that vertex. GraphLab automatically uses all available cores and memory in the machine.

Existing graph benchmarks using GraphLab include Breadth-First Search (BFS), K-means, Logistic regression, Page Rank, Connected Components, Triangle Counting, Collaborative Filtering, and Belief Propagation [24, 25]. However, the three graph applications in this project are not included.

3.2 GraphMat

GraphMat [26] is a single-node multi-core graph framework written in C++. GraphMat takes vertex programs and maps them to generalized sparse matrix-vector multiplication operations. We get the productivity benefits of vertex programming while enjoying the high performance of an easy-to-understand and reason about matrix backend, while allowing users with knowledge of vertex programming a smooth transition to a high-performance environment. GraphMat is a multi-core optimized vertex programming model that achieves within 1.2x of native, hand-coded, optimized code on a variety of graph algorithms.

Existing graph benchmarks using GraphMat include Page Rank, Breadth-First Search (BFS), Triangle Counting, Collaborative Filtering, and Single Source Shortest Path (SSSP) [27]. However, the three graph applications in this project are not included.

3.3 Summary

According to the survey, neither Graph Bridge, Minimum Spanning Tree, nor Strongly Connected Components have precedents for benchmark graph tools, which is of great research significance considering their usage scenarios in many real-life situations. Therefore, these three applications are suitable as graph applications to implement and compare the performance and programmability of two graph libraries, MASS Java and Spark.

Chapter 4. PARALLELIZATION GRAPH APPLICATIONS

This chapter describes the design and implementation details of three graph applications using MASS Java and Spark respectively. It also includes simple demonstration to explain how the application works.

It is worth noting that in order to make the application have better performance, we chose to replace the built-in standard Java collections with `Fastutil` Collection [28]. Built-in Java collections such as `HashMap`, `HashSet` can consume large amounts of memory as the collection size increases and are tightly coupled to the internal data structures used. Optimizing performance to increase running speed benefits from the use of different internal data structures, such as `IntList`, `IntOpenHashSet`, and the use of Fastutil collections.

4.1 GRAPH BRIDGE

4.1.1 *MASS Java Design and Implementation*

When finding a bridge in a graph, three concepts need to be considered: 1) Discovery value means the time when first visit this vertex; 2) Parent value means which vertex does it come from; and 3) Low value means the vertices will update the low values during the backtracking session, so that the vertices are in the same component if they have the same low values. When the low value of vertex a is larger than the discovery value of the vertex b, we have found a bridge.

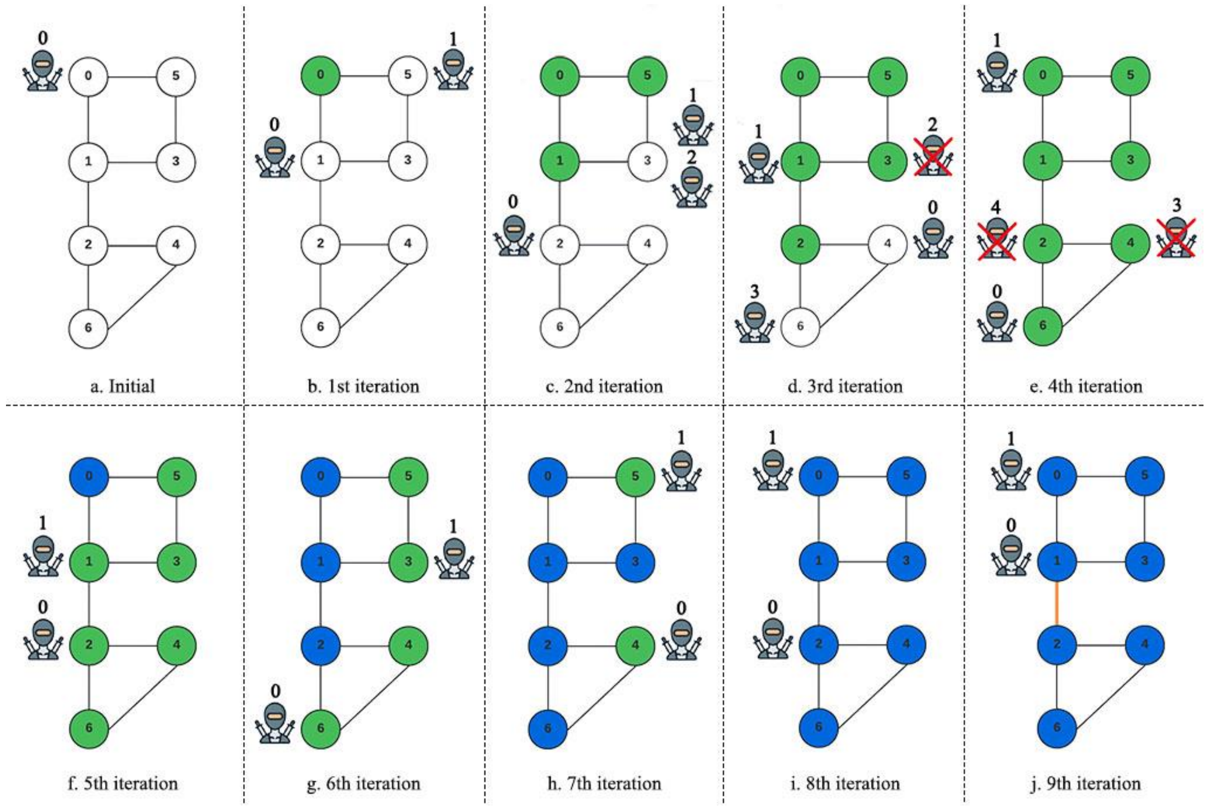


Figure 4.1 Demonstration of Graph Bridge in parallel

As seen in Figure 4.1 (a), initially, `Places` are generated from the dataset file, and generated one agent on `Place` 0. At every iteration of `Agent` function execution, the agent examines the neighborhood and performs the following actions. If there is a valid neighbor, the `Agent` migrates to the neighbor with the smallest id, and then `spawn()` child agent with parent agent information in the remaining neighbors; otherwise, `kill()` this agent. At the same time, the agents update their own information (discovery value, parent value and low value). If the `Agent` visits a certain vertex for the second time, it will enter a cycle and backtracking phases. The `Agent` will return along the original path, update the low value accordingly, and mark all the `Places` of the cycle as being within the cycle. If the agent visits the vertex for the third time, it means that the cycle phase ends, but the backtracking phase continues and starts looking for bridges. If a bridge is found, the information of the discovered bridge will be stored on the `Place` with a smaller endpoint id of the bridge, and the agent will continue backtracking phase until the agent enters the range of a cycle or returns to the root node, then the `Agent` will be `kill()`. After repeating this procedure until all agents are killed, all graph bridges will exist on places with smaller id. Finally, the `Main()` driver collects graph bridges from all places in parallel.

The following Listing 4.1 shows the main function of the graph bridge application, and its main computation is in the function `gb()`.

Listing 4.1 Code snippet of driver `GB` in Graph Bridge

```
public class GB {
    public static void main(String[] args)    {
        GraphPlaces graphPlaces = init(graphPath, printPlaces);
        gb(graphPlaces);
        ObjectSet<Bridge> results = processResults(graphPlaces);
        stopMASS();
    }
}
```

As shown in Listing 4.2, the the `gb()` function covers the main functions of calling MASS Java, line 9 and 12 `callAll()` and line 10 and 13 `manageAll()`. By passing parameters to the line 9 `callAll()` function, specific functions in `MyAgent` are called to make `Agents` complete the specified steps. The first step is to complete the computation of the `Agent` arriving at a `Place`, and the second step is to complete the migration of the `Agents`. line 10 and 13 `manageAll()` is a function that allows all `Agents` to start processing, `spawn()`,`migrate()` and `kill()`, at the same time. The following two application using MASS Java will use these APIs as well, so we will not repeat this later.

Listing 4.2 Code snippet of `gb()` in Graph Bridge driver

```
1. private void gb(GraphPlaces graphPlaces) {
2.     Agents crawlers = new Agents(0, MyAgent.class.getName(), null,
   graphPlaces, 1);
```

```
    crawlers.callAll(MyAgent.init_);

    int curAliveAgents = crawlers.nAgents();
    int iteration = 0;

    while (curAliveAgents > 0) {
        crawlers.callAll(MyAgent.onArrival_);
        crawlers.manageAll();

        crawlers.callAll(MyAgent.migrateTo_);
        crawlers.manageAll();

        iteration++;
    }
}
```

4.1.2 *Spark Design and Implementation*

Spark implementation to parallelize Graph Bridge application uses basic Spark abstraction, Resilient Distributed Datasets (RDD). Graph vertices maps to a `JavaPairRDD` and newly generated graph vertices map to new `JavaPairRDD` in each iteration, due to the immutable nature of Spark RDD. We did not choose to use Spark GraphX mainly because `EdgeRDD` and `VertexRDD` in GraphX are extended from org.apache.spark.rdd.RDD. But our application needs to continuously update the information of vertices and iterate according to the updated information, using Spark `JavaPairRDD` undoubtedly more flexible than GraphX, it can store the vertex id as a value for processing, instead of accessing and disassembling each vertex when using VertexRDD. The Spark version was revised based on previous lab members Maxwell Seto's version [29].

In Spark implementation as shown in Listing 4.3, finding bridge happens through a series of narrow-dependency transformations operators (such as `flatMapToPair()`), followed by an expensive wide-dependency transformation operators (such as `reduceByKey()`), that shuffles data across all the computing nodes. Finally, the driver gathers the bridges on each vertex via `collect()` action operator.

Initially, vertices are generated in the graph from the dataset file. At every iteration of function execution, the active vertex examines the neighborhood and performs the following actions. If there is a valid neighbor, set the current vertex as “VISITED” and other neighbor vertices are set to “ACTIVE”, and the new discovery value and low value are passed to other neighbor vertices; otherwise, set the state of this vertex to “DEAD”. If there is currently a vertex that meets the backtracking condition, it is judged whether the edge is a bridge. At each step, we use `reduceByKey()` to update the information of each vertex. After repeating this procedure until all vertices are non-active, all graph bridges will exist on the vertices. Finally, the driver `collects()` graph bridges from all vertices in parallel.

Listing 4.3 Code snippet of Graph Bridge using Spark

```
public class GB implements Serializable {
    private static Object2ObjectMap<Integer, IntSet> gb(JavaPairRDD<Integer, Vertex> graphRDD) {
        final Broadcast<IntList[]> broadcastAdjList = jsc.broadcast(adjList);
        CollectionAccumulator<String> accumulatorBridges = jsc.sc().collectionAccumulator("accumulatorBridges");

        int iteration = 0;

        // Depth-first seach loop.
        while (graphRDD.count() > 0) {
            // Step 1 of depth-first seach: Visited a vertex.
            graphRDD = visited(graphRDD);

            // Step 2 of depth-first seach: Expanded a vertex.
            graphRDD = expanded(graphRDD, broadcastAdjList, accumulatorBridges);

            iteration++;
        }
        return processComponentResults(accumulatorBridges);
        }
}
```

4.2 MINIMUM SPANNING TREE

4.2.1 *MASS Java Design and Implementation*

When finding a minimum spanning tree in a graph, one concepts need to be considered: Cutset, which is a set of edges with exactly one endpoint in the subgraph. When all vertices are on a tree without cycle and the total weights number is the smallest, we have found the minimum spanning tree.

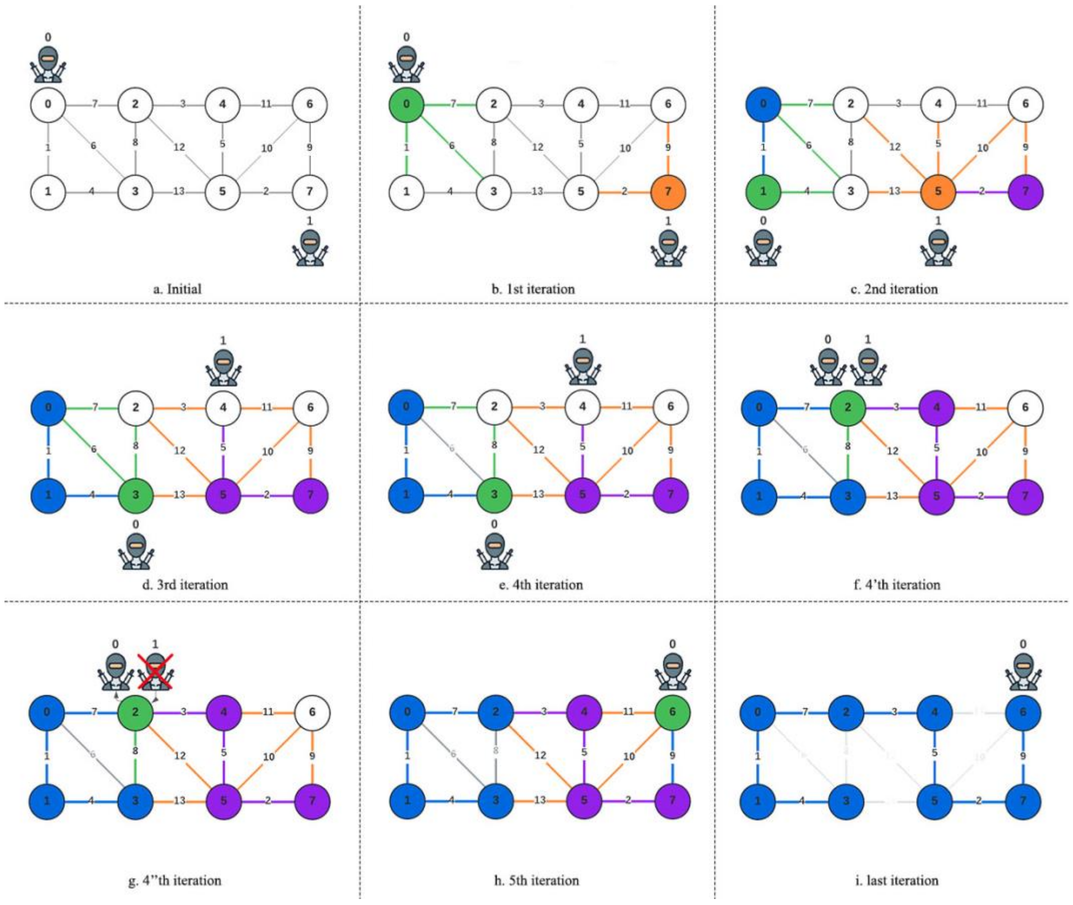


Figure 4.2 Demonstration of Minimum Spanning Tree in parallel

As seen in Figure 4.2 (a), initially, `Places` are generated from the dataset file, and generated agents based on the input parameter and we have two agents in this case. At every iteration of agent function execution, the agent performs the following actions. Select the edge with the smallest cost from the cutset at the current `Place`, and migrate to the other end of the edge. If two agents visit the same `Place`, pass the information to the `Agent` with the smaller id and `kill()` the other one. If the agent finds that the current smallest edge will form a cycle in the the generated minimum spanning tree, this edge will be discarded and continue to explore the second smallest edge. When the size of minimum spanning tree is equal to the size of the graph, `kill()` the last `Agent`. Finally, the driver collects the final minimum spanning tree from all `Places` in parallel.

The following Listing 4.4 shows the main function of the minimum spanning tree application, and its main computation is in the function `mst()`.

Listing 4.4 Code snippet of driver `MST` in Minimum Spanning Tree

```
public class MST {
    public static void main( String[] args ){
        // Generate Places.
        GraphPlaces graphPlaces = init(graphPath, printPlaces);
        mst(graphPlaces);
        stopMASS();
    }
}
```

The following Listing 4.5 shows the `mst()` function of the minimum spanning tree application, which also covers calling the main functions of MASS Java.

Listing 4.5 Code snippet of `mst()` in Minimum Spanning Tree driver

```
private void mst(GraphPlaces graphPlaces) {
    Agents crawlers = new Agents(agentId++, MyAgent.class.getName(),
null, graphPlaces, graphPlaces.size());
    crawlers.callAll(MyAgent.init_);
    crawlers.manageAll();

    int curNumOfEdges = 0;
    int targetNumOfEdges = graphPlaces.size() - 1;
    int iteration = 0;

    // Finding loop until all edges have been found.
    while (curNumOfEdges < targetNumOfEdges) {
        curNumOfEdges = 0;

        // Step 1. on arrival
        crawlers.callAll(MyAgent.onArrival_);
        crawlers.manageAll();

        // Step 2. migrate
        crawlers.callAll(MyAgent.migrateTo_);
        crawlers.manageAll();

        // Step 3. Get the number of current edges
        processMSTResult(graphPlaces);
        curNumOfEdges = result.size();

        iteration++;
    }
}
```

4.2.2 *Spark Design and Implementation*

Spark implementation to parallelize Minimum Spanning Tree application uses Spark abstraction, Resilient Distributed Datasets (RDD). Graph vertices maps to a `VertexRDD` and newly generated graph vertices map to new `VertexRDD` in each iteration, due to the immutable nature of Spark RDD.

In Spark implementation, minimum spanning tree happens through a series of transformations operators (such as `flatMapToPair()` and `flatMap()`) and finally, the driver gathers the edges of minimum spanning tree on each vertex via collect action operator.

Initially, vertices are generated in the graph. At every iteration of function execution, it is divided into two functions of `visited()` and `migrate()`. The active vertex examines the neighborhood and performs the following actions in `visited()` function. Get the edge with the minimum cost, if this edge will not form a loop on the current minimum spanning tree, add it. Otherwise, discard the edge and choose the edge with the second smallest cost. The `migration()` function will get the cutset of the new vertex in the broadcast variable and add to the cutset of current vertex. Then, calculate the number of edges in the current minimum spanning tree. If it is equal to the vertex minus one, the program ends; otherwise, continue to repeat the above steps. After repeating this procedure, the minimum spanning tree will be generated.

The following Listing 4.6 shows the `mst()` function of the graph bridge application, which covers the main function of calling Spark. `flatMap()` is an operator that allows vertices to be processed in parallel to find the edge with the minimum cost, and `reduce ()` is to deduplicate all vertices.

Listing 4.6 Code snippet of Minimum Spanning Tree using Spark

```
public class MST implements Serializable {
    private static void mst(String filePath, boolean printGeneratedRDD, boolean printEachRDD) {
        final Broadcast<ObjectRBTreeSet<Edge>[]> broadcastCutsets = jsc.broadcast(cutsets);

        int curNumOfEdges = 0;
        int targetNumOfEdges = (int) graphRDD.count() - 1;
        int iteration = 0;

        // Finding loop until all edges have been found.
        while (curNumOfEdges < targetNumOfEdges) {
            curNumOfEdges = 0;

            // Step 1. Visited each vertex
            graphRDD = visited(graphRDD, broadcastCutsets);

            // Count the current number of found edges.
            curNumOfEdges = countEdges(graphRDD);

            // Step 2. Migrate to the next vertex.
            graphRDD = migrate(graphRDD);
```

```

                iteration++;
            }
        }
}
```

4.3 STRONGLY CONNECTED COMPONENTS

4.3.1 *MASS Java Design and Implementation*

`GraphPlaces` and `GraphAgent` are also used to parallelize Graph Bridge application. GraphPlaces map to graph vertices that hold neighbors' information throughout the lifetime of the application. GraphAgent starts from each vertex, searches for strongly connected components in parallel in the graph and calculates the total number of strongly connected components.

When finding a strongly connected component in a graph, one concept need to be considered: Low value. The vertices will update the low values during the backtracking session, so that the vertices are in the same component if they have the same low values.

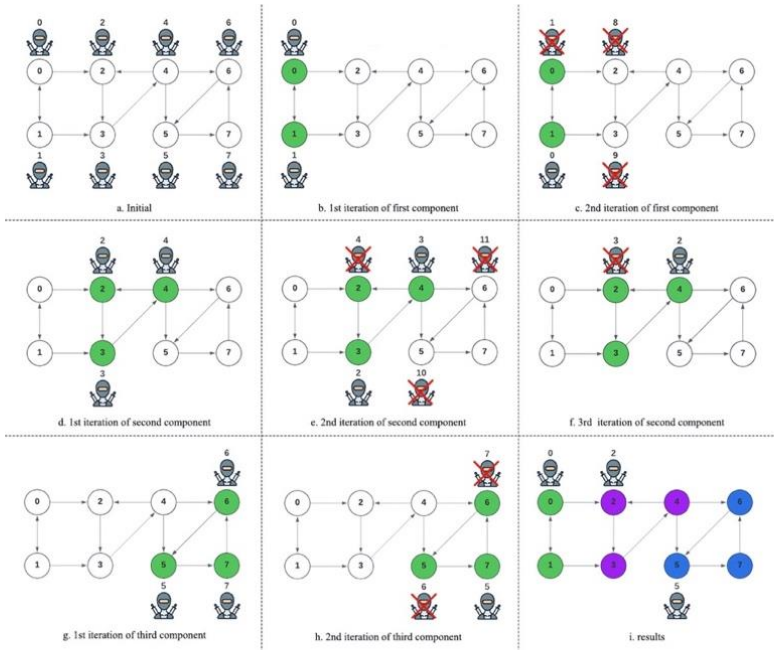

a. Initial
b. 1st iteration of first component
c. 2nd iteration of first component
d. 1st iteration of second component
e. 2nd iteration of second component
f. 3rd iteration of second component
g. 1st iteration of third component
h. 2nd iteration of third component
i. results

Figure 4.3 Demonstration of Strongly Connected Components in parallel

As seen in Figure 4.3 (a), initially, `Places` are generated from the dataset file, and generated agents at each `Place`. At every iteration of `Agent` function execution, the agent examines the neighborhood and performs the following actions. If there is a valid neighbor, the `Agent` migrates to the neighbor with the smallest id, and then `spawn()` child agent with parent agent information in the remaining neighbors; otherwise, `kill()` this agent. At the same time, the agents update their own information (low value). If the `Agent` visits a certain vertex for the second time, it will enter a component and backtracking phase. The `Agent` will return along the original path, update the low value accordingly, and update the low value of all the `Places` in this component. If the agent visits the vertex for the third time, it means that a component is found, update the information of this component on the `Place`, and `kill()` the agent. After repeating this procedure until all agents are killed, all components will exist on places with smallest id. Finally, the `Main()` driver collects components from all places in parallel.

Listing 4.7 Code snippet of driver `SCC` in SCC using MASS Java

```
public class SCC {
    public static void main( String[] args ){
        // Generate Places.
        GraphPlaces graphPlaces = init(graphPath, printPlaces);
        scc(graphPlaces);
        stopMASS();
    }
}
```

4.3.2 *Spark Design and Implementation*

Spark implementation to parallelize Graph Bridge application uses basic Spark abstraction, Resilient Distributed Datasets (RDD). Graph vertices maps to a `JavaPairRDD` and newly generated graph vertices map to new `JavaPairRDD` in each iteration, due to the immutable nature of Spark RDD.

In Spark implementation, finding strongly connected components happens through a series of narrow-dependency transformations operators (such as `flatMapToPair()`). Finally, the driver gathers the components on each vertex via `collect()` action operator.

Initially, vertices are generated in the graph from the dataset file. At each iteration of function execution, the active vertex examines the neighborhood and performs the following actions. If there are valid neighbors (the neighbor vertex id is larger than the current vertex id), a different exploration path is generated. Otherwise, set the finish status of current vertex to *true*. If the vertex visits a certain vertex for the second time, it meets the backtracking conditions, obtain the vertices list of the current component and stores it on the collection accumulator and set the finish status of current vertex to *true*. After repeating this process until all vertices are inactive, all components will exist on the vertex. Finally, the driver collects all components from collection accumulator in parallel.

Listing 4.8 Code snippet of SCC using Spark

```
public class SCC implements Serializable {
    private static Object2ObjectMap<Integer, IntSet> scc(JavaPairRDD<Integer, Vertex> graphRDD) {
        final Broadcast<IntList[]> broadcastAdjList = jsc.broadcast(adjList);
        CollectionAccumulator<String> accumulatorComps = jsc.sc().collectionAccumulator("accumulatorComps");
        int iteration = 0;
        while (graphRDD.count() > 0) {
            // Step 1 of depth-first seach: Visited a vertex.
            graphRDD = visited(graphRDD);
            // Step 2 of depth-first seach: Expanded a vertex.
            graphRDD = expanded(graphRDD, broadcastAdjList, accumulatorComps);
            iteration++;
        }
        return processComponentResults(accumulatorComps);
    }
}
```

Chapter 5. PARALLEL PERFORMANCE AND PROGRAMMABILITY EVALUATION

This chapter presents experimental results on the MASS and Spark parallel implementations discussed in Chapter 4, and discusses performance analysis and programmability analysis of parallel applications.

In the performance analysis, we discusses according to the following methods:

Measure the performance: We use the same datasets as input data and the same execution environment (a cluster of eight to twelve computing nodes provided by the University of Washington, Bothell, DSLab) to compare the execution time of executing different parallel libraries version of the graph applications under different nodes (node 1, 2, 4, 8 or 12). Two rounds of testing were performed, with each application tested three times in each round. The final results is taken as the average of the three test results of the second round.

Optimized the applications after first round of testing: It should be noted that all applications are implemented and optimized using `Fastutil` [28], a fast and compact type-specific collection for Java. This optimization step only modifies the data structure type of variables in all applications, but does not change other parts of code. The optimization results are remarkable. Many out of memory errors that occurred in the first round of testing can be successfully completed in second round of testing.

Datasets: Applications were tested using the following Table 5.1 datasets. Among them, the Graph Bridge and the Strongest Connected Component applications use real datasets. These datasets are from biological networks and are scale-free and can generate many agents, which are very suitable for testing with corresponding graph applications in our project. And the Minimum

Spanning Tree uses datasets generated from GraphGen, a laboratory program. This is because no suitable dataset for testing the performance of the minimum spanning tree was found when searching for the real dataset. Most of the existing real datasets are belongs to social networks or communication networks, with a lot of additional information, and it will be tedious to convert the data set that can be used for the Minimum Spanning Tree.

Table 5.1 Datasets Graph Properties

Datasets	Vertices	Edges	Weighted	Directed	Graph Applications
Dolphin	62	318	No	No	Graph Bridge
Power	4,941	6,594	No	No	Graph Bridge
500	500	63,300	Yes	No	Minimum Spanning Tree
1000	1000	250,639	Yes	No	Minimum Spanning Tree
Yeast	688	1,078	No	Yes	Strongly Connected Components
P2P	6,301	20,777	No	Yes	Strongly Connected Components

In the programmability analysis, we discusses and evaluates according to the following methods or terminology:

LoC (Lines of Code): LoC is divided into three parts — Agent code, Place code, and Boilerplate code. 1) Agent code represents the lines of code that is only related to the Agent. MASS have agent code, but Spark don't have agent code; 2) Place code represents the lines of code that is only related to the Place. MASS have place code and the code in each operator in Spark can be regarded as place code; and 3) Boilerplate code represents the lines of standard code that are intended to initialize and manage the parallelization framework and are not related to application implementation or computation.

Boilerplate Code Ratio: It measures the percentage of boilerplate code to the total number of lines of code. Smaller boilerplate code ratio means easiness to set up the parallel environment.

Lines of code measurement: presents in this section omitted comment and blank lines and both versions followed standard java coding convention.

Cyclomatic Count: represents the number of `if`, `while`, `for`, and `switch` statements, which is a count of the number of decisions in the code. It is a software metric used to determine the complexity of an application. The higher the count, the more complex the code.
Parallel Methods or Operators Count: represents the number of used parallel methods. The higher the count, the more complex the code.

5.1 GRAPH BRIDGE

5.1.1 *Performance Comparison*

Table 5.2 and the left of Figure 5.1 shows that the execution speed of the MASS version increases and then decreases with the number of nodes when using the small data set Dolphin. It runs fastest on two computing nodes, and its execution time is reduced by 40.169% compared with that of one computing node. However, it is significantly slower when using 4 computing nodes. Such execution results reveal that MASS requires greater memory overhead and more communication time between nodes as the number of nodes increases during the execution of MASS version. There was no significant change in the execution speed of the Spark version, which revealed that when running a smaller data set, the resources provided by multiple computing nodes were offset by the overhead generated by the communication between the nodes.

The figure on the right of Figure 5.2 shows that the MASS version execution time decreases significantly with the increase of compute nodes when using the larger data set Power. This execution result reveals that when running a larger data set, the overhead associated with communication between nodes is less than the performance boost to the execution speed associated with more resources. In the Spark version, due to its immutable RDD feature, the new RDD generated by each iteration will prompt the recalculation of the RDD with previous dependence. Therefore, when fewer compute nodes are used, the memory will burst the stack due to excessive storage of RDD. This feature also affects the results of 4 and 8 computing nodes. Its performance is not as good as that of the MASS version.

Table 5.2 Performance Comparison of Graph Bridge

Dataset	Libraries	1 Node	2 Nodes	4 Nodes	8 Nodes
Dolphin	MASS	6.261	3.746	37.472	20.132
	Spark	35.233	32.138	32.233	33.884
Power	MASS	1514.241	1124.539	705.428	508.925
	Spark	OutofMemory	OutofMemory	742.194	602.852

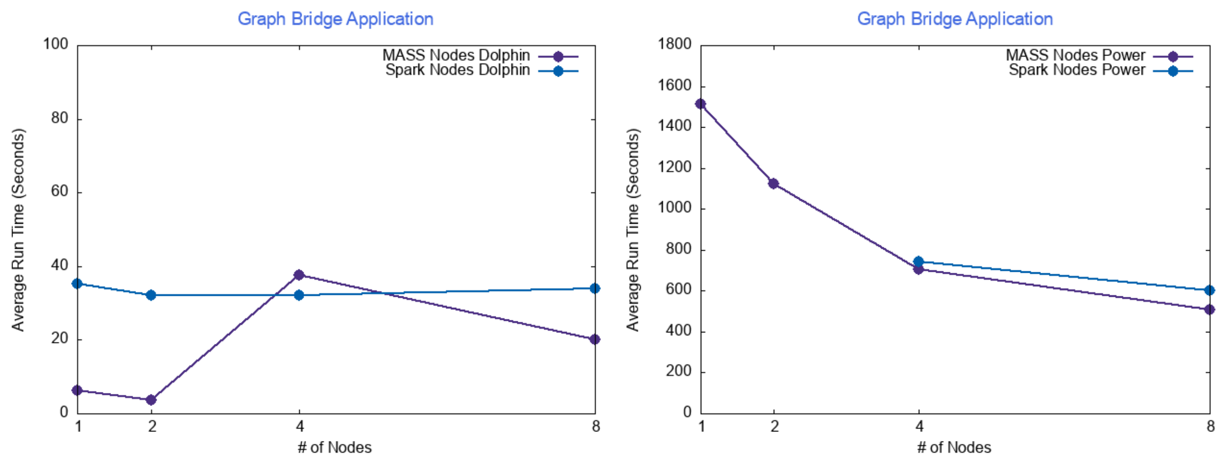


Figure 5.1 Performance Comparison of Graph Bridge using Dolphin

5.1.2 *Programmability Comparison*

Table 5.3 indicates that the overall LoC of MASS is lower than that of Spark due to the different parallel paradigms. The Boilerplate Code Ratio of MASS accounts for 11.73% of the total LoC, indicating that MASS is easy to set up parallel environment and execute applications. MASS version includes four classes (`MyAgent`, `MyVertex`, `ArgsToAgent`, `Bridge`), some of which have only parameterized constructors for passing parameters to specific agents or places during invocations of parallel methods. This framework-specific implementation increases the overall LoC. Because Spark cannot communicate directly between vertices, this causes it to add more logical judgments when transforming RDDs. For example, the Place object of the MASS version can store and update information, however, Spark has to called `reducingBy()` to achieve this feature, Spark needs to list all the situations one by one, so that it can accurately update the latest and correct version of fields when there are multiple versions generated in the same iteration.

Table 5.3 Lines of Code (LoC) and Boilerplate Code Ratio in Graph Bridge

Modules	MASS LoC	Spark LoC
Agent code	147	-
Place code	26	179
Boilerplate code	23	53
Total Loc	196	232
Boilerplate Code Ratio	11.73%	22.85%

Table 5.4 shows the branch counts comparison of this application, showing that the code of Spark version is more complex than MASS version. The Spark version uses a large number of `if()` conditional statements to list various situations in detail. In addition, the data structure of Spark version uses arrays to reduce memory usage. When transforming RDD, it is necessary to use the `for()` loop multiple times to traverse the neighbors to find the next vertex that needs to be visited.

Table 5.4 Cyclomatic Count in Graph Bridge

	MASS Cyclomatic Count	Spark Cyclomatic Count
if	18	38
while	2	1

for	3	11
switch	2	0
main()	1	1
Total	26	51

Table 5.5 presents the programmability analysis based on the number of parallel methods used in MASS and Spark version. Spark uses more parallel methods than the MASS version. MASS uses two parallel methods, `init()` and `returnResults()` in `VertexPlace` and uses three parallel methods, `init()`, `onArrival()` and `migrateTo()` in `GraphAgent`. Spark uses four transformation operators, `mapToPair()`, `flatMapToPair()`, `flatMap()` and `filter()` , and three action operators `count()`, `collect()` and `foreach()`. Spark uses more parallel operators than the MASS version implementation. The results show that MASS is more practical and implementable than Spark.

Table 5.5 Parallel Methods or Operators Count in Graph Bridge

MASS Parallel Methods		Spark Parallel Operators	
MASS Place	2	Transformations	4
MASS Agent	3	Actions	3
Total	5	Total	7

5.2 MINIMUM SPANNING TREE

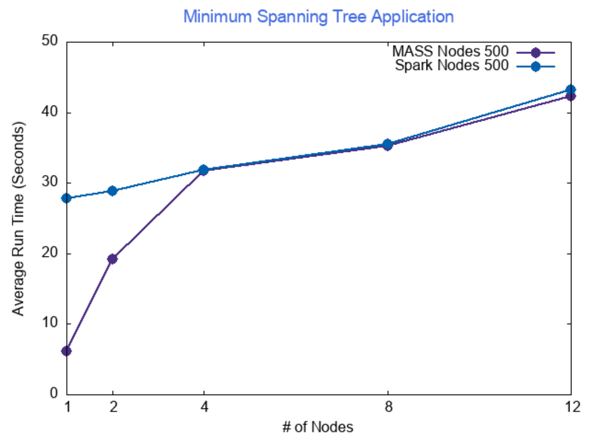
5.2.1 *Performance Comparison*

Table 5.6 and Figure 5.2 shows that the execution speed of the MASS version increases with the number of computing nodes. This execution result reveals that when the MASS version is executed, as the number of nodes increases, MASS requires greater memory overhead and more communication time between computing nodes. This is because in the later stage of application execution, the agent make more random jumps between computing nodes, which brings more overhead. The execution speed of the Spark version also increases with the increase of computing nodes, but it is not particularly significant, indicating that the resources provided by multiple computing nodes and the overhead generated by the communication between nodes are offset. Both figures in Figure 5.2 show that MASS outperforms Spark on a single computing node and two computing nodes, showing the superiority of MASS' computing power.

It should be noted that the MASS version can kill the most of the agents in 0.6 seconds, and there are only 3 active agents left. The Spark version found 493 edges out of a total of 499 in 4.2 seconds. Both versions show the disadvantage of parallel processing of the minimum spanning tree, requiring more time to find the last few vertices distributed on the cluster, in the later stage of execution, as the communication between computing nodes increases, and the increase in information stored by MASS Agent and Spark Vertex, this kind of communication brings huge overhead, which causes the execution time to take more than ten times the time when there are only a few active agents and active vertices left.

Table 5.6 Performance Comparison of Minimum Spanning Tree

Datasets	Libraries	1 Node	2 Nodes	4 Nodes	8 Nodes	12 Nodes
500	MASS	6.158	19.184	31.777	35.250	42.323
	Spark	27.843	28.859	31.881	35.508	43.188
1000	MASS	37.440	64.883	85.989	103.560	116.129
	Spark	85.592	87.224	90.065	92.136	100.720



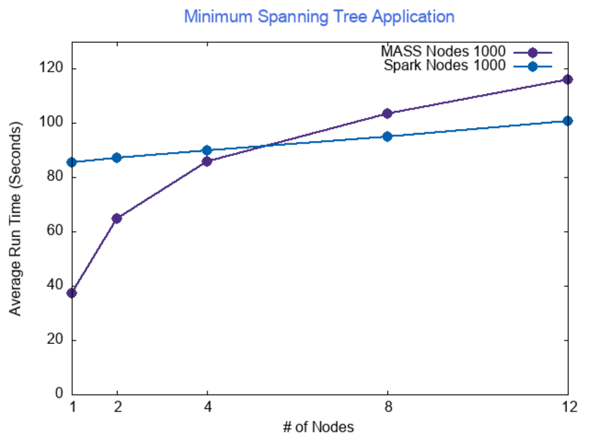


Figure 5.2 Performance Comparison of Minimum Spanning Tree

5.2.2 *Programmability Comparison*

Table 5.7 shows that the overall LoC of MASS is similar to Spark. But the Boilerplate Code Ratio of MASS only accounts for 11.73% of the total loc, indicating that MASS is easy to set up a parallel environment and execute applications. MASS version includes 3 classes (`MyAgent`, `MyVertex`, `Edge`), some of which have only parameterized constructors for passing parameters to specific agents or positions during invocation of parallel methods. This framework-specific implementation increases the overall LoC. Spark increases the overall LoC because it requires more logical judgments.

Table 5.7 Lines of Code (LoC) and Boilerplate Code Ratio in MST

Modules	MASS LoC	Spark LoC
Agent code	122	-
Place code	32	135
Boilerplate code	25	45
Total Loc	179	180
Boilerplate Code Ratio	13.96%	25%

Table 5.8 shows the branch counts comparison of minimum spanning tree application, showing that the code of Spark version is more complex than MASS version. When the Spark version needs to enumerate various situations in detail, it uses many `if` conditional statements.

Table 5.8 Cyclomatic Counts in MST

	MASS Cyclomatic Count	Spark Cyclomatic Count
if	16	29
while	1	1
for	5	6
switch	2	0
main()	1	1
Total	25	37

Table 5.9 presents the programmability analysis based on the number of parallel methods used in MASS and Spark version. Spark uses more parallel methods than the MASS version. MASS uses two parallel methods, `init()` and `returnResults()` in `VertexPlace` and uses three parallel methods, `init()`, `onArrival()` and `migrateTo()` in `GraphAgent`. Spark uses four transformation operators, `mapToPair()`, `flatMapToPair()` and `flatMap()`, and three action operators `count()`, `collect()` and `foreach()`. Consistent with the previous discussion results, Spark uses more parallel operators than the MASS version implementation. The results show that MASS is more practical and implementable than Spark.

Table 5.9 Parallel Methods or Operators Count in MST

MASS Parallel Methods		Spark Parallel Operators	
MASS Place	2	Transformations	3
MASS Agent	3	Actions	3
Total	5	Total	6

5.3 Strongly Connected Components

5.3.1 *Performance Comparison*

Table 5.10 and Figure 5.3 shows that the MASS version consistently performs better than the Spark version, and that frequent communication between vertices via agents can significantly improve performance. Furthermore, as the number of nodes increases, the MASS version has a significant speed-up, with time reductions of 53%, 52%, and 27%, respectively. Judging from the execution results, this shows that MASS's flexible communication produces low overhead without affecting performance. However, Table 5.10 shows that both MASS and Space encountered an out-of-memory when testing a large data set, indicating that for this strongly connected components application, MASS agent and Spark vertex carry a large amount of information for calculation, which brings huge overhead as the edges of dataset increases.

Table 5.10 Performance Comparison of Strongly Connected Components

Datasets	Libraries	1 Node	2 Nodes	4 Nodes	8 Nodes
Yeast	MASS	220.044	104.216	49.172	35.893
	Spark	289.873	146.811	72.612	48.432
P2P	MASS	OutofMemory	OutofMemory	OutofMemory	731.33
	Spark	OutofMemory	OutofMemory	OutofMemory	OutofMemory

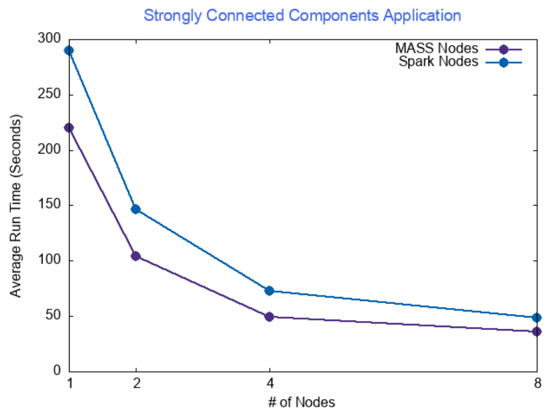


Figure 5.3 Performance Comparison of Strongly Connected Components

5.3.2 *Programmability Comparison*

Table 5.11 Lines of Code (LoC) and Boilerplate Code Ratio in SCC

Modules	MASS LoC	Spark LoC
Agent code	135	-
Place code	29	167
Boilerplate code	29	47
Total Loc	193	214
Boilerplate Code Ratio	15.03%	21.96%

Table 5.11 indicates that the overall LoC of MASS is lower than that of Spark due to the different parallel paradigms. The Boilerplate Code Ratio of MASS accounts for 15.03% of the total LoC, indicating that MASS is easy to set up parallel environment and execute applications. MASS version includes three classes (`MyAgent`, `MyVertex`, `ArgsToAgent`), some of which have only parameterized constructors for passing parameters to specific agents or places during invocations of parallel methods. This framework-specific implementation increases the overall LoC. Spark increases the overall LoC because it requires more logical judgments.

Table 5.12 Cyclomatic Count in SCC

	MASS Cyclomatic Count	Spark Cyclomatic Count
if	14	29
while	1	1
for	3	11
switch	2	0
main()	1	1
Total	21	42

Table 5.12 below shows the branch counts comparison of this application, showing that the code of Spark version is more complex than MASS version. When the Spark version needs to enumerate various situations in detail, it uses many if conditional statements.

Table 5.13 Parallel Methods or Operators Count in SCC

MASS Parallel Methods		Spark Parallel Operators	
MASS Place	2	Transformations	2
MASS Agent	3	Actions	3
Total	5	Total	5

Table 5.13 above presents the programmability analysis based on the number of parallel methods used in MASS and Spark version. MASS version uses the same number of parallel methods as Spark. MASS uses two parallel methods, `init()` and `returnResults()` in `VertexPlace` and uses three parallel methods, `init()`, `onArrival()` and `migrateTo()` in `GraphAgent`. Spark uses two transformation operators, `mapToPair()` and `flatMapToPair()` and three action operators `count()`, `collect()` and `foreach()`. Consistent with the previous discussion results, Spark uses more parallel operators than the MASS version implementation. The results show that MASS is more practical and implementable than Spark.

Chapter 6. CONCLUSION & FUTURE WORK

This project completed the parallel implementation of three graph applications Graph Bridge, Minimum Spanning Tree, and Strongly Connected Components using two frameworks MASS Java and Spark Java, and compared the performance and programmability of these two frameworks. All three applications were executed using at least 1, 2, 4, and 8 compute nodes.

The execution results show that, for testing with small datasets, the MASS Java version performs best when running on a single computing node or two computing nodes, but when running on more than 4 computing nodes, the performance of MASS Java degrades rapidly due to high overhead. The performance of the Spark version does not change significantly with different computing nodes. For testing with large datasets, the MASS Java version significantly improved execution performance as the number of computing nodes increased, and the execution times were consistently lower than Spark. Additionally, we compared programmability as well. After calculating Lines of Code, Boilerplate Code Ratio, and Cyclomatic Count per application, and measuring and evaluating the programmability, the conclusions show that: the programmability of the MASS Java version has generally better than the Spark version.

This project verifies the correctness of the parallel results by comparing the parallel results of MASS Java with those of Spark, as well as evaluating the performance and programmability. Three graph applications implemented based on MASS and Spark shows that MASS has generally better performance and programmability. Different from the Spark version, MASS supports communication between vertices, which greatly increases the flexibility of algorithm implementation and reduces the complexity of code. In addition, from the perspective of development engineers or users, MASS allows users to write their solutions from the perspective of the driver's seat and more suitable for developing graph applications.

As for future work, graph applications developed by MASS Java can be compared with more libraries for performance and programmability, such as JgraphT [30] and ringo [31] (Java graphic libraries), to obtain a more comprehensive performance comparison analysis of MASS in the field of parallel frameworks.

The source code for the MASS Java and Spark implementations is available in the repository (https://bitbucket.org/mass_application_developers/mass_java_appl/branch/caroline/graph_applications).

BIBLIOGRAPHY

[1] J. A. Bondy; U. S. R. Murty, Graph Theory with Applications, University of Waterloo, 1982.

[2] DSLab, "Multi-Agent Spatial Simulation (MASS)," University of Washington, Bothell, [Online]. Available: http://depts.washington.edu/dslab/MASS/index.html.

[3] DSLab, "Distributed System Lab (DSLab)," University of Washington, Bothell, [Online]. Available: https://depts.washington.edu/dslab/.

[4] DSLab, "MASS Java Manual," University of Washington, Bothell, 31 Aug 2016. [Online]. Available: https://depts.washington.edu/dslab/MASS/docs/MASS%20Java%20Technical%20Manual.pdf.

[5] DSLab, "MASS C++: Parallel-Computing Library for Multi-Agent Spatial Simulation," University of Washington, Bothell, [Online]. Available: https://depts.washington.edu/dslab/MASS/docs/MassCpp.pdf.

[6] DSLab, "MASS CUDA: Parallel-Computing Library for Multi-Agent Spatial Simulation," [Online]. Available: https://depts.washington.edu/dslab/MASS/docs/MassCuda.pdf.

[7] Jason Woodring; Matthew Sell; Munehiro Fukuda; Hazeline Asuncion; Eric Salathe, "A Multi-Agent Parallel Approach to Analyzing Large Climate Data Sets," in *In 37th IEEE International Conference on Distributed Computing Systems*, Atlanta, GA, 2017.

[8] "Apache Spark," [Online]. Available: https://spark.apache.org/.

[9] "Spark RDD Programming Guide," [Online]. Available: https://spark.apache.org/docs/latest/rdd-programming-guide.html.

[10] J. Donner, After Access: Inclusion, Development, and a More Mobile Internet, The MIT Press, 2015.

[11] Cevher, Volkan; Becker, Stephen; Schmidt, Mark, "Convex Optimization for Big Data: Scalable, randomized, and parallel algorithms for big data analytics," *IEEE Signal Processing Magazine (Volume: 31, Issue: 5),* 2014.

[12] Charles M. Macal; Michael J. North, "Agent-based modeling and simulation," in *Proceedings of the 2009 Winter Simulation Conference (WSC),* 2009.

[13] B. Luger, "Distributed Graph Data Structures," [Online]. Available: http://depts.washington.edu/dslab/MASS/reports/BrianLuger_su21.pdf.

[14] Brian Luger; Munehiro Fukuda, "Wiki Graph Programming with MASS Java," [Online]. Available: https://bitbucket.org/mass_library_developers/mass_java_core/wiki/GraphPlaces%20Class%20and%20Cytoscape%20Integration.

[15] J. Gilroy, "Dynamic Graph Construction and Maintenance," DSLab, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/JustinGilroy_whitepaper.pdf.

[16] Justin Gilroy; Satine Paronyan; Jonathan Acoltzi; Munehiro Fukuda, "Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory," [Online]. Available: http://faculty.washington.edu/mfukuda/papers/biggraphs20.pdf.

[17] "Spark Cluster Mode Overview," [Online]. Available: https://spark.apache.org/docs/latest/cluster-overview.html.

[18] "Spark GraphX," Apache, [Online]. Available: https://spark.apache.org/docs/3.2.1/api/java/org/apache/spark/graphx/Pregel.html.

[19] Grzegorz Malewicz; Matthew H. Austern; Aart J. C. Bik, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data, ACM, pp. 135-146*, New York, NY, USA, 2010.

[20] R. Tarjan, "Tarjan's bridge-finding algorithm," [Online]. Available: https://en.wikipedia.org/wiki/Bridge_(graph_theory)#Tarjan's_bridge-finding_algorithm.

[21] Kwara, Nigeria; Joyce Ayoola; Emmanuel Asani, "A Comparative Study Of Minimal Spanning Tree Algorithms," *2020 International Conference in Mathematics, Computer Engineering and Computer Science (ICMCECS)*, 2020.

[22] R. Tarjan, "Tarjan's strongly connected components algorithm," [Online]. Available: https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm#:~:text=Tarjan's%20strongly%20connected%20components%20algorithm%20is%20an%20algorithm%20in%20graph,path%2Dbased%20strong%20component%20algorithm..

[23] "GraphLab," Carnegie Mellon University, [Online]. Available: https://en.wikipedia.org/wiki/GraphLab.

[24] Y. Low, "GraphLab: A Distributed Abstraction for Large Scale Machine Learning," [Online]. Available: https://www.cs.cmu.edu/~ylow/thesis/thesis.pdf.

[25] "Benchmarking of Distributed Computing Engines Spark and GraphLab for Big Data Analytics," *2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService), 2016, pp. 10-13, doi: 10.1109/BigDataService.2016.11.*.

[26] "GraphMat," [Online]. Available: https://github.com/narayanan2004/GraphMat.

[27] Narayanan Sundaram; Nadathur Satish; Md Mostofa Ali Patwary, "GraphMat: high performance graph analytics made productive," [Online]. Available: https://dl.acm.org/doi/10.14778/2809974.2809983.

[28] "fastutil: Fast & compact type-specific collections for Java™," [Online]. Available: https://fastutil.di.unimi.it/.

[29] M. Seto, "Graphing Applications in Spark and MASS Java," University of Washington, Bothell, [Online]. Available: http://depts.washington.edu/dslab/MASS/reports/MaxSeto_su21.pdf.

[30] B. Naveh, "JGraphT," [Online]. Available: https://jgrapht.org/.

[31] SNAP, "Ringo: In-Memory Graph Exploration System," Stanford Network Analysis Platform (SNAP), [Online]. Available: http://snap.stanford.edu/ringo/.

APPENDIX

Appendix shows the command lines to execute the applications in VS Code terminal.

0. Before running Spark applications

Download `fastuti-7.2.0.jar` from http://www.java2s.com/example/jar/f/download-fastutil720jar-file.html, and put it in this directory.

```
/home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar
```

1.1 Compile and run the MASS Java version of Graph Bridge

1) Compile

```
$ cd mass_java_appl/Graphs/GraphBridge/GraphBridge_MASS
$ mvn clean install package
```

2) Add `fastutil` dependency in maven `pom.xml`

```
<!-- https://mvnrepository.com/artifact/it.unimi.dsi/fastutil -->
<dependency>
    <groupId>it.unimi.dsi</groupId>
    <artifactId>fastutil</artifactId>
    <version>8.5.2</version>
</dependency>
```

3) Add `run.sh` file

```
cd target
java -jar GraphBridge-1.0-SNAPSHOT.jar
/home/NETID/<YOUR_NETID>/mass_java_appl/Graphs/GraphBridge/GraphBridge_
MASS/InputGraphs/UndirectedGraphs/$1.dsl $2 $3
```

4) Run

```
$ ./run.sh <graph_dsl_file_name> <print_Places> <print_results_bridges>
```

** The default value for both Boolean values are `false`.*

5) Example:

```
$ ./run.sh dataset true true
```

1.2 Compile and run the Spark version of Graph Bridge

1) Add `compile.sh` file

```
/usr/lib/jvm/java-1.8.0/bin/javac -cp /home/NETID/<YOUR_NETID>/spark-
2.3.1-bin-hadoop2.7/jars/spark-core_2.11-
2.3.1.jar:/home/NETID/<YOUR_NETID>/spark-2.3.1-bin-
hadoop2.7/jars/spark-sql_2.11-2.3.1.jar:/home/NETID/<YOUR_NETID>/spark-
2.3.1-bin-hadoop2.7/jars/scala-library-
2.11.8.jar:/home/NETID/<YOUR_NETID>/spark-2.3.1-bin-
hadoop2.7/jars/spark-graphx_2.11-2.3.1.jar:google-collections-
1.0.jar:/home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar $1.java
$2.java
/usr/lib/jvm/java-1.8.0/bin/jar -cvf $1.jar $1*.class $2*.class
```

2) Compile

```
./compile.sh GB Vertex
```

3) Add `run.sh` file

```
# Run local
spark-submit --class GB --master local --jars
/home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar GB.jar
/home/NETID/<YOUR_NETID>/mass_java_appl/Graphs/GraphBridge/GraphBridge_
Spark/UndirectedGraphs/$1.txt $2 $3 $4

# Run on clutser
# spark-submit --class GB --master spark://cssmpi6h.uwb.edu:<YOUR_PORT>
--driver-memory 14G --executor-memory 14G --executor-cores 4 --jars
/home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar GB.jar
```

```
/home/NETID/<YOUR_NETID>/mass_java_appl/Graphs/GraphBridge/GraphBridge_
Spark/UndirectedGraphs/$1.txt $2 $3 $4
```

4) Run

```
./run.sh <graph_txt_file_name> <print_results_bridges>
<print_generated_graphRDD> <print_each_graphRDD>
```

** The default value for all Boolean values are `false`. The last Boolean is mostly useful for debugging.*

5) Example:

```
./run.sh dataset true true true
```

2.1 Compile and run the MASS Java version of Minimum Spanning Tree

1) Compile

```
$ cd mass_java_appl/Graphs/MinimumSpanningTree/MinimumSpanningTree_MASS
$ mvn clean install package
```

2) Add `fastutil` dependency in maven `pom.xml`

```
<!-- https://mvnrepository.com/artifact/it.unimi.dsi/fastutil -->
<dependency>
    <groupId>it.unimi.dsi</groupId>
    <artifactId>fastutil</artifactId>
    <version>8.5.2</version>
</dependency>
```

3) Add `run.sh` file

```
cd target
java -jar MinimumSpanningTree-1.0-SNAPSHOT.jar
/home/NETID/<YOUR_NETID>/mass_java_appl/Graphs/MinimumSpanningTree/Mini
mumSpanningTree_MASS/InputGraphs/UndirectedGraphs/$1.dsl $2 $3 $4
```

4) Run

```
./run.sh <graph_dsl_file_name> <#_of_agents> <print_Places>
<print_results_mst>
```

** The default value for both Boolean values are `false`.*

5) Example:

```
./run.sh dataset 8 true true
```

2.2 Compile and run the Spark version of Minimum Spanning Tree

1) Add `compile.sh` file

```
/usr/lib/jvm/java-1.8.0/bin/javac -cp /home/NETID/<YOUR_NETID>/spark-
2.3.1-bin-hadoop2.7/jars/spark-core_2.11-
2.3.1.jar:/home/NETID/<YOUR_NETID>/spark-2.3.1-bin-
hadoop2.7/jars/spark-sql_2.11-2.3.1.jar:/home/NETID/<YOUR_NETID>/spark-
2.3.1-bin-hadoop2.7/jars/scala-library-
2.11.8.jar:/home/NETID/<YOUR_NETID>/spark-2.3.1-bin-
hadoop2.7/jars/spark-graphx_2.11-2.3.1.jar:google-collections-
1.0.jar:/home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar $1.java
$2.java $3.java
/usr/lib/jvm/java-1.8.0/bin/jar -cvf $1.jar $1*.class $2*.class
$3*.class
```

2) Compile

```
./compile.sh MST Vertex Edge
```

3) Add `run.sh` file

```
# Run local
spark-submit --class MST --master local --driver-memory 14G --executor-
memory 14G --executor-cores 4 --jars
/home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar MST.jar
/home/NETID/<YOUR_NETID>/mass_java_appl/Graphs/MinimumSpanningTree/Mini
mumSpanningTree_Spark/InputGraphs/UndirectedGraphs/$1.txt $2 $3 $4

# Run on clutser
# spark-submit --class MST --master
spark://cssmpi6h.uwb.edu:<YOUR_PORT> --driver-memory 14G --executor-
memory 14G --executor-cores 4 --jars
/home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar MST.jar
/home/NETID/<YOUR_NETID>/mass_java_appl/Graphs/MinimumSpanningTree/Mini
mumSpanningTree_Spark/InputGraphs/UndirectedGraphs/$1.txt $2 $3 $4
```

4) Run

```
./run.sh <graph_txt_file_name> <print_results_mst>
<print_generated_graphRDD> <print_each_graphRDD>
```

** The default value for all Boolean values are `false`. The last Boolean is mostly useful for debugging.*

5) Example:

```
./run.sh dataset true true true
```

3.1 Compile and run the MASS Java version of Strongly Connected Components

1) Compile

```
$ cd
mass_java_appl/Graphs/StronglyConnectedComponents/StronglyConnectedComp
onents_MASS
$ mvn clean install package
```

2) Add `fastutil` dependency in maven `pom.xml`

```
<!-- https://mvnrepository.com/artifact/it.unimi.dsi/fastutil -->
<dependency>
    <groupId>it.unimi.dsi</groupId>
    <artifactId>fastutil</artifactId>
    <version>8.5.2</version>
</dependency>
```

3) Add `run.sh` file

```
cd target
```

```
java -jar StronglyConnectedComponents-1.0-SNAPSHOT.jar
/home/NETID/<YOUR_NETID>/mass_java_appl/Graphs/StronglyConnectedCompone
nts/StronglyConnectedComponents_MASS/InputGraphs/DirectedGraphs/$1.dsl
$2 $3
```

4) Run

```
$ ./run.sh <graph_dsl_file_name> <print_Places> <print_results_scc>
```

** The default value for both Boolean values are `false`.*

5) Example:

```
$ ./run.sh dataset true true
```

3.2 Compile and run the Spark version of Strongly Connected Components

1) Add `compile.sh` file

```
/usr/lib/jvm/java-1.8.0/bin/javac -cp /home/NETID/<YOUR_NETID>/spark-
2.3.1-bin-hadoop2.7/jars/spark-core_2.11-
2.3.1.jar:/home/NETID/<YOUR_NETID>/spark-2.3.1-bin-
hadoop2.7/jars/spark-sql_2.11-2.3.1.jar:/home/NETID/<YOUR_NETID>/spark-
2.3.1-bin-hadoop2.7/jars/scala-library-
2.11.8.jar:/home/NETID/<YOUR_NETID>/spark-2.3.1-bin-
hadoop2.7/jars/spark-graphx_2.11-2.3.1.jar:google-collections-
1.0.jar:/home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar $1.java
$2.java
/usr/lib/jvm/java-1.8.0/bin/jar -cvf $1.jar $1*.class $2*.class
```

2) Compile

```
./compile.sh SCC Vertex
```

3) Add `run.sh` file

```
# Run local
# spark-submit --class SCC --master local --driver-memory 14G --
executor-memory 14G --executor-cores 4 --jars
/home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar SCC.jar
```

```
/home/NETID/<YOUR_NETID>/mass_java_appl/Graphs/StronglyConnectedCompone
nts/StronglyConnectedComponents_Spark/InputGraphs/DirectedGraphs/$1.txt
$2 $3 $4

# Run on clutser
spark-submit --class SCC --master spark://cssmpi6h.uwb.edu:
<YOUR_PORT>--driver-memory 14G --executor-memory 14G --executor-cores 4
--jars /home/NETID/<YOUR_NETID>/jars/fastutil-7.2.0.jar SCC.jar
/home/NETID/<YOUR_NETID>/mass_java_appl/Graphs/StronglyConnectedCompone
nts/StronglyConnectedComponents_Spark/InputGraphs/DirectedGraphs/$1.txt
$2 $3 $4
```

4) Run

```
./run.sh <graph_txt_file_name> <print_results_scc>
<print_generated_graphRDD> <print_each_graphRDD>
```

** The default value for all Boolean values are `false`. The last Boolean is mostly useful for debugging.*

5) Example:

```
./run.sh dataset true true true
```