Graph Streaming in MASS Java

Yan Hong

A whitepaper

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2022

Program Authorized to Offer Degree:

Computer Science & Software Engineering

University of Washington

Abstract

Graph Streaming in MASS Java

Yan Hong

Chair of the Supervisory Committee:
Dr.Munehiro Fukuda
Computing & Software Systems

This project is to facilitate graph streaming in agent-based big data computing where agents find a shape or attributes of a huge graph. Analyzing and processing massive graphs in general has become an important task in different domains because many real-world problems can be represented as graphs such as biological networks and neural networks. Those graphs can have millions of vertices and edges. It is quite challenging to process such a huge graph with limited resources as well as in a reasonable timeframe. The MASS (Muti-Agent Spatial Simulation) library has already supported graph data structure (GraphPlaces) which is distributed on a cluster of computing nodes. However, when processing a big graph, we may still encounter the following two problems. The first is the construction overhead that will delay the actual computation. The second is limited resources that slow down graph processing. To solve those two problems, we implemented graph streaming in MASS Java which repetitively reads a portion of a graph and processes it while reading the next graph portion. It supports HIPPIE and MATSim file formats as the input graph files. We also implemented two graph streaming benchmarks: Triangle Counting and Connected Components, to verify the correctness of and evaluate the performance of graph streaming. Those two programs were executed with 1 - 24

computing nodes, which demonstrates the significant CPU-scalable and memory-scalable performance improvements. We also compared the performance with the non-streaming solution. Graph streaming avoids the explosive growth of the agent population and loads only a small portion of a graph, both efficiently using limited memory space.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

This chapter starts with a brief introduction to MASS as this graph streaming project is built into it. Then we will discuss why we should do graph streaming and what are the project goals I would like to achieve in this project.

1.1 BACKGROUND

MASS (Multi-Agents Spatial Simulation) is an agents-based parallel programming library to do computation over a cluster of nodes [1]. It provides an intuitive programming framework to do big data processes and can simulate a lot of real-life problems like bioinformatics, climate change, social networking, etc. Remote nodes fork the process, and the processes communicate with each other via TCP connections.

There are two key concepts in the MASS library: Places and Agents. Places is a distributed array of place elements over a cluster of computing nodes. It is managed by a set of global indices and each place element can be identified with an index. Data can be saved in a specific place and exchanged between places.

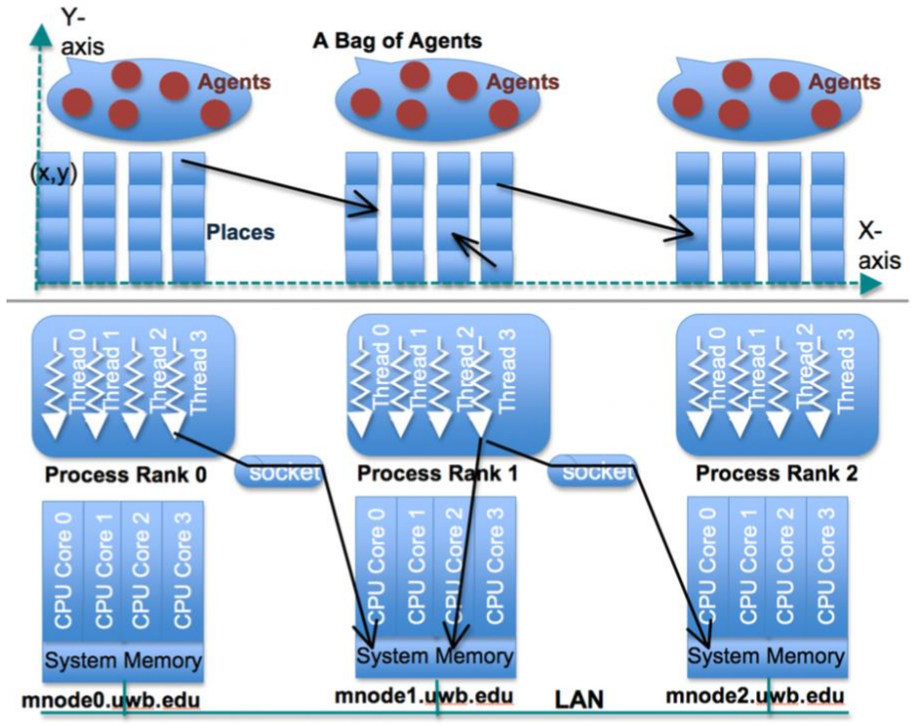


Figure 1.1 MASS library data model

Agents are a set of execution instances that can migrate over places. Each agent object can specify the next place's index to indicate where to migrate next. When an agent resides on a place, it can manipulate the data saved on that place. Agents are grouped into bundles. On each computing node, multiple threads check-in each agent one by one and process each agent's request.

On top of the Places class, MASS also supports other data structures including Binary Tree, QuadTree, Continuous Space [2], and Graph [3]. Those agent-navigable can better meet users' various computing needs.

1.2 MOTIVATION

Analyzing and processing massive graphs has become an important task in different domains because many real-world problems can be represented as a graph, for example, social networking, biological networks, and neural relationships. Those graphs can have millions of vertices and edges. It's quite challenging to process such a huge graph with limited resources as well as in a reasonable time. To meet the demands of processing rapidly growing graphs, some solutions have emerged. Currently, parallelization and using distributed memory are common techniques to solve huge graph problems. The MASS library has already supported graph data structure (GraphPlaces) which is distributed on a cluster of computing nodes. However, when processing a big graph, we may still encounter the following two problems.

The first problem is that with a big graph, the construction of the whole graph is a big overhead which will delay the actual computation. According to an experiment conducted by Brain Luger, reading and constructing a graph from a 69 GB Hippie file takes more than 70 minutes [3] which means we have to wait for more than 70 minutes before processing the graph. And any subgraphs that have been loaded in the memory can only stay there.

The other problem is that with limited resources, processing a huge graph could be very slow. When using the MASS library to solve problems, a lot of the applications rely on the migration of agents. To process a graph, we should first create an agent on each of the vertices. The larger a graph is, the more agents are created. In most cases, agents should spawn child agents which will result in much more agents. The most common functions in Agents class like *callAll(), manageAll()* and *migrate()* require to process all the alive agents. When those functions are invoked, multiple threads on each computing node take care of each agent one by one, so processing a larger bag of agents will take a much longer time.

This is the motivation to implement graph streaming in MASS, which repetitively reads a portion of the graph and processes it while reading the next graph portion.

1.3 PROJECT GOALS

This project aims to implement graph streaming in the MASS Java library which includes:

1) Support to stream big graph files to distributed memory. During the streaming, do computation and construction at the same time so that we can reduce the construction overhead.
2) Support HIPPIE (Human Integrated Protein to Protein Interaction Reference) files and MATSim (Multi-Agent Transport Simulation) files as the input graph files.
3) Implement two graph streaming benchmark programs to verify the correctness and performance of the graph streaming solution. By evaluating those two programs, we would like to prove that graph streaming can improve the performance of processing big graphs, and it can help to solve the problem when memory is not enough to process the whole graph.

For the concept of “big graph files”, the biggest file that strictly follows the HIPPIE format is 50 MB. The biggest protein-to-protein file we can find is 69 GB which doesn’t strictly follow the HIPPIE format. Eventually, we are aiming to process the 69 GB file, but for my project, when verifying the implementation and conducting the performance evaluation, we will target files that meet the following requirements: reading the whole file and constructing the whole graph with a single node takes about 5 – 10 minutes, processing the whole graph on a single node takes more than 30 minutes. Those two requirements can make sure my experiments are more doable as well as provide meaningful insights.

CHAPTER 2. RELATED WORK

This chapter intends to overview the MASS library from the viewpoint of graph programming and to differentiate MASS graph streaming from related projections.

2.1 GRAPHS IN MASS

2.1.1 *GraphPlaces*

The MASS library has already supported the graph data structure. It was originally implemented by Justin Gilroy [4] and refactored by Brain Luger [3]. In the MASS library, the graph data structure is called GraphPlaces. GraphPlaces is an extension of the existing MASS Places class with the capability to support simulations. Each GraphPlaces consists of vertices. Vertices with the graph are distributed across all nodes as shown in Figure 2.1 below. When we add vertices to the graph, we will follow a round-robin fashion so that the vertices are balanced across the entire cluster. Each vertex was represented by a VertexPlace object and the VertexPlace class is an extension of the MASS Place class. The vertex contains a list of outgoing edges and information about the vertex itself.

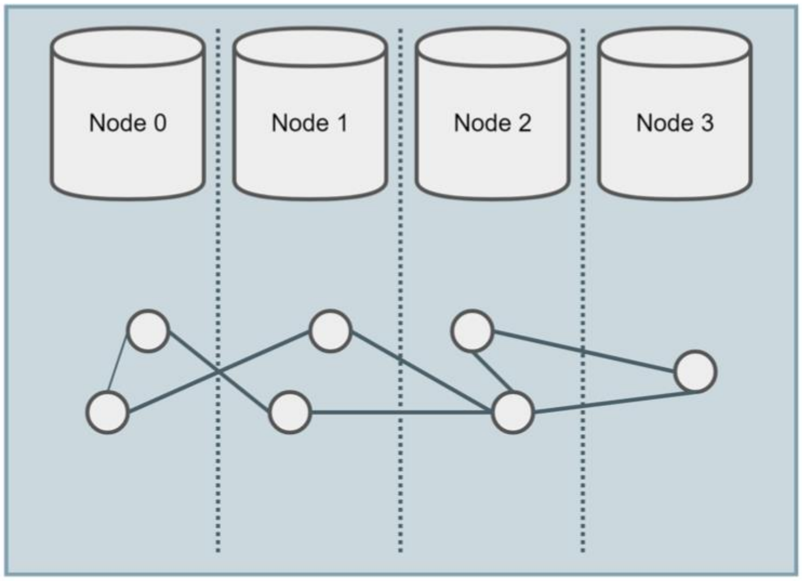


Figure 2.1 Visual representation of a distributed graph

When constructing a graph, users can load the graph from a supported graph format including Hippie, Matsim, UW-Bothell proprietary DSL, and SAR. Users can also choose to add vertices and edges manually by calling *Addvertex() and AddEdge()* methods.

On each computing node, vertices are stored in a single vector as shown in Figure 2.2. The single vector can grow dynamically with the size of the graph so that we don't have to know the graph size in advance. The implementation also has a removal queue as shown in the red rectangle of Figure 2.2. It was used to recycle the removed vertices so that we don't have to shift all the elements in the container. When a vertex is removed, it will be added to the queue. On the next call to addVertex(), the recycled vertices will be dequeued. After constructing a graph, agents can migrate over an edge from one vertex to another.

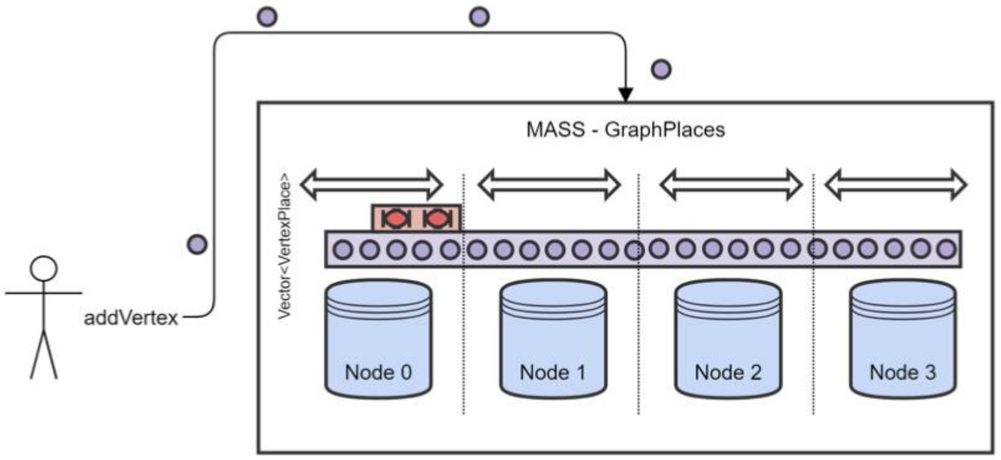


Figure 2.2 Vertex container and distribution

2.1.2 *Hippie & MATSim*

HIPPIE stands for Human Integrated Protein to Protein Interaction Reference [5]. The MASS library has supported loading graphs from HIPPIE files. To provide additional support for HIPPIE data, Brian Luger implemented a new class called **Hippie** in the *graph* package. The Hippie class is extended from GraphPlaces and maintains more contextual data. Along with the Hippie class, HippieVertex and HippieEdge have also been implemented to allow users to better represent HIPPIE data within MASS. HippieVertex contains the protein key and ID, and HippieEdge contains the weight of edges and other contextual data.

MATSim stands for Muti-Agent Transport Simulation [6]. It is also a supported graph file format in MASS. Like the Hippie class, **Matsim**, **MatsimVertex,** and **MatsimEdge** have been implemented within the *graph* package to better represent MATSim data. The Matsim class is also extended from GraphPlaces and the MatsimVertex class is extended from the VertexPlace class. Those classes maintain additional contextual data to enable developers to conduct simulations in MASS.

Since my project supports Hippie files and MATSim files, graph streaming is implemented on top of the Hippie class and Matsim class.

2.2 Spark Streaming and GraphX

2.2.1 *GraphX*

GraphX [8] is a distributed graph processing framework built on top of Spark. Spark, a data-parallel computing tool, runs as a set of processes. The main application also called the driver works to coordinate the master node and the worker nodes. The most important concept in Spark is RDD (Resilient Distributed Datasets) which is an immutable distributed collection of objects. Each RDD is divided into several partitions and processed on different computing nodes.

GraphX utilizes the RDD concept and represents a graph using two RDDs, *VertexRDD* and *EdgeRDD*. It provides many popular graph-related operators like *mapVertices*(), *mapEdges*(), *joinVertices*(), *groupEdges*() etc. Since vertices should be processed in the context of neighbors, GraphX also introduces the *triplet* concept which can join the structure of vertices and edges as shown in Figure 2.3 below. Besides that, GraphX implements the Pregel API [9], which automates repetitive messages passing from a vertex to its neighbors, using its *sendMsg*() and *mergeMsg*() functions.

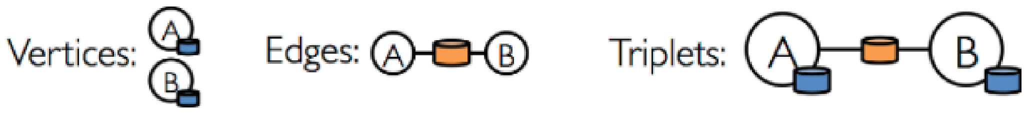


Figure 2.3 Vertices, edges, and triplets in GraphX

However, due to the nature of Spark's data partition and shuffle operations, the performance of GraphX is worse than graphs in MASS [10]. For the performance of incremental graph construction, GraphX performs 50%+ slower than MASS with multiple nodes. Also, GraphX needs to read whole graphs first and then process the whole graphs at once which has the construction overhead.

2.2.2 *Spark Streaming*

Spark Streaming is an extension of Spark that supports the processing of live data streams [11]. As shown in Figure 2.4, it receives an input data stream and divides the data into batches. Spark treats each batch of data as an RDD and processes them using RDD operations like *map*, reduce, *join*, and *window*. Finally, the results of the RDDs are returned in batches. This framework is scalable, high-throughput, and fault-tolerant.

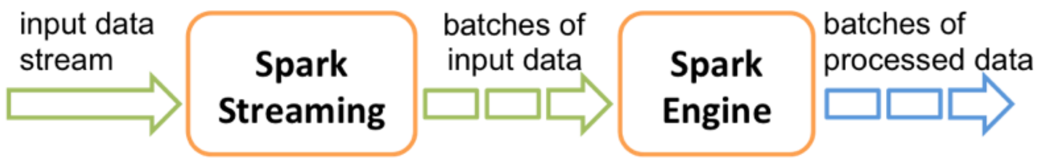


Figure 2.4 Spark Streaming workflow

When using Spark Streaming to process graphs, the graph-related operations and algorithms can be applied to the data streams, so it treats each micro-batch as a graph and generates batches of processed graph data. Although some applications only require performing computations on data streams, for most real-life problems, stream processing comes as a step in a larger application. For graphs, data streams may be subgraphs of the whole graphs. However, Spark Streaming doesn't support dynamically updating a graph as the stream goes [12][13] and constructing one complete graph from the data streams. To process incremental continuous graphs using Spark Streaming, additional work should be done. Some researchers implement data storage on top of Spark Streaming to store graph structures. The data storage can be implemented with RDD, IndexedRDD, or Redis [12]. When a data stream comes, the vertices and edges will be stored in user-implemented storage so that the graph structure can be kept. This is a workable solution but will increase complexity.

In conclusion, Spark Streaming provides support for efficient window operations on unbounded data streams but lacks support for processing large graphs that dynamically update along with the streams.

CHAPTER 3. GRAPH STREAMING SUPPORT IN MASS JAVA

This chapter discusses how we designed the solution to support graph streaming in MASS. The implementation details are also included in this chapter. The implementation consists of three parts: file pre-processing, graph streaming construction, and graph streaming processing.

3.1 DESIGN OVERVIEW

3.1.1 *Workflow*

To support graph streaming in MASS Java, we incorporated two strategies. One is splitting a graph file into several smaller ones. Then, we load and process one subgraph at a time. The other is processing one subgraph and constructing the next subgraph at the same time. As shown in Figure 3.1 below, for an input graph file, we need to sort the file by vertex ID first, then read and construct the graph from the sorted file. The sorted file can help us identify the boundary of each subgraph, and further tell us which vertices have been constructed and which haven't. Therefore, sorting is an important step before actual construction.

Take the graph file in Figure 3.1 as an example, after sorting, the graph file is split into three sections. During the first period, we read the first section of the file and construct the first subgraph. When the first subgraph is ready, we come into the second period to process the first subgraph and start to construct the second subgraph. The processing and construction are carried out at the same time. The followings are in a pipelined fashion. During each period or cycle, the current latest subgraph is being processed and the next subgraph is being constructed simultaneously. In the end, the whole graph is available in the memory and all the subgraphs have been processed.

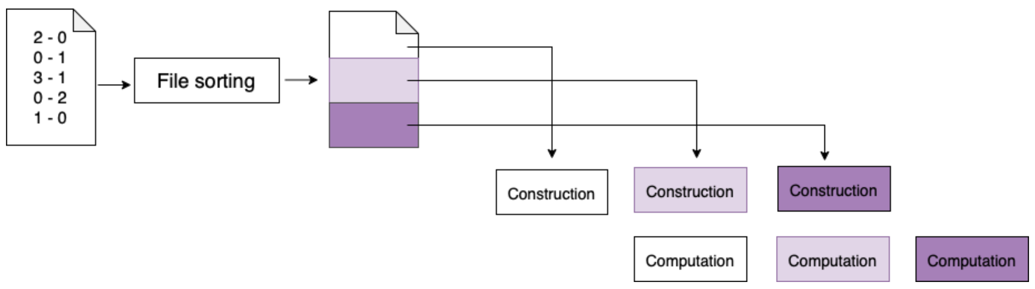


Figure 3.1 Workflow of graph streaming in MASS Java

Figure 3.2 focuses on the construction part. The construction maintains the graph data structure as well as the boundary of each subgraph. When we populate agents on each subgraph, the agents can move along the edges over the graph. This design can not only reduce the construction overhead by overlapping the construction and processing but also prevent any explosive growth of the agent population because the processing range is a much smaller subgraph for each cycle.

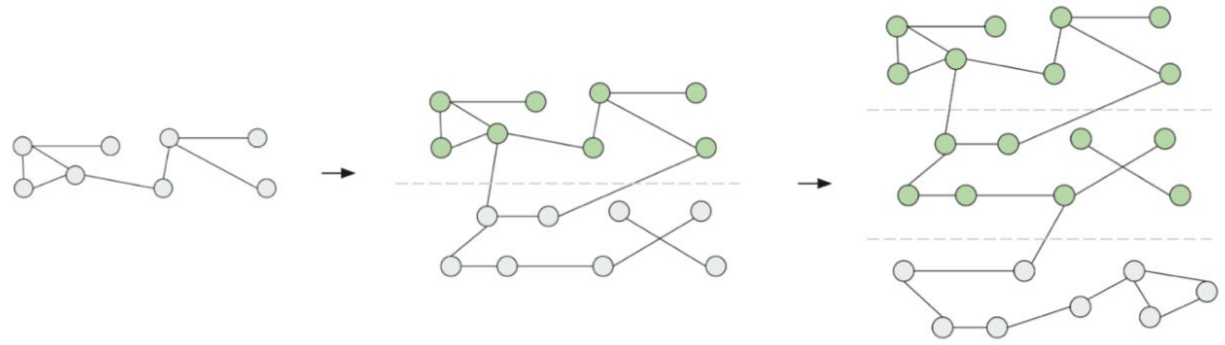


Figure 3.2 Virtual representation of graph streaming construction

3.1.2 *Class Diagram*

To support graph streaming in the MASS Java library, we implemented several new classes within the *graphstreaming* package. Some major classes include HippieStreaming, MatsimStreaming, HippieStreamingVertex, MatsimStreamingVertex, and GraphStreamingAgent as shown in Figure 3.3.

The HippieStreaming class and the MatsimStreaming class inherit MASS base classes Hippie and Matsim respectively. Compared with the Hippie class and the Matsim class, what they mainly do is supporting the simultaneous read and construction operations of a given graph, controlling the loading thread, maintaining a sorted map as the vertices' container, and having other methods to support streaming. The HippieStreamingVertex class and the MatsimStreamingVertex class inherit HippieVertex and MatsimVertex respectively. Those two classes maintain lists of intermediate results data which are important during the processing phase. The GraphStreamingAgent extends from MASS's existing class Agent. The GraphStreamingAgent carries the boundary data and has some supporting methods to handle connections between subgraphs.

Besides those classes, we also implemented the GraphFile class and the FileUtils class in the *graphstreaming* package to handle the input graph files. The GraphFile class integrates the information of file path, file format, and whether it is sorted together. The FileUtils class wraps up the sorting and file splitting scripts.

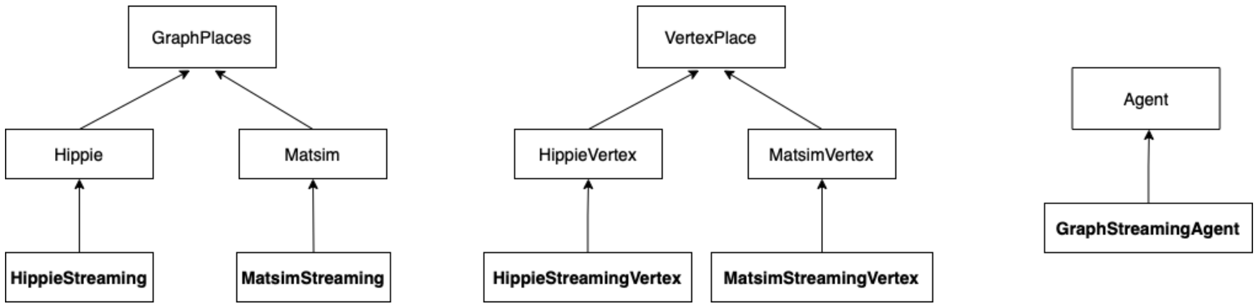


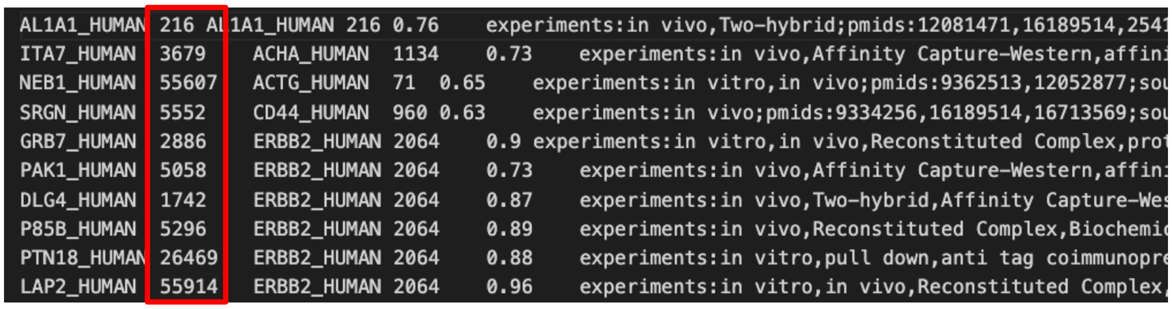
Figure 3.3 Class Diagram of Graph Streaming in MASS Java

3.2 File Pre-processing

The graph streaming in MASS Java supports two file formats, Hippie and Matsim. To better support the graph construction, files need to be sorted by vertex ID.

3.2.1 *Hippie Files*

Graph information in the Hippie format is a list of tab-delimited edges as shown in Figure 3.4 below. The first two columns are a pair of a protein key and an ID which can represent a source vertex of the edge. The next two columns are a pair of an interaction key and an ID which can represent the destination vertex of the edge as well as the neighbor of the source vertex. The followings of each line are the attributes of the edges. To sort a Hippie file, we need to sort by the second column which is the ID of each source vertex as shown in the red rectangle.



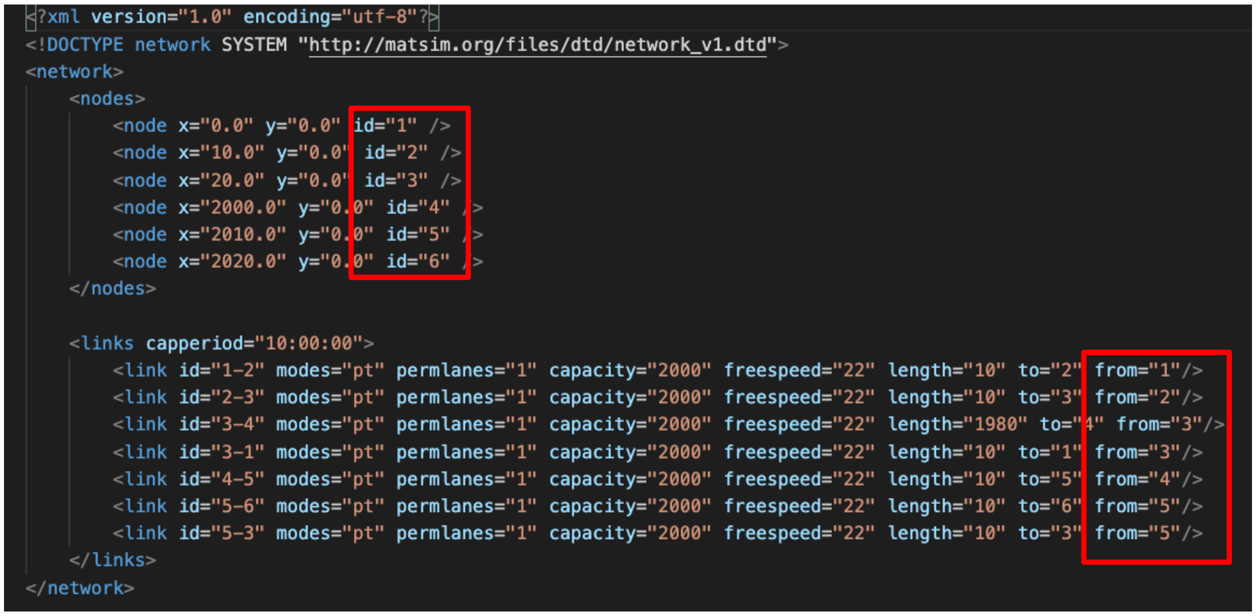
```
AL1A1_HUMAN 216 AL1A1_HUMAN 216 0.76    experiments:in vivo,Two-hybrid;pmids:12081471,16189514,2541
ITA7_HUMAN  3679    ACHA_HUMAN  1134    0.73    experiments:in vivo,Affinity Capture-Western,affini
NEB1_HUMAN  55607   ACTG_HUMAN  71  0.65    experiments:in vitro,in vivo;pmids:9362513,12052877;sou
SRGN_HUMAN  5552    CD44_HUMAN  960 0.63    experiments:in vivo;pmids:9334256,16189514,16713569;sou
GRB7_HUMAN  2886    ERBB2_HUMAN 2064    0.9 experiments:in vitro,in vivo,Reconstituted Complex,prot
PAK1_HUMAN  5058    ERBB2_HUMAN 2064    0.73    experiments:in vivo,Affinity Capture-Western,affini
DLG4_HUMAN  1742    ERBB2_HUMAN 2064    0.87    experiments:in vivo,Two-hybrid,Affinity Capture-Wes
P85B_HUMAN  5296    ERBB2_HUMAN 2064    0.89    experiments:in vivo,Reconstituted Complex,Biochemic
PTN18_HUMAN 26469   ERBB2_HUMAN 2064    0.88    experiments:in vitro,pull down,anti tag coimmunopre
LAP2_HUMAN  55914   ERBB2_HUMAN 2064    0.96    experiments:in vitro,in vivo,Reconstituted Complex,
```

Figure 3.4 Excerpt from a Hippie file

3.2.2 *Matsim Files*

The MATSim files include two sections: nodes, and links as shown in Figure 3.5 below. The nodes section includes all the information related to vertices. The links section can also be seen as a list of edges. The **from** attribute includes a source vertex ID, the **to** attribute is a destination vertex ID, and other attributes are related to the edge.

To sort MATSim files and make the later construction easier, a Matsim file needs to be split into two new files. One is a nodes file that only includes the nodes section, and the other is a links file that includes the links section. Then we sort those two files separately. The nodes file is sorted by the **id** attribute and the links file is sorted by the **from** attribute. This preprocess results into two sorted files.



```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE network SYSTEM "http://matsim.org/files/dtd/network_v1.dtd">
<network>
    <nodes>
        <node x="0.0" y="0.0" id="1" />
        <node x="10.0" y="0.0" id="2" />
        <node x="20.0" y="0.0" id="3" />
        <node x="2000.0" y="0.0" id="4" />
        <node x="2010.0" y="0.0" id="5" />
        <node x="2020.0" y="0.0" id="6" />
    </nodes>

    <links capperiod="10:00:00">
        <link id="1-2" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="2" from="1"/>
        <link id="2-3" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="3" from="2"/>
        <link id="3-4" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="1980" to="4" from="3"/>
        <link id="3-1" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="1" from="3"/>
        <link id="4-5" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="5" from="4"/>
        <link id="5-6" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="6" from="5"/>
        <link id="5-3" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="3" from="5"/>
    </links>
</network>
```

Figure 3.5 Excerpt from a MATSim file

3.2.3 *Sorting Solution*

We use the GNU parallel and GNU Sort tools. GNU parallel [7] is a shell tool for executing jobs in parallel. GNU sort is a Linux command that is bundled in GNU Coreutils. It can sort files by line numerically or alphabetically. The implementation of the GNU sort employs the merge sort algorithm. Since we are targeting to process huge files, GNU parallel tool is needed to run sort so that the sorting time can be significantly reduced. The command to achieve this is *parsort.* Below are some experimental results of sorting files by using the *parsort* command. We can see that the parallel sort can help us sort a huge file in a reasonable time.

Table 3.1 File sorting experiments results by using the parsort command

File Size	**Number of Lines**	**Time (mins)**
2.5 GB	120 million	1 min
26 GB	1400 million	34 mins
40 GB	2100 million	87 mins

3.3 Graph Streaming Construction

3.3.1 *Dedicated thread for construction*

One important strategy we used in this project is processing the current subgraph and constructing the next subgraph at the same time. To achieve this, we created a child thread called the loading thread that is dedicated to graph streaming construction. By doing this, the main thread can execute the main function and perform computation on the current subgraph while the loading thread is constructing the next subgraph simultaneously.

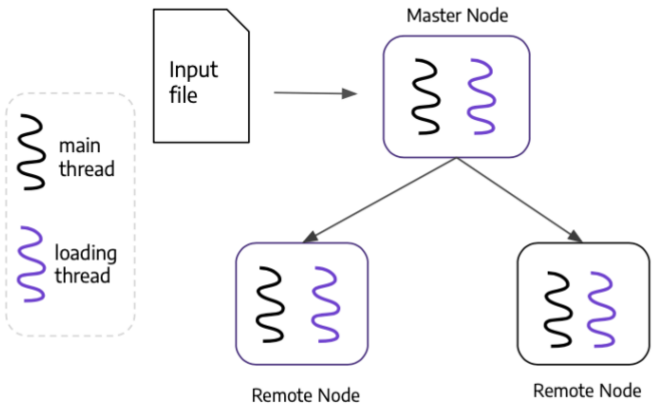


Figure 3.6 loading thread on the master node and remote nodes

As shown in Figure 3.6 above, both the master nodes and the remote nodes have those two threads. The loading thread on the master node is responsible for reading from an input file. It also controls where to resume and suspend reading the file for each cycle. After reading, this thread parses the data and constructs vertices and edges either on its own node or sends the corresponding request to remote nodes. Each vertex of the graph is represented by a HippieStreamingVertex object or a MatsimStreamingVertex object. Each vertex is allocated to a different computing node, (which is called the owner) by its global index, which distributes vertices in a round-robin fashion.

To reduce the number of messages that need to be sent to the remote nodes for vertices and edges construction, the network data will be cached on the master node. When a given cached limit is reached, a bulk of network data will be sent to the corresponding remote nodes. This strategy matches the Hippie and Matsim object constructions.

On each remote node, upon a MASS process launches, the main thread is always waiting for new messages from the master node. Besides the conventional MASS messages to manage Places and Agents, we created several new message types which contain the word "streaming", all used by the main and loading threads. Those message types are specifically for graph streaming construction including "*GRAPH_HIPPIE_STREAMING_OPS", "GRAPH_MATSIM_STREAMING_OPS", and "GRAPH_MATSIM_UPDATEV_STREAMING_OPS"*.

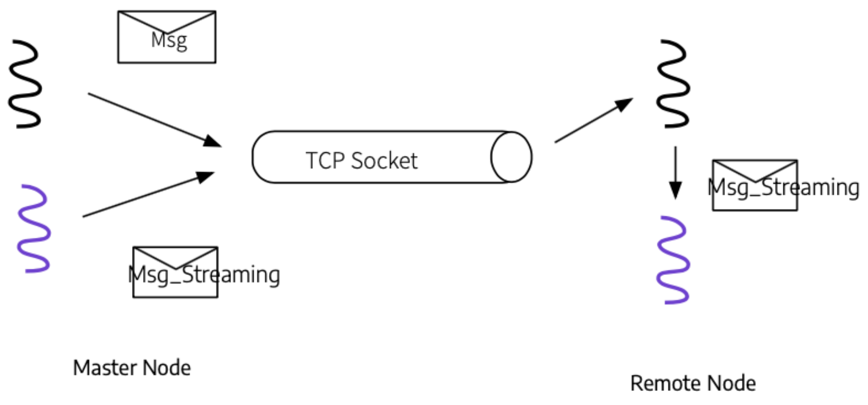


Figure 3.7 Messages sent from the master node to the remote node

As shown in Figure 3.7, since the loading thread takes charge of only graph streaming constructions, it always sends messages marked with "Streaming". While the main thread just sends out normal messages related to the computation and agents, for example, *"callAll"*, *"manageAll"*, etc. On the remote nodes, when the main thread receives messages from the master node, it will check the message type, first. If it is a "Streaming"-marked message, the main thread will create a loading thread to handle the message and thereafter continues to wait for the next message. Otherwise, it handles the message by itself. In this way, we can make sure the loading thread and the main thread on remote nodes can work together and won't interfere with each other.

3.3.2 *Message Distribution among threads*

On remote nodes, when the loading thread and main thread work on different tasks at the same time, both need to send the *Ack* message back to the master node after the tasks are finished. On the master node, the loading thread and the main thread are both waiting for the ack message from the remote node. In such a case, we need to assure that the threads on the master node can receive their corresponding messages. This in turn means that the main thread should receive *Ack* sent from the main thread of a remote node and the loading thread should receive *Ack* sent from a remote loading thread.

To achieve this, as shown in Figure 3.8, we created a new message type called *Ack_Streaming*. Whenever the loading thread finishes its work, it sends a *Ack_Streaming* message back to the master node, whereas the main thread sends back an *Ack* message. Besides that, on the master node, a listener thread is also implemented which is dedicated to listening on given TCP ports and receiving messages sent from all the remote nodes. After receiving the messages, the listener thread checks their action type and then deliver them to the corresponding thread, based on their action type. *Ack_Streaming* messages will be sent to the loading thread and the *Ack* messages will be sent to the main thread.

The communication between threads on the master node is implemented with *BlockingQueue*. Each communication channel between the master node and a remote node maintains two Blocking Queue objects. One is for *Ack_Streaming* messages and the other is for *Ack* messages. The listener thread puts messages into the corresponding queue. The main and the loading thread fetches messages from the *Ack* and *Ack_Streaming* message queue, respectively.

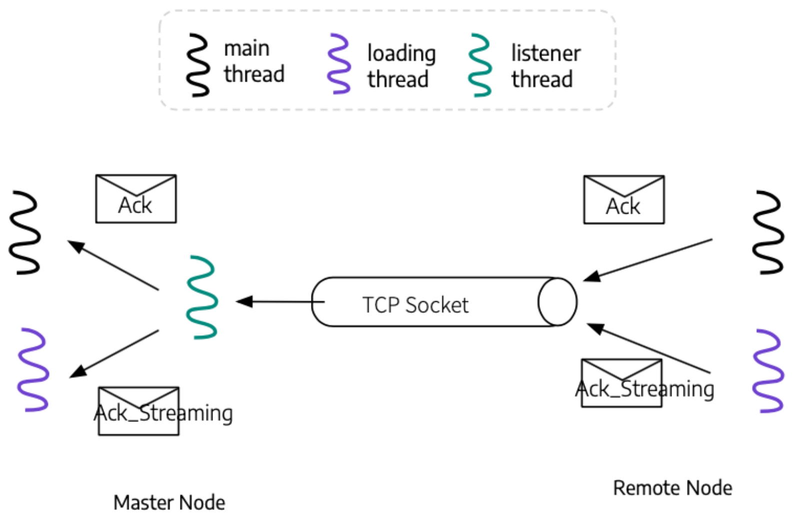
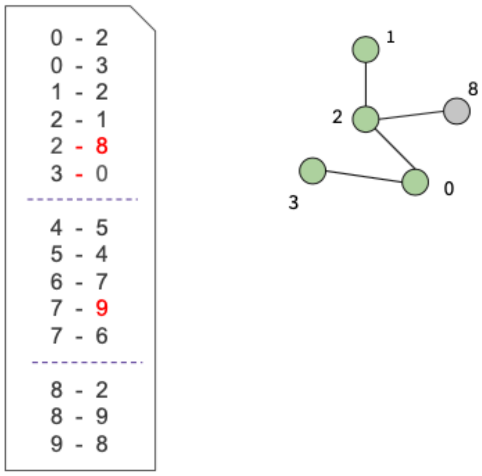


Figure 3.8 Listener thread distributes messages

3.3.3 *Loading thread on the master node*

The loading thread on the master node reads a portion of an input file from top to down line by line. Each line is parsed to construct a source vertex, its outgoing edge, and the neighbor into a distributed graph. The number of lines to read in each cycle depends on the chunk size which is the number of vertices we would like to process per cycle. Users can specify the chunk size based on their needs. When the loading thread reads a file, it will count the number of source vertices that have been constructed on the fly. All the outgoing edges and neighbors starting from a same source vertex will be constructed along with the source vertex so that they won't be split into different cycles. When the count equals the chunk size and all the edges and neighbors related to the last vertex have been constructed, the loading thread will stop for the current cycle. A file pointer is maintained so that in the next cycle, the file can be read from where it stops in the previous cycle. Figure 3.9 shows a simplified Hippie file which has been sorted by the source vertices. If we set the chunk size to four, the whole file will be fully loaded in three cycles. The first cycle has vertex 0, 1, 2, and 3. The second cycle has vertex 4, 5, 6, and 7. The lase cycle has vertex 8 and 9. Next to the file, it's the first subgraph we constructed from the file.



```
0 - 2
0 - 3
1 - 2
2 - 1
2 - 8
3 - 0
-----------
4 - 5
5 - 4
6 - 7
7 - 9
7 - 6
-----------
8 - 2
8 - 9
9 - 8
```

Figure 3.9 Example of graph streaming construction

The default chunk size is 4000. It will be used if users don't specify a number for the chunk size. The main consideration in determining this default chunk size is the waiting time for the first subgraph's construction before the actual computation. After some experiments, we found that 10 seconds is a reasonable time, and about 4000 vertices can be constructed within that time.

Although the example file in Figure 3.9 is an undirected and unweighted graph, graph streaming supports directed, undirected, weighted, and unweighted graphs thanks to how we construct graphs and store related information. For undirected graphs, each edge will be stored as an outgoing edge in both vertices, while directed graphs store an edge on the source vertex of the edge. Along with each edge, other available information about the edge, like weight will also be stored on the vertices. Since all the information is stored on each vertex, when processing the graph, the graph format doesn't matter to the agents because they can just do computation and make decisions based on the information on the vertices.

There are two pairs of concepts related to graph streaming construction, (1) the lower boundary vs. the upper boundary and (2) complete vertices vs. incomplete vertices.

Lower boundary and upper boundary: When the loading thread constructs a subgraph, it maintains that subgraph's lower and upper boundary. The lower boundary is related to the smallest source vertex ID it reads, and the upper boundary is the biggest source vertex ID. In each cycle, we can get the lower boundary and the upper boundary of the current subgraph that we are going to process. Since in the memory, we maintain a cumulative graph constructed from the very beginning, those boundary data can help us locate the latest portion of the graph that is ready to be processed. The vertices whose ID are within the boundary range are the ones to be processed in the current cycle. The lower boundary is excluded in the range, and the upper boundary is included. For example, in Figure 3.8, the lower boundary and upper boundary of the first subgraph are MIN_VALUE and 3, the second subgraph's boundary is 3 to 7, and the third one is 7 to MAX_VALUE.

To make it more efficient to locate the latest subgraph with giving boundary data, we created a sorted map as another vertex container on each computing node. The sorted map container is an addition to the existing single vector vertices container, which inherits from the GraphPlaces class. The key of the sorted map is vertex ID, and the value of the map is the reference of the corresponding vertex. When constructing a vertex, the vertex will be added to the vector, and its reference will also be put into the sorted map. By doing this, with the lower boundary and upper boundary, we can quickly get a submap within the range. All the values of the submap are the references of the vertices that need to be processed. The sorted map is implemented with *ConcurrentSkipListMap*, which is thread-safe.

Complete vertices and incomplete vertices: Complete vertices are the vertices that all the information about them is available in the memory, including the info related to the vertices, their outgoing edges, and neighbors. In Figure 3.8, when the first subgraph is ready to be processed, the vertex 0 – 3 are complete vertices because we know all their information. The vertex 8 is incomplete because its outgoing edges and neighbors' information are in the file's third section, which hasn't been constructed in the memory. For the same reason, when we process the second subgraph, vertex 9 is incomplete. It's easy to infer whether a vertex is complete by the numerical relationship between vertex ID and the upper boundary. If the ID of a vertex is equal to or smaller than the upper boundary, the vertex is complete. Otherwise, it is an incomplete vertex. When processing incomplete vertices, they need to be handled specially, and we will discuss it in detail in section 3.4.

For MATSim files, the reading and construction logic is similar to Hippie files. First, the loading thread will read the sorted nodes file from top to down and construct vertices. When we stop reading, we can get the subgraph's lower and upper boundary. Then the loading thread will read the sorted links file and construct edges based on the boundary data. Two file pointers are maintained, one for the nodes file and the other for the links file.

3.4 GRAPH STREAMING PROCESSING

3.4.1 *Agent Initialization*

Some graph applications in MASS (such as triangle counting and connected components) populate agents on each of the vertices and then let them start the computation. For graph streaming, since we process a portion of the graph at a time, we only need to populate agents on that portion of the graph instead of the whole graph. Also, the agents should know the boundary so that when they migrate, they can know whether they are out of the range.

To achieve those goals, we implemented a new class called GraphStreamingAgent. The GraphStreamingAgent class inherits the MASS base Agent class. On top of the Agent class, the GraphStreamingAgent class has two more properties: the lower boundary and the upper boundary of the current subgraph that is being processed. By keeping that, whenever agents migrate, they can know whether they reached out of the range or not. When users define their own agents, the agents should extend the GraphStreamingAgent class.

To initialize agents only on a portion of the graph, a new Agents constructor has been implemented. The lower and the upper boundaries of the current subgraph are the parameters needed for the constructor. The constructor locates the latest subgraph according to the boundary data and creates agents on each of the vertices within the subgraph. It also passes the boundary data to each of the agents. Then they can migrate and compute, based on the user-defined logic. The Agents constructor should be called in each cycle after we get the graph instance and boundary data.

3.4.2 *Computation Suspension*

After the initialization of agents, they can do computation and migration. A lot of applications rely on the migration of agents. During migration, the agents may encounter a situation where they reach an incomplete vertex. For example, in Figure 3.8, during the first processing cycle, we first initialize agents on vertex 0, 1, 2, and 3. When the agent moves from vertex 2 to vertex 8, it finds that vertex 8 is incomplete. In such a case, the computation should be suspended at vertex 8 because we do not know its neighbors at this time and thus, we cannot make the decision about where to move that agent. The function currentPlaceBeyondBoundary() of the GraphStreamingAgent class tells whether an agent is on an incomplete vertex.

To suspend the computation, we also implemented a function called *suspendComputation*() in the GraphStreamingAgent class. When an agent calls this function, it will put the intermediate results it carries on the current vertex and kill itself. We have implemented a storage list on the HippieStreamingVertex and the MatsimStreamingVertex class to store intermediate results in this class of vertex. This function depends on a user application and thus does not have to be always called.

3.4.3 *Computation Resumption*

If computation has been suspended at some vertices, it should be resumed later. The time to resume the computation is when the vertices are within the processing range. Figure 3.8 shows such an example that computation stopped at vertex 8 and later should be resumed in the third cycle as vertex 8 has become a complete vertex in that cycle and it is within the processing range of 7 to MAX_VALUE.

To resume computation, if the previous agents only update and deposit their data on the vertex, the next cycle can initialize agents with deposited and let them continue the computation. If the previous agents have put intermediate data on the storage of the vertex, after the initialization of agents on the latest subgraph, extra agents should be created to take the data in the storage and continue the computation. We have implemented *resumeAgents*() for this purpose. For each cycle, when this method is called, each vertex within the current processing range will be checked to see whether there are intermediate results stored in its storage. Then according to the number of intermediate results on each vertex, a corresponding number of agents will be created to carry the intermediate data and continue the computation along with all other agents. In this way, the computation can be continued. As shown in Figure 3.9 below, if there are four intermediate results on the vertex, then four more agents should be created, and each carries an intermediate data.

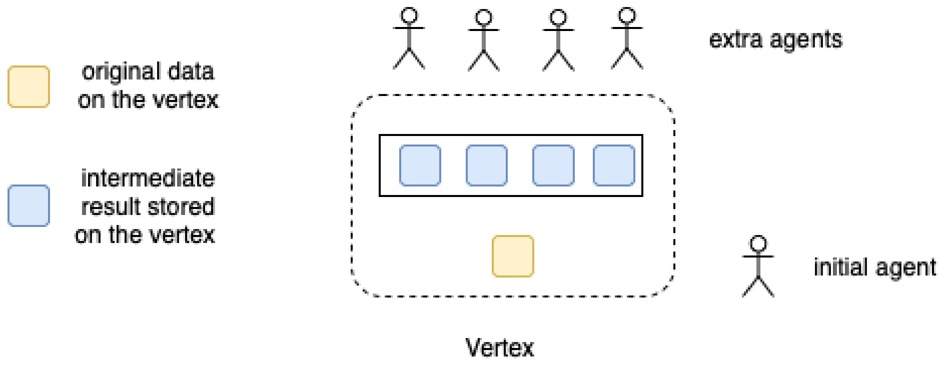

original data on the vertex
intermediate result stored on the vertex
extra agents
initial agent
Vertex

Figure 3.10 Diagram of resumeAgents()

3.4.4 *Invoking methods on part of the Graph*

In the current MASS library, we have the *callAll*() method for GraphPlaces. It can invoke a function on all the vertices of a graph. To align with the design of graph streaming that processes a portion of the graph at a time, we implemented a new *callAll*() method with the lower boundary and upper boundary as the method parameters to invoke a function only on a subgraph. By calling this method, a subgraph will be located according to the boundary data. Then the function will be executed on each of the vertices within the subgraph instead of the whole graph.

3.5 USER-LEVEL CODING FRAMEWORK

When users use graph streaming within the MASS library, they don’t need to understand all the implementation details. Here are some important methods that are exposed to the users. List 3.1 shows a coding framework.

hasNext(): This method returns a Boolean value to indicate whether there is still part of the graph that hasn’t been processed. This method can be put into a while loop to keep checking if we need to continue processing after each cycle.

next(): This method returns a graph instance that incorporates the latest subgraph. This returned graph instance is a cumulative graph from the very beginning. In the meantime, the loading thread will start to construct the next portion of the graph in the background. This method should be called when the hasNext() returns true.

get_lowerBoundary() & get_upperBoundary(): Those methods return the lower boundary and the upper boundary of the latest subgraph that is ready to be processed in the current cycle. By using those boundary data, the latest subgraph can be located in the returned graph instance.

getNeighborsLabelID(): This is a method of the HippieStreamingVertex/MatsimStreamingVertrx class. It returns an array of vertex IDs of the vertex's neighbors.

syncOneCyle(): This method will sync the main thread and the loading thread at the end of each cycle. To make sure both the computation and construction have finished. It should be called at the end of each cycle.

close(): This method will stop the listener thread and clean up any intermediate files. It should be called before the MASS.finish().

Listing 3.1 Coding framework of graph streaming

```
1   MASS.init();
2   HippieStreaming hstreaming = new HippieStreaming(…);

3   While (hstreaming.hasNext()) {

4           Hippie hippie = hstreaming.next();

5           lowerBoundary =  hstreaming. get_lowerBoundary();
6           upperBoundary = hstreaming. get_upperBoudnary();

7           Agents agents = new Agents(lowerBoundary, upperBoundary);

8         //let the agents process the subgraph
9           ……..

10          hstreaming.syncOneCycle();
11   }

12   hstreaming.close();
13   MASS.finish();
```

As shown in List 3.1 above, on line 7, we created agents only on the latest portion of the graph. Then we can let the agents start to process the latest portion of the graph.

CHAPTER 4. PERFORMANCE EVALUATION

This chapter evaluates the performance of our graph streaming solution. The evaluation was conducted with two classic graphs applications: Triangle Counting and Connected Components. Both applications were executed with 1 – 24 computing nodes.

4.1 ENVIRONMENT SETUP

The applications were executed on the HERMES cluster and the CSSMPI cluster at the University of Washington Bothell. Those two clusters have 24 computing nodes in total. The detailed information about the computing nodes is shown in Table 4.1 below. When executing the applications, the first 12 computing nodes are from the HERMES cluster, and the rest 12 computing nodes are from the CSSMPI cluster.

Table 4.1 Execution environments of HERMES and CSSPMI clusters

# Computing Nodes	**# Logical CPU Cores**	**CPU Model**	**Memory**	**Cluster**
3	4	Intel Xeon 5150 @ 2.66 GHz	16 GB	HERMES
4	8	Intel Xeon E5410 @ 2.33 GHz	16 GB	HERMES
5	4	Intel Xeon Gold 5220R @ 2.20 GHz	16 GB	HERMES
12	4	Intel Xeon Gold 6130 @ 2.10 GHz	16 GB	CSSMPI

For this project, we focused on evaluating the performance of HippieStreaming with Hippie graph files because Hippie files are easier to generate. In addition, the implementation of MatsimStreaming is very similar to HippieStreaming. To generate Hippie files, we modified a random graph generator application called GraphGen.java in the DSL lab. This modified application can generate graph files that strictly follow the HIPPIE format with the number of vertices as an input parameter.

4.2 TRIANGLE COUNTING

4.2.1 *Algorithm and implementation*

Triangle Counting is to find the number of triangles that exist in a graph. In MASS, Triangle Counting can be solved by letting each agent migrate through a series of three steps. At first, one agent is created on one vertex. Steps 1-2: each agent propagates itself to all neighbor places with a smaller vertex ID than the current place (spawning children if more than one neighbor fits this criterion). If there is no neighbor available, the agent should be killed. Step 3: all remaining agents attempt to return to their source. If the agent can return to the source, it discovered a triangle.

This algorithm can be applied to graph streaming easily. For graph streaming, we simply run this algorithm repeatedly on each subgraph and then add up the result from each subgraph. As shown in Figure 4.1 below, after each subgraph is ready, we create an agent on each vertex of the subgraph, and the agents will move to the neighbors with smaller IDs or move back to the source. Since the graph is constructed in order of vertex ID from small to big, the vertices with smaller IDs must already be in the memory and complete for any vertices of the current subgraph. Therefore, this problem won't involve computation suspension and resumption.

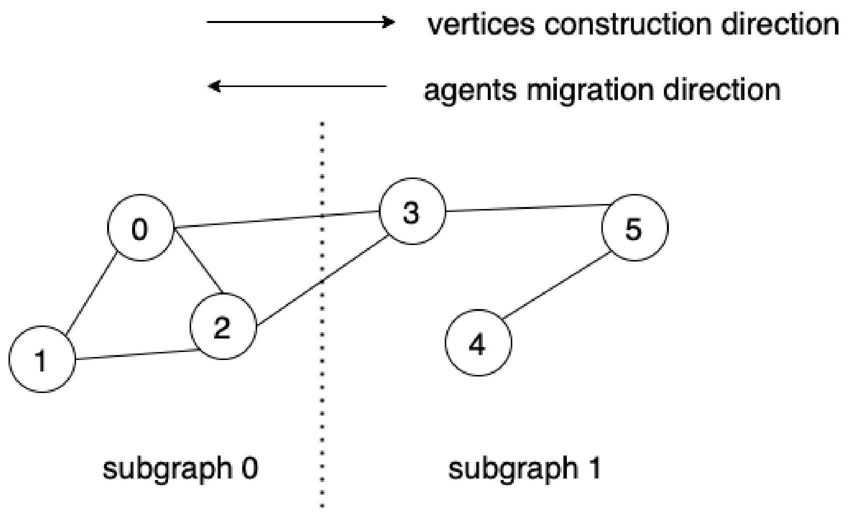


Figure 4.1 Diagram of Triangle Counting on graph streaming

4.2.2 *Performance of Parallel Computing*

We generated a hippie file to verify the Triangle Counting. The hippie file is 290 MB with 40,000 vertices. The average degree of each vertex is 50. We did three rounds of the tests with 1 - 24 computing nodes. In each round, the number of vertices per cycle was set to 2000, 4000, and 8000. For the tests of each setup, we measured three times and used their average execution time to evaluate the performance.

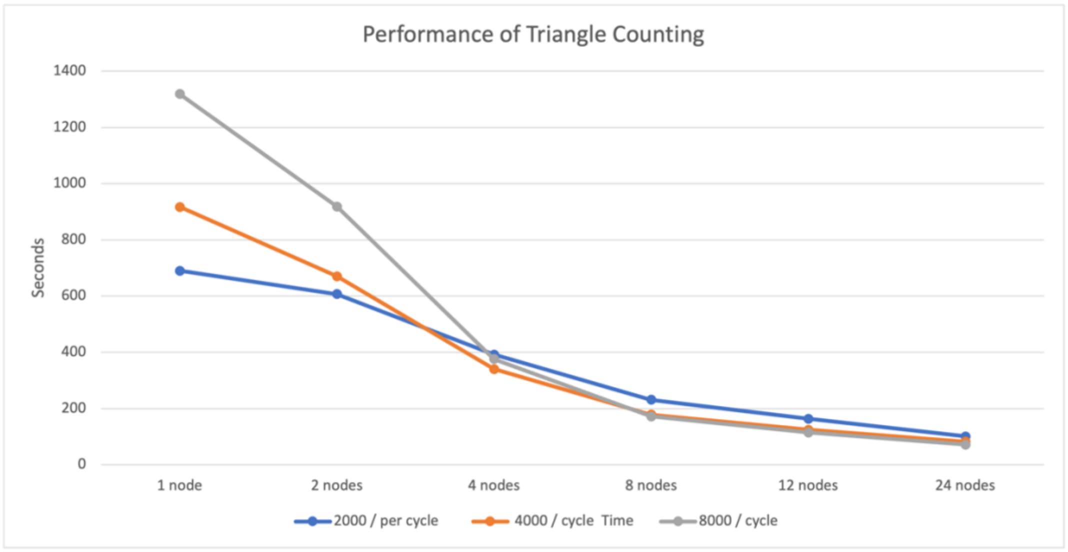


Figure 4.2 Execution result of Triangle Counting on HippieStreaming

As shown in Figure 4.2, from the trend, we can see that with 24 computing nodes, the performance improves significantly compared with a single node, no matter how many vertices are processed in each cycle. The performance on 1 node and 2 nodes indicates that the smaller the subgraph per cycle, the better the performance. The reason is that, with 1 or 2 computing nodes, smaller subgraphs lead to less vertices and agents allocated on each node which can maximize the resource efficiency. With more than 4 computing nodes, a smaller distributed subgraph can result in even less agents and vertices per each node. The optimization brought by less agents cannot compensate the communication overheads. Therefore, slightly bigger subgraphs can have better performance for more computing nodes. This result tells us that using multiple nodes can significantly improve the performance. The optimal chunk size may be different with a different number of computing nodes.

4.2.3 *Performance Comparison of Streaming and non-Streaming*

We also measured the performance of the non-streaming solution to run Triangle Counting with the same input file. The chunk size is set to 40,000 so that the program reads an entire file, complete a graph construction, and then processes the whole graph at once. Table 4.2 shows the result. The application cannot be executed successfully until we have 4 computing nodes. With 1 and 2 nodes, we received an out-of-memory error. As previously shown in Figure 4.1, the graph streaming solution can run successfully on 1 or 2 nodes. This result demonstrates that our graph streaming solution serves as a useful mechanism when the memory is not enough to process a huge graph.

Besides that, with 4 computing nodes, the execution time is 890 seconds which is longer than the streaming solution. This result can also prove that the graph streaming solution can efficiently use a small memory space, thus improving performance.

Table 4.2 Execution result of Triangle Counting with non-streaming

#Computing nodes	Execution time (seconds)
1	Out of Memory
2	Out of Memory
4	890.422

4.3 Connected Components

4.3.1 *Algorithm and Implementation*

A connected component represents "a maximal set of vertices such that there is a connection between every pair of vertices". The components are separate "pieces" of a graph such that there is no connection between the pieces. Chang Liu has implemented a MASS program to find the number of connected components in a graph [14]. The algorithm repeatedly spreads the smallest vertex ID in a component to its neighbors until all the vertices in the component get the same ID. It then collects all the component IDs from each vertex.

For graph streaming, as shown in Figure 4.3, after the subgraph 0 is ready, we start to process subgraph 0 and construct subgraph 1 at the same time. The dotted lines in subgraph 1 mean that those vertices and edges are being constructed simultaneously with subgraph 0's computation. Note that, in the construction process, they may not be in the memory yet. When applying this algorithm to graph steaming, we need to modify it a little bit to handle connections between different subgraphs, which includes checking boundaries and only collecting the finalized component IDs.

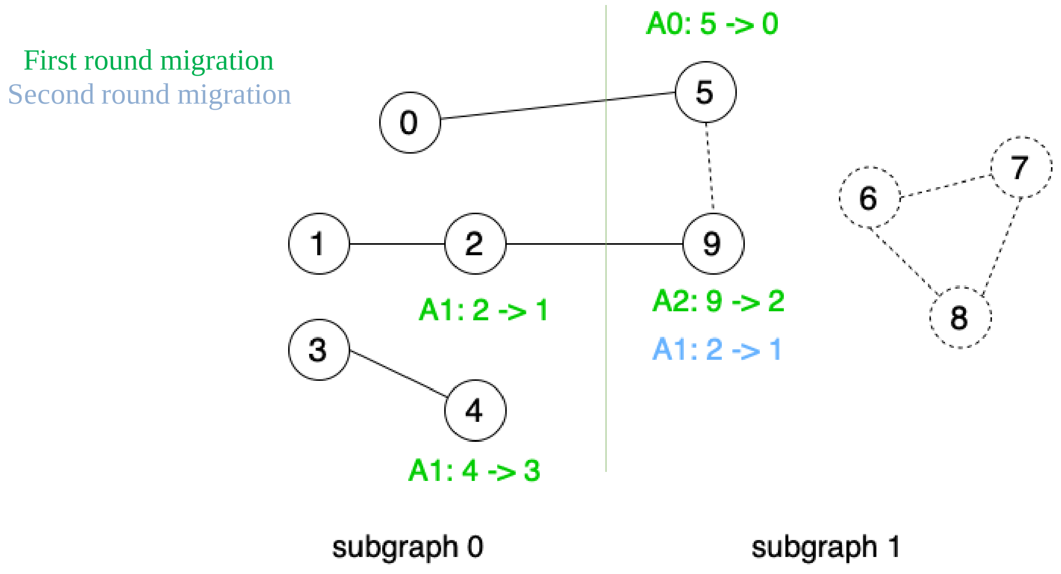


Figure 4.3 Diagram of Connected Components on graph streaming

The algorithm is as follows. When processing the subgraph 0, we populate agents on vertex 0 - 4. Each agent takes two pieces of information: an origin ID and a component ID. The origin ID is the place's vertex ID from which the agent is initially created. The component ID is the value that needs to be spread, initialized as its place's component ID. Each place maintains a component ID that it currently belongs to. In the beginning, each place's component ID equals its vertex ID. After initializing agents, agents migrate to neighbors with smaller component IDs than theirs.

Check boundary: Each agent should check if the current place is beyond the upper boundary after each migration. If yes, the agent should be killed (i.e., computation suspended), and the agent's origin ID should be returned. The origin ID indicates which vertex connects to future vertices. In Figure 4.3, when agent 0 migrates to vertex 5, it updates the component ID of vertex 5 to 0 and kills itself because vertex 5 is beyond the current processing range (0 - 4). The origin ID 0 of agent 0 is returned, which tells us that vertex 0 connects to an incomplete vertex. The same is for agent 1 and agent 2. When those two agents migrate to vertex 9, they also need to be killed and return origin IDs 1 and 2.

Collect finalized component ID: When there is no agent left, the subgraph 0 has been processed, and we need to collect the component IDs of vertices within subgraph 0. The vertex 0, 1, and 2 should be excluded because they connect to future incomplete vertices so that their current component IDs may not be finalized. In subgraph 0, we got 1 component with ID 3.

Then we can repeatedly apply the algorithm to the next subgraph. For example, when processing subgraph 1, the initial agents on vertex 5 and vertex 9 have origin IDs as 5 and 9, and the component ID as 0 and 1. In this way, the components IDs 0 and 1 can continue to be propagated. Eventually, the component with ID 0 is collected in the second cycle.

4.3.2 *Performance of Parallel Computing*

To evaluate the performance of Connected Components on graph streaming, we generated a Hippie file using the generator introduced before. The file is 140 MB with 25,000 vertices. The average degree of each vertex is 50. As the Triangle Counting, we also ran three rounds of the experiments. In each round, the chunk size of each cycle is 2000, 4000, and 6000. The experiments were conducted with a single computing node up to 24 computing nodes. Each test with a different setup was executed three times, and each datapoint shown in Figure 4,4 below is the average time of three execution results.

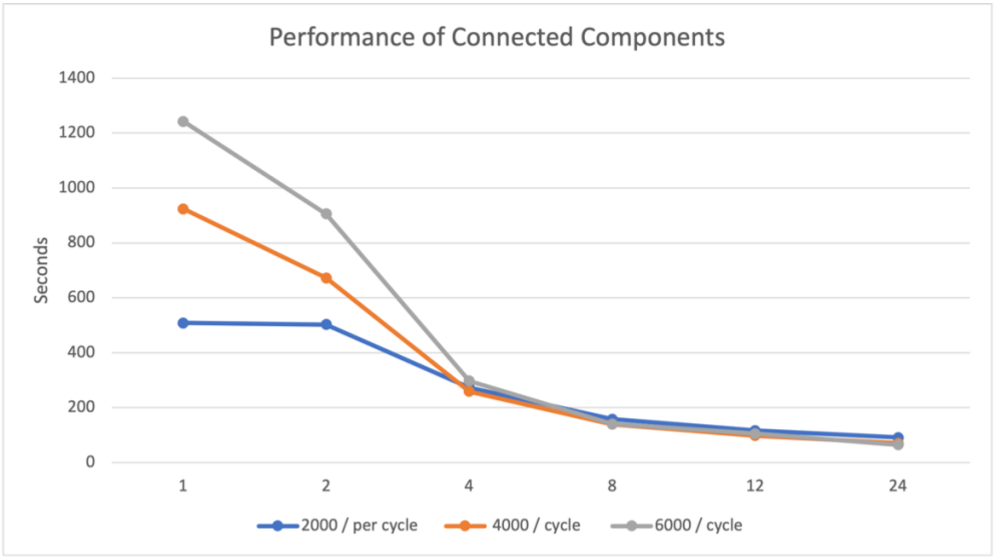


Figure 4.4 Execution result of Connected Components on HippieStreaming

The above chart demonstrates significant CPU-scalable and memory-scalable performance improvements. No matter how many vertices we processed in one cycle, the improvement trend is quite typical. On average, the performance on 24 computing nodes improved 10 times compared with a single node. Besides that, we also found the same trend as the Triangle Counting application. In this case, with 1 – 4 computing nodes, the smaller each subgraph, the better the performance. When there are more than 4 computing nodes, bigger subgraphs can perform better.

4.3.3 *Performance Comparison of Streaming and non-Streaming*

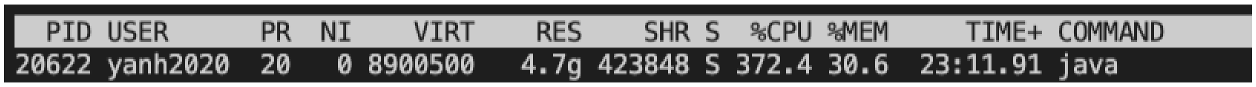
We also tested this application with non-streaming on a single node. We did this by setting the chunk size to 25,000, which is equal to the whole graph size. In this way, the entire graph was constructed first, then processed at once without streaming. By doing this, we would like to find out that with limited resources (single node), whether the graph streaming solution can perform better. The performance results are summarized in Table 4.3. Those experimental results indicate that, on a single node, the performance of the graph streaming solution is much better than the non-streaming solution.

One reason that affects the performance is the number of alive agents that remain in the system and wait to be processed. When processing a graph, first, we need to create an agent on each vertex. For the non-streaming graph, the initial agent population is 25,000. Before each migration, agents may need to spawn child agents, leading to much more agents. When calling *migrate(), callAll()* and *manageAll(),* all those agents should be processed. Multiple threads check-in and out each agent one by one and process their requests, so processing a bigger bag of agents takes longer-time. Also, the application needs multiple rounds of migration to finish the processing. For graph streaming with chunk size 2000, in each cycle, the number of agents is 2000. With the same number of threads, processing 2000 agents took much less time than 25,000. And because the subgraph is smaller, it needs less rounds to finish, which can also reduce the total time. Therefore, repetitively processing a smaller graph several times can perform better than processing a big graph all at once. From the execution time, we found out that repetitively processing 2000 initial agents 13 times (13 cycles in total when the chunk size is 2000) is about 7.5 times better than processing 25,000 initial agents all at once.

Table 4.3 Execution of Connected Components

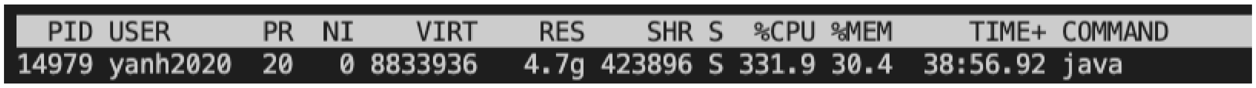
# Computing nodes	**Streaming**	**Execution time (seconds)**	**Performance improvement**
1	No streaming	4085.26	1.00
1	Streaming with chunk size 6000	1213.205	3.37 times
1	Streaming with chunk size 4000	902.164	4.53 times
1	Streaming with chunk size 2000	532.133	7.68 times

To find out other potential reasons why the performances have such a difference, we also checked the memory usage when running this application with streaming and non-streaming. As shown in Figure 4.5 and 4.6, the virtual memory usage for both solutions are about 8.8 GB, and the resident memory usage for both are 4.7 GB. The same memory usage indicates that for those two solutions, the time spent on memory swapping makes little difference in their performance.



```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
20622 yanh2020  20   0 8900500   4.7g 423848 S 372.4 30.6  23:11.91 java
```

Figure 4.5 Memory usage of the non-streaming solution



```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
14979 yanh2020  20   0 8833936   4.7g 423896 S 331.9 30.4  38:56.92 java
```

Figure 4.6 Memory usage of graph streaming solution

4.4 SUMMARY OF EVALUATION

This chapter presents two graph streaming benchmarks to verify and evaluate the performance of the graph streaming implementation. The execution of Triangle Counting and Connected Components with HippieStreaming verified our implementation of graph streaming. The result obtained from the graph streaming solution is the same as the non-streaming solution. Our performance evaluation demonstrated the following advantages of graph streaming.

1) Parallel computing with multiple nodes can significantly improve performance. This can be seen from both the applications.

2) When memory is not enough to process the whole graph, graph streaming can help solve the problem. This can be backed up by the Triangle Counting application. With 1 and 2 computing nodes, the non-streaming solution caused an out-of-memory error while the graph streaming could run successfully.
3) With limited resources, graph streaming can improve performance compared with non-streaming. This observation stems from both applications. For the Triangle Counting, the performance of the graph streaming solution is 2.5 times better than non-streaming with 4 computing nodes. For Connected Components, the graph streaming's performance is 7.5 times better with a single node.

In this project, we only tested our implementation on undirected and unweighted graphs because we used existing graph benchmark programs. But due to the way we construct graphs that for each vertex, all its outgoing edges, edge weights, and neighbors are stored at its vertex place, our implementation can support directed, undirected, weighted, and unweighted graphs. In the future, we should test and verify our implementation on all those graphs.

When evaluating the performance, we didn't compare it with Spark Streaming. The reason is that extra data storage is needed to be implemented on top of Spark Streaming to maintain the graph structures. Some researchers proposed several ways to implement the data storage, but we didn't find any public and ready-to-use solutions. Therefore, it is not easy for us to compare the performance of graph streaming in MASS and Spark Streaming within the timeframe of this project. But this is also a work item that should be done in the future.

CHAPTER 5. CONCLUSION

Graph streaming is built into MASS, an agent-based parallel programming library, to better support massive graph processing. It is implemented by repetitively reading a portion of a graph and processing it while reading the next graph portion, which can reduce the construction overhead and prevent explosive growth of the agent population. HIPPIE and MATSim are two supported input graph file formats. This project also includes the implementation of two graph streaming benchmarks: Triangle Counting and Connected Components, to evaluate the performance of graph streaming. Both applications were executed with 1 – 24 computing nodes.

The execution results demonstrate significant CPU-scalable and memory-scalable improvements with multiple computing nodes. Furthermore, we compared the performance with non-streaming solutions. Graph streaming improved the performance when the resource is limited. It also served as a useful mechanism when the memory was not enough to process a huge graph all at once. Graph streaming avoids the explosive growth of the agent population and loads only a small portion of a graph, both efficiently using limited memory space and further improving the performance.

This project also has some limitations. First, if a graph is too big to fit into the distributed memory, this solution doesn't work as the whole graph is maintained in memory. Another is that this solution doesn't have advantages when processing small graphs because the computation suspension and resumption will affect the performance.

As for future work, the followings are some work items we should do:

1) Test graph streaming on directed and weighted graphs to verify the correctness of our implementation.

2) Compare the performance of graph streaming in MASS with Spark Streaming.

3) Expand graph streaming to support the file format SAR and DSL.

4) Explore other file partition strategies as the current implementation partitions a given file from top to down, which may end up too many connections between subgraphs. We should explore other strategies to split graphs into subgraphs so that the connections between subgraphs are as few as possible.

5) Expand graph streaming to support live data streams. This project streams data from static input files. To meet the demands of processing real-time evolving graphs from live feeds of social network platforms, we can expand the streaming source to include live data streams.

BIBLIOGRAPHY

[1] M.Fukuda, Parallel-Computing Library for Multi-Agent Spatial Simulation in Java, 2010

[2] Yuna Guo, Construction of Agent-navigable Data Structure from Input Files, 202, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/YunaGuo_whitepaper.pdf

[3] Brian Luger, Distributed Data Structures, 2021, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/BrianLuger_su21.pdf

[4] Justin Gilroy, Dynamic Graph Construction and Maintenance, 2020, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/JustinGilroy_whitepaper.pdf

[5] "HIPPIE." [Online]. Available: http://cbdm-01.zdv.uni-mainz.de/~mschaefer/hippie/information.php

[6] "MATSim.org." [Online]. Available: https://matsim.org/

[7] O. Tange (2011): GNU Parallel - The Command-Line Power Tool, login: The USENIX Magazine, February 2011:42-47.

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

[9] 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.

[10] Gilroy, J., Paronyan, S., Acoltzi, J., & Fukuda, M. (2020, December). Agent-navigable dynamic graph construction and visualization over distributed memory. In 2020 IEEE International Conference on Big Data (Big Data) (pp. 2957-2966). IEEE.

[11] "Spark Streaming" [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[12] Abughofa, T., & Zulkernine, F. (2017, December). Towards online graph processing with spark streaming. In 2017 IEEE International Conference on Big Data (Big Data) (pp. 2787-2794). IEEE.

[13] M. Zaharia, "Continuous Applications: Evolving Streaming in Apache Spark 2.0," Databricks Engineering Blog, 2016. [Online]. Available: https://databricks.com/blog/2016/07/28/continuous- applications-evolving-streaming-in-apache-spark-2-0.html

[14] Chang Liu, Development of Application Programs Oriented to Agent-Based Data Analysis, 2020, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/ChangLiu_wi20.pdf

APPENDIX A

Appendix A shows the execution results of Triangle Counting on HippieStreaming and Connected Components on HippieStreaming.

The input file for Triangle Counting is “40000_50.tsv” in the *mass_java_appl* repo under the branch “yan/graphstreaming_apps”.

The input file for Connected Components is “25000_40c.tsv” in the *mass_java_appl* repo under the branch “yan/graphstreaming_apps”.

Table 1 Triangle Counting with chunk size = 2000

	test 1	**test 2**	**test 3**
1 node	640.581	713.22	716.05
2 nodes	593.816	614.808	612.197
4 nodes	380.083	399.716	397.289
8 nodes	229.119	232.474	232.351
12 nodes	165.353	165.156	160.304
24 nodes	98.224	102.154	102.294

Table 2 Triangle Counting with chunk size = 4000

	test 1	**test 2**	**test 3**
1 node	833.476	959.386	957.879
2 nodes	658.061	677.636	676.245
4 nodes	345.328	338.555	338.815
8 nodes	175.727	181.586	177.708
12 nodes	126.061	120.65	127.161
24 nodes	80.572	84.08	81.139

Table 3 Triangle Counting with chunk size = 8000

	test 1	**test 2**	**test 3**
1 node	1134.91	1432.274	1390.235
2 nodes	889.956	923.345	941.507

4 nodes	375.211	376.642	376.434
8 nodes	170.404	171.635	173.888
12 nodes	115.835	113.622	114.147
24 nodes	69.334	75.803	69.411

Table 4 Connected Components with chunk size = 2000

	test 1	**test 2**	**test 3**
1 node	532.133	511.948	503.49
2 nodes	502.601	501.964	503.636
4 nodes	264.589	271.921	283.72
8 nodes	151.085	159.367	164.534
12 nodes	111.443	117.841	121.542
24 nodes	85.269	93.863	95.26

Table 5 Connected Components with chunk size = 4000

	test 1	**test 2**	**test 3**
1 node	902.164	940.353	928.922
2 nodes	666.147	678.881	672.025
4 nodes	254.375	259.45	263.559
8 nodes	133.947	138.194	146.94
12 nodes	94.877	99.572	100.594
24 nodes	66.836	73.766	71.725

Table 6 Connected Components with chunk size = 6000

	test 1	**test 2**	**test 3**
1 node	1213.205	1258.901	1255.169
2 nodes	903.239	908.839	905.044
4 nodes	288.979	302.547	301.039

8 nodes	137.591	141.875	141.781
12 nodes	95.838	119.347	100.802
24 nodes	63.82	65.198	67.326

This project graph streaming is in the *mass_java_core* repo under the branch "yan/develop".

Graph streaming benchmark programs are in the *mass_java_appl* repo under the branch "yan/graphstreaming_apps".

Install GNU Parallel tool: https://www.gnu.org/software/parallel/parallel_tutorial.html