© Copyright 2024

Yuan Ma

An Implementation of Multi-User Distributed Shared Graph

Yuan Ma

A white paper

submitted in partial fulfillment of the

requirements for the degree of

Master Of Science in Computer Science & Software Engineering

University of Washington

2024

Reading Committee:

Professor Munehiro Fukuda, Chair

Professor Kelvin Sung

Professor Robert Dimpsey

Program Authorized to Offer Degree:

Master of Science in Computer Science & Software Engineering

University of Washington

Abstract

An Implementation of Multi-User Distributed Shared Graph

Yuan Ma

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

Many real-world applications such as social or biological networks can be modeled as graphs. With the increasing size of graphs, graph databases also become a popular research area. Graph databases usually require maintaining the original structure of the graph over distributed disks or preferably over distributed memory to function. Compare to those popular data-streaming tools that need to disassemble the graph into texts before processing, it's reasonable to introduce agent-based graph computing in which we deploy agents to graphs without modifying the original shape. In this research, we introduce using Multi-Agent Spatial Simulations Library (MASS) for graph computing. Currently, most agent-based modeling (ABM) libraries including MASS focus on parallelization of ABM simulation programs. However, database systems need to accept, handle, and protect many queries from different users simultaneously, while MASS hasn't provided users with this capability. Therefore, this project aims at implementing a high-performance multi-user distributed shared graph and trying to add this feature to the MASS library. We have conducted

research on many popular data streaming tools and distributed cache. By addressing the challenges they have on programming graph applications, we proposed and implemented a high-performance distributed shared graph structure within the MASS library. Through the performance and programmability comparison between MASS and Hazelcast (which has a distributed HashMap data structure, thus enables distributed graph construction), we demonstrated that MASS GraphPlaces has better speed when processing graph queries and at the same time offers better parallel performance when programming graph applications such as Triangle Counting.

TABLE OF CONTENTS

List of Figures iii

List of Tables iv

Chapter 1. Introduction 6

Chapter 2. Background 8

- 2.1 Graphs in MASS 8
- 2.2 Previous Work 9

Chapter 3. Related Work 11

- 3.1 Data Streaming 11
- 3.2 Distributed Cache 12
- 3.3 Challenges 14

Chapter 4. Agent-Based Distributed Shared Graph 15

- 4.1 Multi-User Distributed Shared Graph 15
  - 4.1.1 Design 15
  - 4.1.2 Implementation 22
- 4.2 Agent-Based Graph in MASS - GraphPlaces 24
  - 4.2.1 Design 24
  - 4.2.2 Implementation 26

Chapter 5. Evaluation 28

- 5.1 Environment Setup 28

5.2 Basic Graph Operations Performance Comparison 29
5.3 Graph Application Programming Comparison 32
5.3.1 Triangle Counting Algorithm 32
5.3.2 Benchmark Settings 33
5.3.3 Performance Comparison 33
5.3.4 Programmability Analysis 37
Chapter 6. Conclusion 38
6.1 Summary 38
6.2 Limitations & Future Work 38
Bibliography 40
Appendix A 42
Appendix B 43

LIST OF FIGURES

Figure 1.1. MASS library model [1]. 6
Figure 2.1. Distributed graph visual representation. 8
Figure 4.1. The Aeron mechanism. 18
Figure 4.2. A multi-user distributed shared graph design. 19
Figure 4.3. An example of graph initialization. 20
Figure 4.4. An example of adding a multi-user vertex. 21
Figure 4.5. Graph representation by distributed map and vector. 25
Figure 4.6. Agent traversal in Triangle Counting. 26
Figure 4.7. Methods in MASS GraphPlaces. 27
Figure 5.1. 3000 vertices graph getVertex() performance comparison. 30
Figure 5.2. 5000 vertices graph getVertex() performance comparison. 30
Figure 5.3. 3000 vertices graph addVertex() performance comparison. 31
Figure 5.4. 5000 vertices graph addVertex() performance comparison. 31
Figure 5.5. 1000 vertices graph construction performance comparison. 34
Figure 5.6. 3000 vertices graph construction performance comparison. 34
Figure 5.7. 10000 vertices graph construction performance comparison. 35
Figure 5.8. 1000 vertices Triangle Counting performance comparison. 36
Figure 5.9. 3000 vertices Triangle Counting performance comparison. 36
Figure 5.10. 10000 vertices Triangle Counting performance comparison. 37

LIST OF TABLES

Table 5.1. CSSMPI and HERMES cluster environment. 29

Table 5.2. Graph data used for Triangle Counting application.. 33

Table 5.3. Quantitative programmability analysis... 37

ACKNOWLEDGEMENTS

Words cannot express my gratitude to my supervisor Prof. Fukuda, for his valuable guidance and advice. A special thanks to Michael Robinson-Elmslie who helped program and measure the Hazelcast applications. I also want to express my thanks to my friends in Distributed System Lab (DSLab), especially Warren and Shahruz for their help and support. I'm pleased to be a member of DSLab.

Chapter 1. INTRODUCTION

Multi-Agent Spatial Simulation (MASS) is an agent-based parallel programming library over a cluster of computing nodes. It provides an intuitive programming framework for big data processing that can simulate many real-world problems including bioinformatics, social networks, climate analysis and geographic information system. As shown in Figure 1.1, MASS processes communicate with each other through TCP connections to perform agent-based modeling (ABM) simulations.

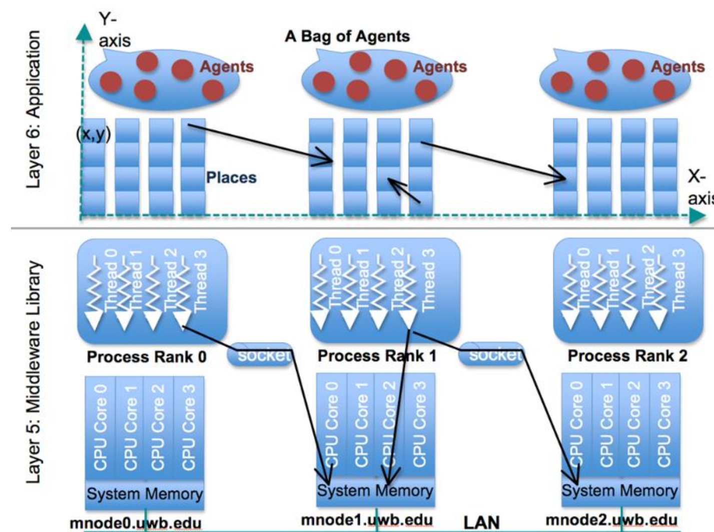

Figure 1.1. MASS library model [1].

MASS mainly contains two classes: Agents and Places. The former represents a collection of mobile objects in a simulation, each named agent. The latter represents a multi-dimensional array space of entities, each named place. Agents can migrate between places, regardless of the specific node or thread they are associated with. When an agent resides in a place, it can make function calls based on the data saved in that place.

While the base class Places is a distributed array, MASS also supports other data structures that extends from the base class including Binary Tree, Quadtree and Graph. These various data structures can meet the various computing needs from the users.

Most big-data computing handles text data with data-streaming tools such as MapReduce [2], Spark [3], and Storm [4]. However, for distributed data structures such as distributed graph, these data-streaming tools need to disassemble the data into texts that cannot remain in their original shape over distributed memory before processing the data. On the other hand, many graph applications including graph databases such as Neo4j [5] requires maintaining the original structure of the graph over distributed memory to function. Therefore, it's reasonable to introduce agent-based graph computing in which we deploy agents to graphs without modifying the original shape of the data structure [6].

Currently, agent-based modeling (ABM) libraries including MASS focus on parallelization of ABM simulation programs. However, database systems need to accept, handle, and protect many queries from different users, and ABM libraries do not have this capability. Therefore, in order to apply ABM libraries to database systems, in particular to graph database, the underlying graph must be accessible and modifiable by multiple users simultaneously. Given the above motivation, this project aims at investigating multi-user distributed shared graph (DSG) and trying to add this new feature to the MASS library.

This project aims to implement a high-performance distributed shared graph and build the implementation into MASS library, the goals include:

1. This project starts with surveys on platforms that facilitate shared space: Unix Shared Memory, Hazelcast [7], Redis [8] and Oracle Coherence [9]. Particularly, we want to check whether they can be used to create a distributed shared graph.

2. Based on the survey and prototyping, we will propose and implement a high-performance multi-user distributed shared graph.
3. The implementation of DSG will replace the original graph data structure in MASS called GraphPlaces and further facilitate relevant methods.
4. After the development of new MASS GraphPlaces, we will complete verification and performance measurement.

Chapter 2. BACKGROUND

This chapter mainly focuses on graph computing using MASS library, and describes the previous work finished by our lab members.

2.1 GRAPHS IN MASS

MASS has already supported a graph data structure called GraphPlaces that extends from the Places class. GraphPlaces store graphs in adjacency list format, which means for each vertex we store its corresponding attributes and its neighbors as a list. As shown in Figure 2.1, when a graph is inserted into a GraphPlaces it will be distributed in the cluster by storing information of vertices in different computing nodes.

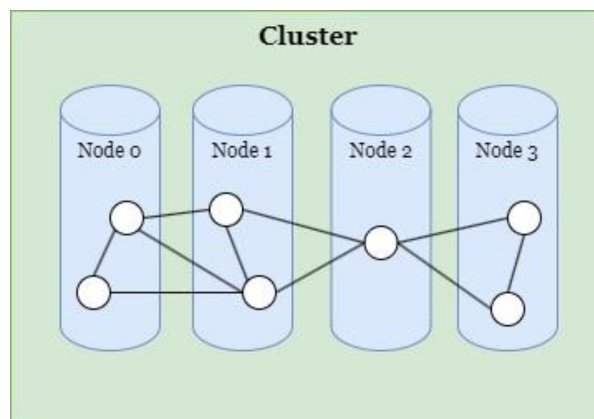

Figure 2.1. Distributed graph visual representation.

The vertices in GraphPlaces are represented by VertexPlace objects, and VertexPlace extends from the MASS Place class. As I mentioned, it stores the list of outgoing edges and the information about itself.

When constructing the graph, users can utilize the public built-in functions provided by GraphPlaces to load the graph from our supported graph formats including HIPPIE (Human Integrated Protein to Protein Interaction Reference), MATSim (Multi-Agent Transport Simulation), UW-Bothell proprietary DSL, and SAR. Users can also manually construct the graph by adding vertices or edges by calling addVertex() and addEdge() method.

After a graph is constructed, if the user wants to remove any vertices or edges they can just call removeVertex() and removeEdge(). Besides, getVertex() can be used for accessing the information of each vertex. Since VertexPlace extends from MASS Place class, agents can also reside in or migrate between VertexPlaces. When agents reside in a VertexPlace, it can make function calls based on the information of the corresponding vertex. As for agent migration, they can migrate over an outer edge from one VertexPlace to another.

2.2 Previous Work

The research done by J. Gilroy, S. Paronyan and J. Acoltzi focused on implementing distributed graph-computing support in MASS library [10]. They implemented the first version of GraphPlaces class coupled with the Cytoscape [11] graph visualization software. By comparing the graph construction performance with MapReduce and Spark, they concluded that the MASS GraphPlaces can be used to incrementally construct a graph in memory efficiently.

Further work by Y. Hong addressed the challenge of parallelizing the graph construction process [6]. Since graph construction for large graphs is time consuming, she introduced and implemented the pipelining approach in which the graph construction overhead is hidden with

graph computation. This new technique further speeds up graph computation using MASS GraphPlaces, and their results showed that the new pipelined implementation increased the performance by 7.7 times when running the Triangle Counting and Connected Components application on large graphs. She also mentioned in the future work section that more work should be done on measuring MASS graph computing performance with large graphs as well as try to interface MASS GraphPlaces to major graph databases.

C. Tsui implemented several agent-based graph applications using MASS GraphPlaces [12]. She completed the parallel implementation of three graph applications Graph Bridge, Minimum Spanning Tree, and Strongly Connected Components using MASS GraphPlaces and Spark. By the performance comparison and programmability analysis of the two frameworks, she concluded that when testing with large graphs MASS GraphPlaces has lower execution time than Spark while at the same time offers better programmability. In the future work section, she mentioned that further work needs to be done to compare MASS GraphPlaces with more libraries.

Lastly, V. Mohan and A. Potturi implemented three other graph applications using MASS GraphPlaces, including Bread-First Search, Triangle Counting, and Range Search [13]. They upgraded the agents in MASS library and automated agent propagation, forking, and flocking over GraphPlaces. By comparing with Repast Simphony [14], an open-source agent-based modeling platform, they demonstrated that their automated agent migration methods not only accelerated the execution but also improved programmability.

By comparing with data streaming tools and other agent-based libraries, we proved that MASS could achieve good performance and programmability when used for graph computing. However, previous students also mentioned that further comparisons need to be done to compare MASS GraphPlaces with more libraries. Since GraphPlaces store graphs using the adjacency list

format, we think a comparison with distributed key-value cache frameworks is necessary because vertices can be represented as a key-value pair of <Vertex ID, Neighbors List>. They also mentioned we need to try to interface MASS GraphPlaces to major graph databases. However, currently MASS doesn't support multi-user feature so as a result it cannot be used together with graph databases. These motivations from previous students motivates this project, so my goal is to implement the multi-user feature in MASS GraphPlaces and compare the performance with other distributed cache frameworks.

Chapter 3. RELATED WORK

This chapter covers the literature on data streaming tools and distributed cache frameworks and further discusses their challenges on implementing graph applications.

3.1 DATA STREAMING

MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks [15]. MapReduce mainly includes two phases: map and reduce. The map function transforms lines of text input into key-value pairs, then the reduce function processes all pairs associated with the same key and produce the final output. All the map and reduce functions are executed in parallel. However, MapReduce writes intermediate output to disk between each stage, as a result the disk I/O operations negatively affect the overall execution speed.

Spark addresses the problem of disk I/O operations and provides an in-memory parallel programming framework. It is an open-source computing framework that unifies streaming, batch, and interactive big data workloads to unlock new applications [16]. The most well-known concept in Spark is RDD (Resilient Distributed Datasets) which is the immutable collection of elements

that distributed over the cluster. However, MapReduce and Spark have one common disadvantage when processing graphs: they need to disassemble the data into texts before processing so as a result the graphs cannot remain in their original shape over the distributed disk or memory.

There are also data streaming frameworks designed for graph processing. Pregel is a graph processing architecture developed at Google. In Pregel architecture, programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology [17]. Apache Giraph [18] is another graph processing system built for high scalability that originates as the open-source counterpart to Pregel. Giraph adds several features beyond the basic Pregel model, including master computation, sharded aggregators, edge-oriented input, out-of-core computation, and more.

GraphX [19] is an embedded graph processing framework built on top of Spark which also implements the Pregel API. It offers a platform to implement graph applications by supporting graph related RDDs such as GraphRDD, VertexRDD, and EdgeRDD. However, because of the nature of Spark's data partition and shuffle operations, the performance of Spark GraphX is worse than MASS GraphPlaces for graph construction [10]. Also, for graph applications such as Graph Bridge and Strongly Connected Components MASS also provides better performance and programmability [12].

3.2 Distributed Cache

Since we want to implement a distributed shared graph storage by representing the graphs in adjacency list format, it's necessary to investigate on the distributed cache frameworks.

Memcached is a high-performance, distributed caching system [20]. It is a well-known, simple, in-memory caching solution mostly used to speed up dynamic Web applications by

alleviating database load. Facebook leverages Memcached as a building block to construct and scale a distributed key-value store that supports the world's largest social network [21]. However, Memcached only supports the key-value pair data type, and because of their design philosophy of avoiding disks as much as possible they do not have built-in support for saving data back to disks for backup.

Oracle Coherence [22] is an in-memory data grid enables fast access to key-value data. Coherence ensures scalability and performance in enterprise applications by providing clustered low-latency data storage, polyglot grid computing, and asynchronous event streaming. Coherence also provides multiple cache types including distributed cache, replicated cache, near cache, and more to support different needs of developers. However, the primary data type in Coherence is key-value pair, which means for most cases Coherence can only be used as a hash table.

Terracotta Ehcache [23] is an open source, distributed cache that provides high-performance, good scalability. Ehcache also introduces high portability by offering a complete implementation of the JSR107 JCache [24], which means developers can code directly with the JCache API. Terracotta's offheap technology also allows applications to store large datasets in-memory, outside of the Java heap, enabling high-performance data access for various types of data beyond simple key-value pairs. However, Ehcache also does not have built-in support for backup of cached data.

Redis [25] is an in-memory data store used as a cache, vector database, document database, streaming engine, and message broker. Redis has built-in replication and different levels of on-disk persistence. It supports different complex data types, with atomic operations defined on those data types. The simplicity, speed, and rich feature set of Redis have made it a popular choice among developers and companies.

Hazelcast [26] is another popular distributed cache solution that primarily focus on in-memory data management and distributed computing. It is a unified real-time data platform combining stream processing with a fast data store. Hazelcast also offers a large variety of data structures such as Map, JCache, Queue, Set, List, and more. Hazelcast Distributed maps, also known as Hazelcast IMaps [27], are key-value pairs that are partitioned across a cluster. Each member in a cluster stores an almost equal number of entries, and all maps are backed up by one other member to avoid data loss. Since Hazelcast focus more on distributed computing performance, it consistently outperforms Redis at scale [28].

3.3 CHALLENGES

Overall, data streaming frameworks are suitable for big data processing and analysis. However, when it comes to graph processing, the data streaming tools need to first transform graphs to texts before processing which affects overall performance. Previous research has shown that MASS GraphPlaces can provide better performance and programmability for graph computing compared to Spark GraphX.

Therefore, we mainly focused on the distributed cache frameworks which serves as a competitor of our distributed shared graph implementation. All of the introduced frameworks are in-memory, distributed and concurrent cache which means they can all be used to store graphs in adjacency list format. However, not all of them supports automate saving or checkpointing back to disk, which means the data stored in cache may have the risk of losing. It is also in doubt that whether distributed cache frameworks are suitable for graph computing. While key-value pairs are usually independent of each other, vertices in a graph have many edges pointing to each other which introduces different access patterns.

Based on the above survey, we plan to implement a high-performance distributed shared graph that addresses the challenges. The graph storage should be in-memory, distributed and can be concurrently accessed by multi-user, so that graph applications like graph databases can interface with our implementation. We also include the backup feature in which graphs are stored back to files for fault tolerance and convenience when we want to repeatedly work on the same graph. Furthermore, this implementation will provide basic graph related operations such as getVertex(), addVertex(), and addEdge(), while at the same time support agent migration within the data structure. Based on the literature review we will compare our implementation with Hazelcast in terms of both performance and programmability.

Chapter 4. AGENT-BASED DISTRIBUTED SHARED GRAPH

This chapter discusses how we managed to address the challenges mentioned in the previous chapter and designed our own implementation of multi-user, in-memory, distributed and coherent cache for storing graphs. Furthermore, we used this implementation to refine the graph structure in MASS, so we will explain our design of the new GraphPlaces implementation.

4.1 MULTI-USER DISTRIBUTED SHARED GRAPH

This subsection focuses on multi-user, in-memory, distributed, and coherent caching features, and we named the system as Distributed Shared Graph (DSG).

4.1.1 *Design*

The first challenge of the system design is to distribute the graph on multiple computing node so that we can utilize the computing resources to achieve parallel programming. We want each computing node to store an even portion of the graph so that we won't waste the memory space,

which can be achieved by a hash function that takes each vertex's identifier as an argument and returns a computing node number. As for storing a graph, we use HashMaps [32] in every computing node with key being the identifier of a vertex and value being the actual vertex object. The vertex object should at least include a list of outgoing edges from that vertex so that we store the graphs in adjacency list format. Since we don't want to launch the programs manually on each computing node, we need to launch remote worker processes from one master computing node. Given this challenge, the solution we came up with is using JSch [29], which is a Java utility for establishing remote connections using SSH2. JSch allows users to execute commands in remote machines, so in our case we can use it to launch remote processes. After launching remote processes, the channels between the master process and the remote processes will be established and can be used for communication. However, during the testing phase we found that although SSH sessions are held over TCP connections, this connection launched by JSch has lower performance than a customized TCP connection. Therefore, for effective communication between processes on different computing nodes, we also establish a complete network of TCP connections among all computing nodes.

Our second challenge is to find out a way to allow multiple users running their programs on the same cluster to share the same graph. First, we need an identifier for every graph so that when different users specified the same identifier, we know that they want to access the same graph. This identifier is a string which we name it as "sharedPlaceName". Initially we tried to use shared memory to store graphs, because shared memory is a memory that can be shared among multiple processes which is a way of inter-process communication. In this way, the graph is distributed and stored in each computing nodes' shared memory, and all users can access the shared memory for the graph data. To be more specific, in Linux the directory "/dev/shm" is an

implementation of the shared memory concept, and it is in the virtual memory which guarantees good performance. We create a graph file under this directory of every involving computing node so that they can be accessed by multiple users. As for the read and write operations to shared memory, they can be achieved by the MappedByteBuffer [30] class in Java programming.

However, when we were testing the performance, we found that the I/O operations to shared memory are too slow which leads to the low performance of put and get operations on graphs. Therefore, we realized that the graph data needs to be cached in the memory of all the processes for effective access. Since we have multiple users in the system who are accessing the same graph, caching the graph in each user's processes causes another problem which is the consistency of data among different users. Our solution is to cache the graphs in each user's process memory for quick access and at the same time implement a write-back write-update (WBWU) cache protocol among user processes. Write-back means we write back the graph data back to shared memory at the termination of a user's program because we really want to back up the data for potential risks and convenience of using the same graph repeatedly. Write-update means whenever a user modifies the graph, this update will be broadcasted to all other users so that they can update the cache in their processes. This broadcast will only happen in one computing node, for example if vertex A is supposed to be stored in computing node 1, then the update of vertex A only affects the user processes running on computing node 1 because the graph is evenly distributed and there's no replication. Since this broadcast needs to be reliably and high-performance, we found a tool called Aeron [31] which offers a low-latency message transport system. During our testing we also considered common communication protocols such as TCP and UDP, but we came into the conclusion that Aeron is the best for implementing our shared feature because of its mechanism shown in Figure 4.1. Aeron uses publication and subscription

mechanisms; publishers can send messages to specific channels and subscribers can subscribe to the channel so that they receive those messages. A channel is specified by an IP address, a port, and a stream ID, which means even if we only use one IP and one port, we can create multiple independent streams by changing the stream ID. As shown in Figure 4.1, all subscribers of a channel will receive messages from publishers of the same channel. If there's a subscriber who joined the channel late, then he will only receive the messages sent later than his join time. This model is suitable for our shared feature because all the users in the same node will be the subscriber and publisher of the same channel, and they can join at any time to get the latest updates from other users. Overall, Aeron and shared memory together achieved the WBWU mechanism of our system.

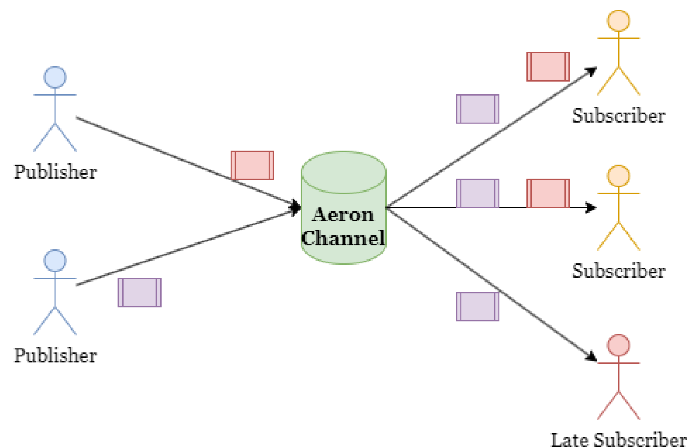

Figure 4.1. The Aeron mechanism.

After considering all the challenges, we came up with the overall system design shown in Figure 4.2. First, from the master node, we use JSch to launch remote processes in other worker nodes. Then we establish a complete network of TCP connections between all computing nodes for efficient communication. It should be noted that the JSch and TCP connections between the processes of User 2 were omitted from the Figure 4.2 due to concerns regarding the potential

cluttering effect on the overall clarity. We backup our graph in the files stored in the directory "/dev/shm", and we cache the most up-to-date graph data in the process memory. There's an Aeron channel launched in each computing node that supports communication between processes of different users. In this way, we designed a multi-user, in-memory, distributed and coherent graph storage system.

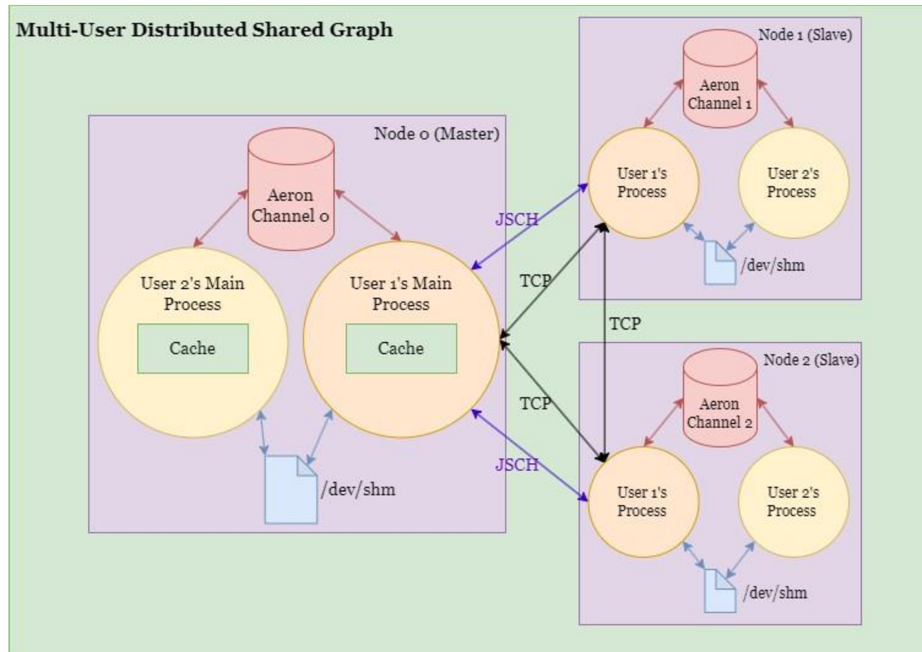

Figure 4.2. A multi-user distributed shared graph design.

Given the above explanation of the overall system, the following goes into more details about the WBWU mechanism among multiple users. The write back of data happens when a user terminates his program, and we use the MappedByteBuffer to write the graph data to the shared memory on all computing nodes. Since each graph has an identifier, the file name will be the combination of the identifier and the corresponding node. For example, one portion of the graph with identifier "myGraph" will be stored in computing node 0 as file "/dev/shm/myGraph-0". If next time anyone launched the program with the same identifier and set of computing nodes, the

program will automatically read from the shared memory on each computing node and retrieve the graph data. However, since we only write back the data at program termination, it's possible that another user's processes are running on the same graph and only those processes contain the most up-to-date data in their caches. Therefore, we also designed a graph initialization mechanism using Aeron as shown in Figure 4.3. When the program of a new user starts, it will first read the shared memory, but the initialized flag is set to false. Then, an initialization request will be sent to the Aeron channel, and the request will be received by all other users currently working on the same graph. They will reply with the most up-to-date data, and the new user process will update its cache correspondingly and set the initialized flag to be true. Once the initialized flag is set, later response messages from other users will be ignored, which avoids the waste of resources. This procedure happens in all related computing nodes.

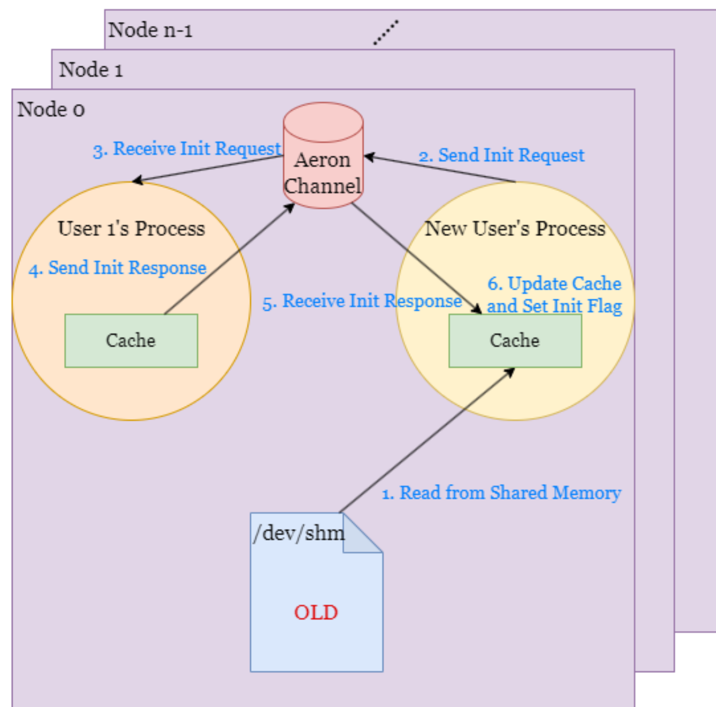

Figure 4.3. An example of graph initialization.

The write update feature can be explained with the addVertex() example shown in Figure 4.4. In the example, suppose there are currently 3 users running their program on the same graph and User 1 called addVertex(). We can calculate from the hash function about which computing node this vertex belongs to. If the result is a remote node we send a message from the master node to that remote node, otherwise we just perform operations locally. User 1 first updates its cache locally, and at the same time it sends a message to the Aeron channel located in that computing node. Aeron guarantees that all subscribers of the channel will receive this message, and the message asks User 2 and User 3 to update their caches by adding the new vertex. We also need to assure that User 1 will ignore this initiated message for avoiding a repetitive operation. If multiple users try to update a vertex with the same ID, then the final data of the vertex depends on the message arrival order at the Aeron channel. Since Aeron guarantees First-In First-Out (FIFO), each message will be processed one by one so the last message broadcasted will decide the result for that vertex.

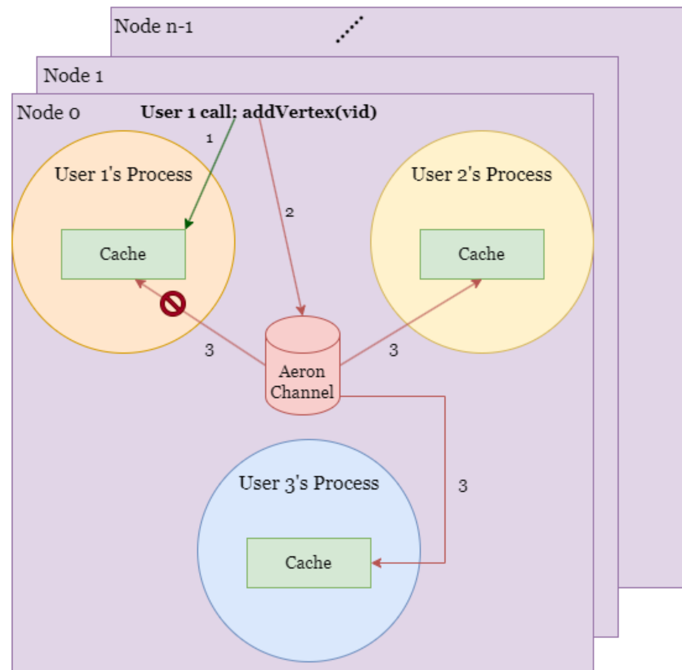

Figure 4.4. An example of adding a multi-user vertex.

4.1.2 *Implementation*

In this section, we would like to present the code snippets of two key classes in our DSG implementation. The first class is MProcess, this class represents processes running on remote computing nodes and is the most important part of our distributed feature. As shown in Listing 4.1, MProcess is a process that runs on remote computing nodes, and it's always listening to the JSch connection between the node and the master node. Once a message is received, MProcess checks the message action type. For example, the message type can be "DSG_SET_VERTEX" which means after the calculation of hash function the master node decided to add a new vertex in a remote node, then the MProcess running there should add the vertex to its cache locally. This remote process can be terminated by sending a message with message type "FINISH".

Listing 4.1. MProcess implementation code snippet

```
1.  while (alive) {
2.          Message m = receiveMessage();

3.          switch (m.getAction()) {
4.                  case ACK:
5.                          sendAck();
6.                          break;

7.                  case FINISH:
8.                          sendAck();
9.                          alive = false;

10.                 case DSG_GET_VERTEX:
11.                         Vertex v = getSharedGraph().getVertex(m.getVertexID());
12.                         sendAck(v);

13.                 case DSG_SET_VERTEX:
14.                         getSharedGraph().updateVertex(m.getVertexID(), m.getVertex());
15.                         sendAck();
16.         }
17. }
```

As for the shared feature, the most important class is called LocalMessageReaderThread. This class represents a thread that is actively listening on the Aeron channel subscription. As introduced in the previous section 4.1.1, Aeron is used when initializing the graph and updating

the shared graph. This thread has a similar structure as MProcess. The only difference is that it listens on an Aeron channel and Aeron uses a message handler as shown in Listing 4.2 to deal with received messages. It should be noted that the difference between the function updateVertex() and updateVertexLocally() is that updateVertex() additionally sends a message to the Aeron channel to notify other users. This message has a type of "DSG_SET_VERTEX_LOCALLY" and will be received by the LocalMessageReaderThread of other processes running in the same computing node. As for message type "DSG_INIT_REQUEST", a message of this type will be received from other users who just launch their programs. The current users can send their cache that contains the most up-to-date data back to the new user with the message type "DSG_INIT_RESPONSE". Once the new user receives the message, it calls setAdjMap() to update its cache and then the new user also has the new data.

Listing 4.2. LocalMessageReaderThread implementation code snippet

```
1.  FragmentHandler handler = (buffer, offset, length, header) -> {
2.        // get the message object
3.        byte[] byteArray = new byte[length];
4.        Buffer.getBytes(offset, byteArray);
5.        Message m = (Message) SerializationUtils.deserialize(byteArray);

6.        if (!m.getUserName().equals(getUserName())) {
7.              // this message is from others, deal with it
8.              if (m.getAction() == DSG_SET_VERTEX_LOCAL) {
9.                    getSharedGraph().updateVertexLocally(m.getVertexID(), m.getVertex());
10.             }
11.             else if (m.getAction() == DSG_INIT_REQUEST) {
12.                   getSharedGraph().helpOtherUsersInit(m.getUserName());
13.             }
14.             else if (m.getAction() == DSG_INIT_RESPONSE &&
    m.getReceiver().equals(getUserName()) && !initialized) {
15.                   initialized = true;
16.                   getSharedGraph().setAdjMap(m.getAdjMap());
17.             }
18.       }
19. };
```

4.2 AGENT-BASED GRAPH IN MASS - GRAPHPLACES

This section focuses on the differences between GraphPlaces and the Distributed Shared Graph implementation in design and implementation. We will explain how agents can traverse the GraphPlaces and help coding graph algorithms.

4.2.1 *Design*

The design of new MASS GraphPlaces is very similar to the Distributed Shared Graph (DSG) introduced in section 4.1. However, graph databases use vertex label and properties which means the identifier for a vertex can be of any type, this is different from the original design in DSG where vertices are identified with integers. Our solution is to let the identifier for a vertex be of data type Object in java, so that we can accept all kinds of identifiers. Also, we utilize the class VertexPlace which extends from the MASS Place class to represent a vertex object that can hold more information. However, we still want to remain the integer IDs for vertices due to the agent migration patterns in MASS. Therefore, we need to design a mapping mechanism from Object IDs to integer IDs, and we would like to use a distributed HashMap [32] together with Vectors [33] to address this problem. As shown in Figure 4.5, there should be a distributed HashMap that maps Object IDs to integer IDs and this HashMap can be accessed and modified by all computing nodes. The integer IDs are sequential and are assigned by the program manually when the vertices are added. As for the hash function of deciding where to store a vertex, we can simply use the below formula:

$$Computing\ Node\ ID = Mod(Vertex\ Integer\ ID,\ \#\ Computing\ Node)\ .$$

After reaching the corresponding computing node to store the vertex, we need to further decide the index of storing that VertexPlace object in the Vector, and this can be achieved by the formula:

$$Index\ in\ Vector = \left\lfloor \frac{Vertex\ Integer\ ID}{\#\ Computing\ Node} \right\rfloor.$$

In this way, we have both Object IDs and integer IDs for all vertices in a graph.

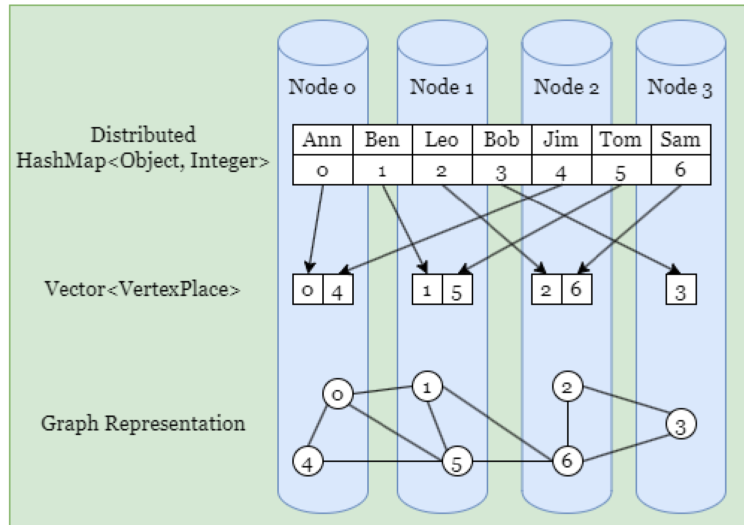

Figure 4.5. Graph representation by distributed map and vector.

Another difference between MASS GraphPlaces and DSG is that GraphPlaces extends from Places so GraphPlaces also needs to support agent migration. In MASS, agents can traverse from one place to another regardless of the computing node or thread they are associated with, and in graphs this corresponds to travelling from one vertex to another. Therefore, our solution is to let each VertexPlace represent a vertex. A VertexPlaces stores identifier and all attributes related to the vertex, as well as all the neighbors of the vertex in adjacency list format. Agents reside in a VertexPlace can migrate to other VertexPlaces along the edges between them. As shown in Figure 4.5, in the agent-based Triangle Counting application we can utilize this agent migration to achieve parallel programming. Initially we spawn agents in every VertexPlace and let them traverse the graph along the edges. Since we don't want to count the same triangle twice, we restrict the agents to only travel along an edge with destination vertex integer ID lower than current vertex. If there's no outgoing edges that fulfill the requirement then the agent will kill itself. However, if an agent has successfully migrated along the edges twice, in accordance with the rule, then it can try to find

if its original vertex is a neighbor of its current vertex. If the original vertex is within reach, then we found a triangle and we should increment the result by 1. In this way, we enabled the agents to migrate in GraphPlaces by traveling from one VertexPlace to another long the edges.

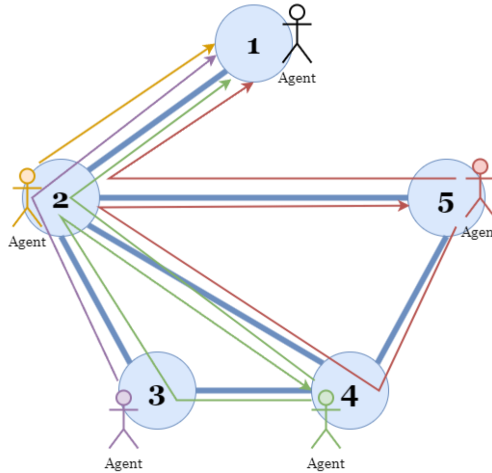

Figure 4.6. Agent traversal in Triangle Counting.

4.2.2 *Implementation*

The implementation of MASS GraphPlaces is very similar to DSG. We also utilize the classes MProcess and LocalMessageReaderThread shown in section 4.1.2, but we added more message action types due to the more supported public function. DSG only supports the two basic functions addVertex() and getVertex(), but as shown in Figure 4.7 MASS GraphPlaces has many public API for users to easily code graph applications. Users can use function loadDSLFile() and loadSARFile() to directly load graphs from corresponding format of graph files. Users can also manually modify the graph by adding or removing vertices and edges. Besides getting a single vertex information by getVertex(), MASS GraphPlaces also supports getGraph() which returns a complete graph model with all information. As for callAll() and exchangeAll(), these are methods that overrides the methods in base class Places. By calling these methods users can invoke a specific user-defined function in all VertexPlaces stored in the GraphPlaces.

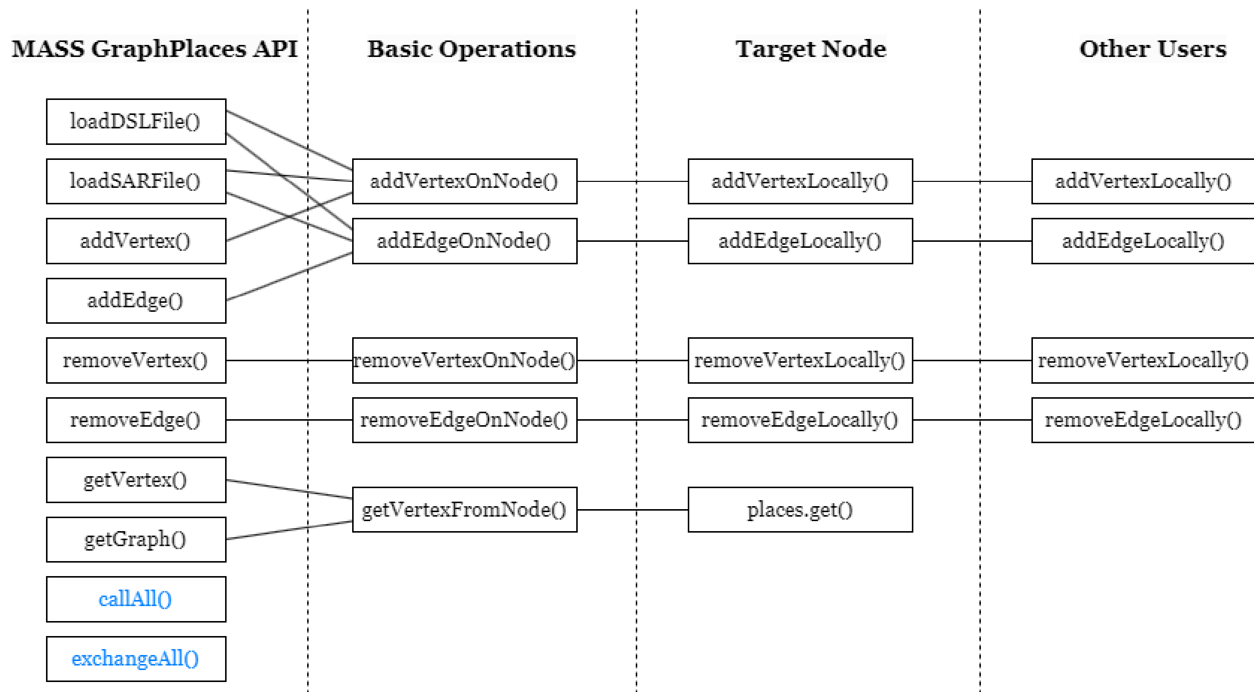

Figure 4.7. Methods in MASS GraphPlaces.

With these many public functions provided, users can use MASS GraphPlaces to code graph applications easily. Triangle Counting is an application that counts the number of triangles formed in a graph. We choose this application because it can be programmed using nearly the same algorithm in MASS and Hazelcast. We also considered other applications such as Connected Components, but since MASS use agent-based approach the complexity of the algorithm is in general different between MASS and Hazelcast. In the previous section, we showed Figure 4.6 which demonstrates the idea of agent-based Triangle Counting. As shown in Listing 4.3, we first initialize MASS and GraphPlaces with name “myGraph”. If this graph named “myGraph” has been loaded before then line 5 of loading graph file can be deleted, and this is because that previous loaded graphs are already stored in shared memory with the graph name as an identifier. Then we can use getGraph() to get the information of the whole graph. After instantiating agents at all vertices, we can start to do the computation. Since a triangle has 3 edges, for the first two steps we let agents travel along an edge to a lower ID vertex, and for the final step we let it travel back to

the source vertex. Since we kill the agents that has nowhere to migrate in each iteration, the number of agents alive at the end of simulation will be the number of triangles in the graph.

Listing 4.3. Triangle Counting MASS application abstract code.

```
1.          MASS.init();
2.          // initialize GraphPlaces with name “myGraph”
3.          GraphPlaces network = new GraphPlaces(“myGraph”);
4.          // load graph from a DSL format graph file
5.          network.loadDSLFile(filePath);
6.          // get list of vertices information
7.          List<Vertex> vertices = network.getGraph().getVertices();
8.          int numberOfVertices = vertices.size();
9.          // instantiate an agent at each vertex
10.         Agents crawlers = new Agents(network, numberOfVertices);
11.         // start triangle counting
12.         for (int i = 0; i < 3; i++) {
13.                 if (i < 2) {
14.                         // the first two steps, travel along edge to lower id vertex
15.                         crawlers.callAll(propagateDown_, i);
16.                 } else {
17.                         // the final step, travel back to original vertex
18.                         crawlers.callAll(migrateSource_, i);
19.                 }
20.                 // spawn, kill, migrate agents
21.                 crawlers.manageAll();
22.         }
23.         // number of triangles equals to number of agents alive at end of simulation
24.         print(“number of triangles = ” + crawlers.nAgents());
25.         MASS.finish();
```

Chapter 5. EVALUATION

This chapter first evaluates the performance of our distributed shared graph implementation, then evaluates the programmability and performance of the new MASS GraphPlaces. The results are compared to the Hazelcast implementation from another student in our lab, Michael Robinson-Elmslie.

5.1 Environment Setup

The applications were executed on the CSSMPI cluster and HERMES cluster at the University of Washington Bothell. During our tests we used up to 20 computing nodes, and the detailed information about those 20 machines can be found in Table 5.1.

Table 5.1. CSSMPI and HERMES cluster environment.

# Computing Nodes	# Logical CPU Cores	CPU Model	Memory
12	4	Intel Xeon Gold 6130 @ 2.10 GHz	16GB
3	4	Intel Xeon 5150 @ 2.66 GHz	16GB
4	8	Intel Xeon E5410 @ 2.33 GHz	16GB
1	4	Intel Xeon 5220R @ 2.20 GHz	16GB

5.2 Basic Graph Operations Performance Comparison

In this section we evaluate the performance of basic graph operations getVertex() and addVertex(). The two operations correspond to accessing the data of a vertex or adding a new vertex. We compared our Distributed Shared Graph implementation with Hazelcast for the two operations. For the Hazelcast implementation of graph storage, we utilized the Hazelcast Distributed Map to store the graph in adjacency list format. To be more specific, for each vertex we store it into a Hazelcast Imap as a key value pair of <vertex ID, vertex object>. The getVertex() and addVertex() methods are implemented using the get() and put() methods of IMap. For the performance measurement, we run our benchmark programs at the master node and count the total time of iterating through all vertex IDs in the graph to perform two operations.

Figure 5.1 and 5.2 show the performance comparison of DSG and Hazelcast on getVertex() operation. We can see that no matter what size the graph is, in both single node and 4-node scenario DSG performs better than Hazelcast. This may be due to the fault tolerance mechanism of Hazelcast. By default, all entries of Hazelcast IMap are backed up by one other member in the cluster to avoid data loss. This introduces high availability but at the same time reduces the

performance. On the other hand, DSG only back up data to the shared memory at program termination, and this ensures the high performance of getVertex().

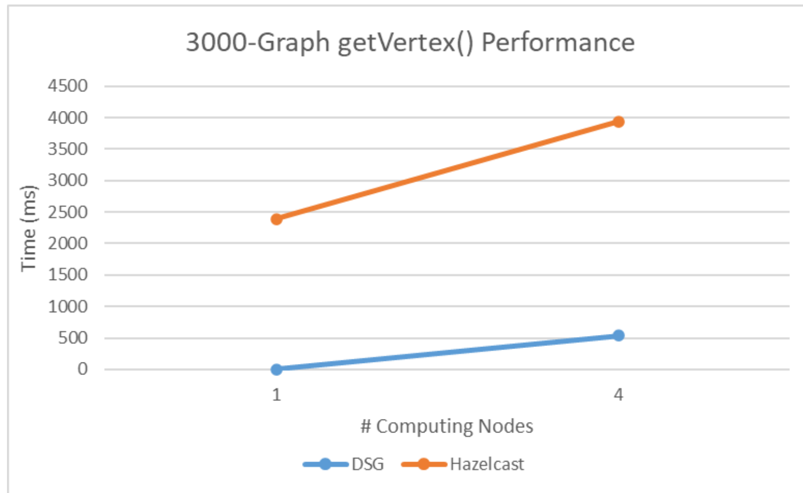

Figure 5.1. 3000 vertices graph getVertex() performance comparison.

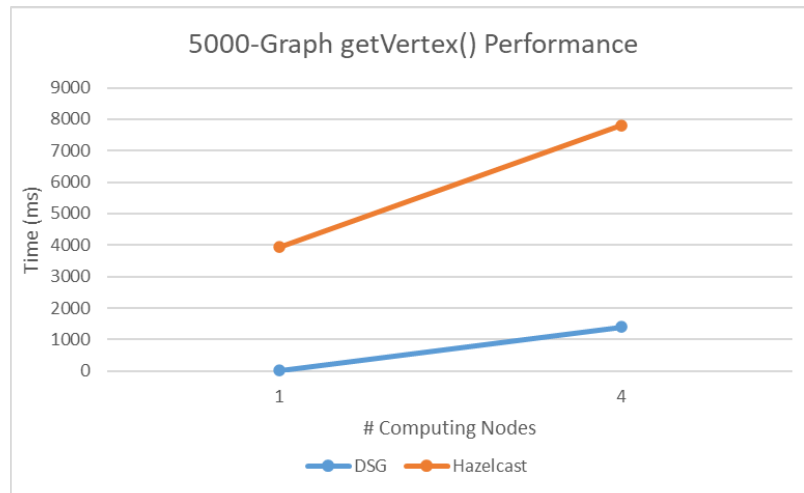

Figure 5.2. 5000 vertices graph getVertex() performance comparison.

As for addVertex() operation, the performance comparison results are shown in Figure 5.3 and 5.4. We can see that when using a single computing node, Hazelcast performs addVertex() quicker than DSG. However, when using 4 computing nodes, the execution time

significantly dropped for DSG and it becomes quicker than Hazelcast. This result can also be explained by the fault tolerance mechanism of Hazelcast. When there's only one computing node, Hazelcast does not create backup data. On the other hand, if multiple computing nodes are used, each insertion creates replicas of the entry and stored in different computing nodes, and this introduces extra overhead.

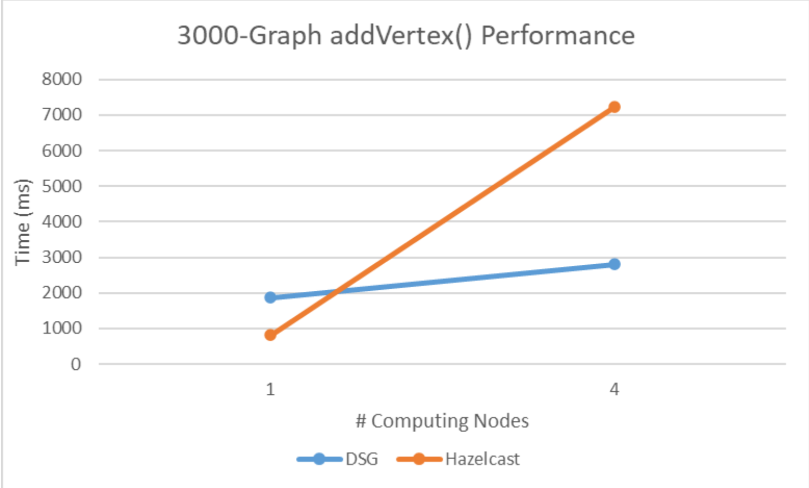

Figure 5.3. 3000 vertices graph addVertex() performance comparison.

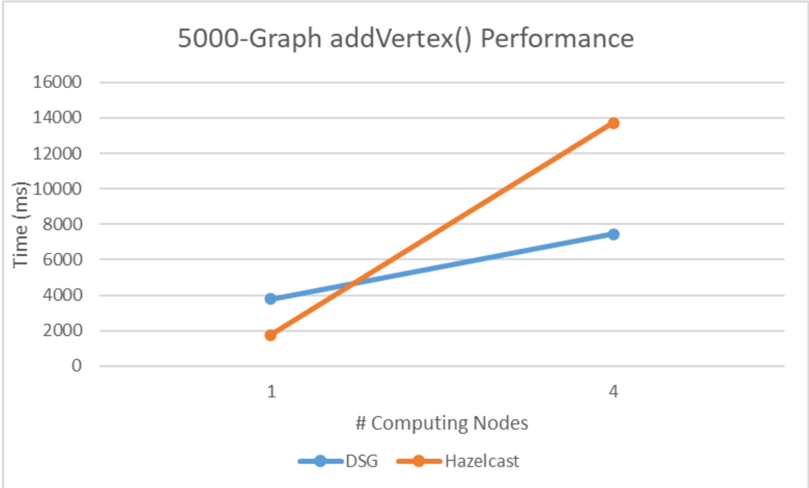

Figure 5.4. 5000 vertices graph addVertex() performance comparison.

Overall, as a distributed in-memory shared graph storage system, DSG performs the basic put and get operations faster than Hazelcast in multi-node scenario. We also introduced other graph queries in both DSG and Hazelcast implementation. The graph queries include getting neighbors of a vertex, getting parents of a vertex, getting grandparents of a vertex, getting vertex with minimum number of edges and getting highest degree vertex. The detailed performance comparison can be found in Appendix A Table 1, and from the data we can conclude that in general DSG performs better than Hazelcast in executing graph queries.

5.3 Graph Application Programming Comparison

In this section we compare MASS GraphPlaces with Hazelcast by running the Triangle Counting application. We compare the performance from graph construction time and application execution time, and also compare the programmability of MASS GraphPlaces with Hazelcast.

5.3.1 *Triangle Counting Algorithm*

Given an undirected graph $G = (V, E)$, the triangle counting problem asks for the number of triangles in this graph. MASS GraphPlaces used the agent-based mechanism to solve this problem as previously discussed in section 4.2.2. On the other hand, since Hazelcast does not support agent-based computing, we implement the Triangle Counting application with the algorithm shown in Listing 5.1. The idea is similar to an agent traversing a triangle path in a graph, but in Hazelcast we can only calculate using the graph data. For each vertex, we mark it as the start point and iterate through its neighbor list. Then for each neighbor vertex, we iterate through its neighbor list again and try to find out neighbors that adjacent to the start point. Since we are using an undirected graph, each unique triangle will be counted 6 times, so we need to divide the result by 6 to get the number of unique triangles in the graph.

Listing 5.1. Triangle Counting algorithm used by Hazelcast implementation.

```
1.          numTriangles = 0
2.          for v = 1 … n do
3.                  for each neighbor u of v do
4.                          for each neighbor w of u do
5.                                  if vw forms an edge then
6.                                          numTriangles = numTriangles + 1
7.          return numTriangles / 6.
```

5.3.2 *Benchmark Settings*

For our experiment, we used the same graphs for MASS GraphPlaces and Hazelcast. The detailed graph data can be found in Table 5.2. During our experiments, we used from 1 computing node up to 8 computing nodes, and we also verified that the programs provide the correct result. All the timing data is in milliseconds and represents the average time of three runs.

Table 5.2. Graph data used for Triangle Counting application.

Number of Vertices	**Number of Edges**	**Number of Triangles**
1000	93480	165138
3000	293804	192146
10000	989990	200053

5.3.3 *Performance Comparison*

Before running the application, MASS and Hazelcast both need to load the graph into the distributed cache, and we call this step graph construction. Figure 5.5, 5.6, and 5.7 shows the graph construction performance comparison between MASS and Hazelcast. From the figures we can see that no matter what size the graph is and how many computing nodes are used, MASS performs better than Hazelcast. This is because that the new MASS GraphPlaces directly adopts the structure

of DSG and as a result the addVertex() method is of high-performance. While MASS and Hazelcast both use addVertex() function to construct a graph, MASS performs better because the addVertex() function performs quicker.

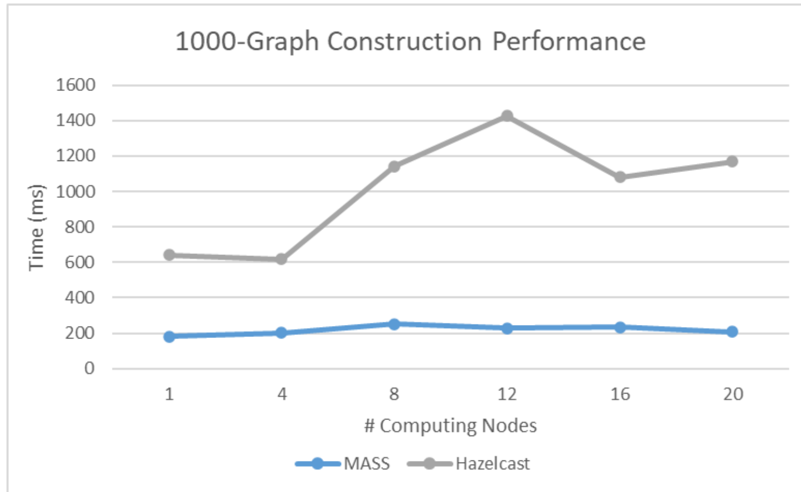

Figure 5.5. 1000 vertices graph construction performance comparison.

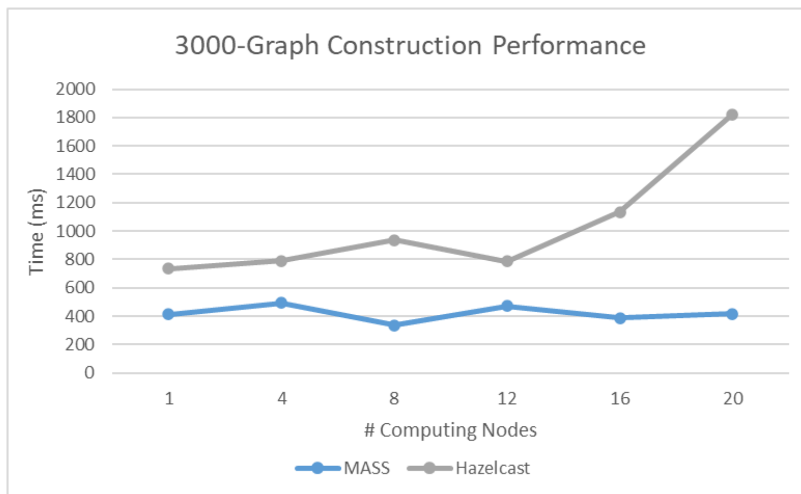

Figure 5.6. 3000 vertices graph construction performance comparison.

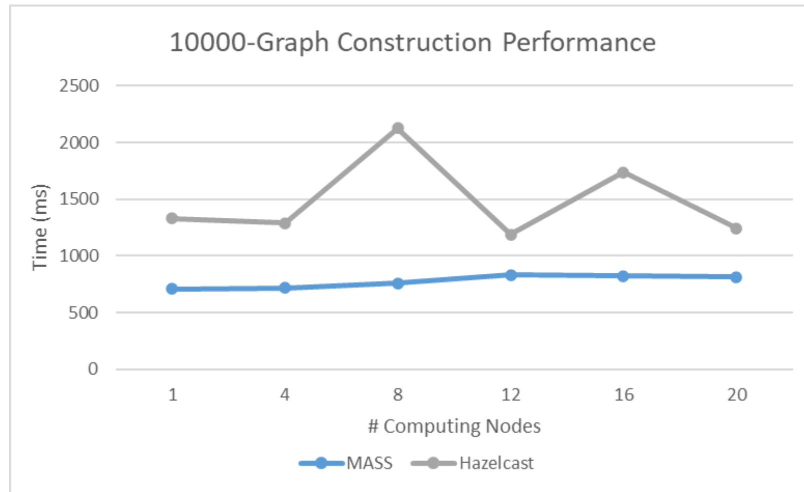

Figure 5.7. 10000 vertices graph construction performance comparison.

When it comes to the execution of Triangle Counting, from Figure 5.8, 5.9 and 5.10 we can see that MASS is nearly twice as fast as Hazelcast when we are using multiple computing nodes. For single node execution, MASS has similar performance to Hazelcast. The problem is that MASS spawns too many agents which takes up more memory space than Hazelcast. This can be proved by the fact that MASS run out of memory when using a single node to execute Triangle Counting on the size 10000 graph. However, from the data we can also see that the performance of MASS improved greatly when adding more computing nodes, while the parallel performance of Hazelcast is not linear. This can be explained by the difference in system design between MASS GraphPlaces and Hazelcast. While GraphPlaces used shared memory for backup to ensure the high performance during computation, Hazelcast focuses more on the availability of data so as a result the overall parallel performance is sacrificed.

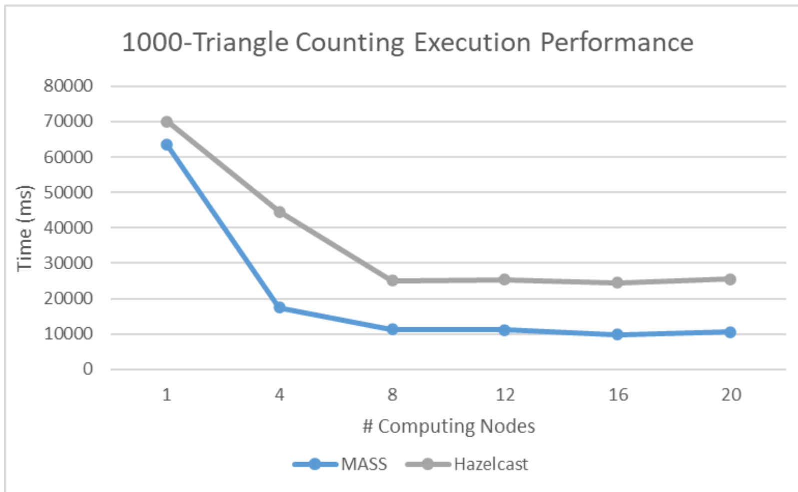

Figure 5.8. 1000 vertices Triangle Counting performance comparison.

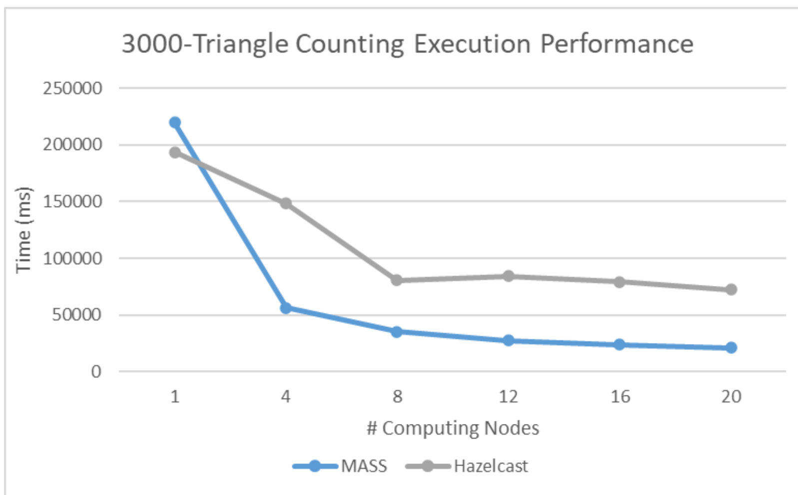

Figure 5.9. 3000 vertices Triangle Counting performance comparison.

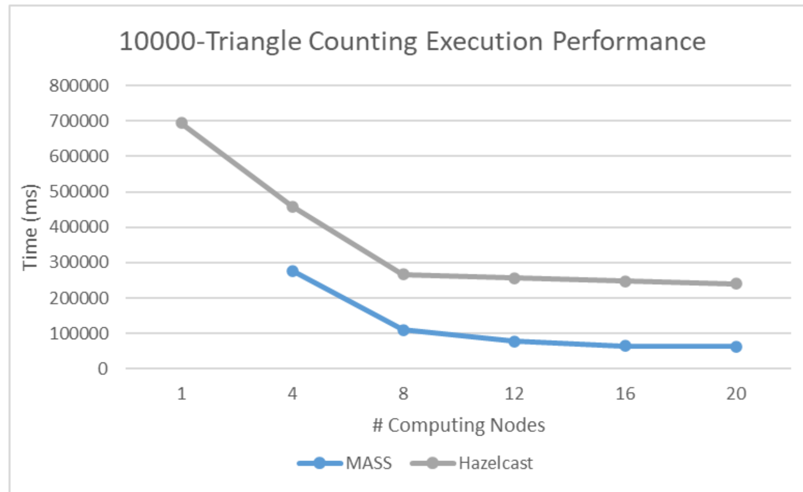

Figure 5.10. 10000 vertices Triangle Counting performance comparison.

5.3.4 *Programmability Analysis*

We also performed a quantitative programmability analysis of the two Triangle Counting applications implemented using different frameworks. As detailed in Table 5.3, the Lines of Code (LoC) and lines of boilerplate code in Hazelcast is higher than MASS, because of the need for code defining the graph related operations. This introduces a bigger percentage of boilerplate code in the Hazelcast Implementation. On the other hand, the cyclomatic complexity of MASS is larger than Hazelcast. This is because that in MASS applications users need to make decisions on the appropriate agent migration methods as well as other graph related built-in functions.

Table 5.3. Quantitative programmability analysis

	LoC	**Boilerplate Code**	**Boilerplate Percentage**	**Cyclomatic Complexity**
MASS	175	13	7.43%	3.875
Hazelcast	372	49	13.17%	2.8

Chapter 6. CONCLUSION

This chapter concludes the whole paper by summarizing the contents and give future directions.

6.1 SUMMARY

This project is motivated by the limitation of data streaming frameworks and the popularity of graph databases. After carefully examining existing systems, we addressed the challenges of existing distributed cache frameworks and proposed a multi-user distributed shared graph implementation. We built the new MASS GraphPlaces based on our implementation and achieved our goal of enabling multiple users to share a same graph. This project also compares the performance and programmability of the new MASS GraphPlaces with Hazelcast by running various graph queries and the Triangle Counting application with up to 20 computing nodes.

The execution results of the graph queries demonstrate the high-performance of our proposed distributed in-memory shared graph storage system. Our designed data structure is also suitable for agents to migrate. We compared the new MASS GraphPlaces with Hazelcast by running the Triangle Counting application, and we came into the conclusion that the revised MASS GraphPlaces has good programmability and performance when programming graph applications. Furthermore, the results also show the significant CPU-scalability of MASS as a parallel programming framework.

6.2 LIMITATIONS & FUTURE WORK

This project also has some limitations. First, from the execution performance we can see that if only one computing node is used to run applications built in MASS GraphPlaces, then the good performance is not guaranteed. Second, although the current hash function used to determine the

distribution of vertices can distribute the graph evenly, it does not take the vertex locality into consideration. This may introduce extra communication between processes when executing graph applications using multiple computing nodes. Lastly, if a graph is too big and as a result too many agents are spawned during the simulation, our program may run out of memory.

Based on the above limitations, we propose the followings for future work:

1. Further improve the performance of GraphPlaces by implementing a better hash function to distribute the vertices of a graph. We need to consider the edges between the vertices to address locality.
2. Currently only GraphPlaces can be shared by multiple users, but GraphPlaces actually extends from Places. Future work should be done to enable the shared feature in the base Places class. Since currently MASS Places is a distributed array, it can be changed to a distributed Vector, and any changes on this Vector needs to be shared among multiple users. Then the next step is to rework on GraphPlaces to delete the original Vector<VertexPlace> and store data in the new Vector<Place>.
3. More graph applications should be developed to further test the performance of MASS GraphPlaces. Since currently we only have Triangle Counting programmed in both MASS and Hazelcast, we can only use it for benchmark. Further work needs to be done to develop more graph applications using MASS and Hazelcast to make a comprehensive comparison.
4. After optimizing GraphPlaces and Places, more work needs to be done to optimize agents in MASS. Since agents are the computational units in agent-based modeling libraries, the performance of agents largely affects the performance of MASS.

BIBLIOGRAPHY

[1] J. Timothy Chuang, Munehiro Fukuda, A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems, In Proc. of the 16th IEEE International Conference on Computational Science and Engineering - CSE 2013, pages 143-150, Sydney, Australia, December 3-5, 2013

[2] MapReduce, "Accessed on: July 18, 2023. [Online]. Available: https://hadoop.apache.org"

[3] Spark, "Accessed on: July 18, 2023. [Online]. Available: https://spark.apache.org"

[4] Storm, "Accessed on: July 18, 2023. [Online]. Available: https://storm.apache.org"

[5] Neo4j, "Accessed on: July 18, 2023. [Online]. Available: https://neo4j.com"

[6] Y. Hong and M. Fukuda, "Pipelining Graph Construction and Agent-based Computation over Distributed Memory," *2022 IEEE International Conference on Big Data (Big Data)*, Osaka, Japan, 2022, pp. 4616-4624

[7] Hazelcast, "Accessed on: July 18, 2023. [Online]. Available: https://hazelcast.com"

[8] Redis, "Accessed on: July 18, 2023. [Online]. Available: https://redis.com"

[9] Oracle Coherence, "Accessed on: July 18, 2023. [Online]. Available: https://www.oracle.com/java/coherence"

[10] J. Gilroy, S. Paronyan, J. Acoltzi and M. Fukuda, "Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory," in 2020 IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 2020 pp. 2957-2966.

[11] Cytoscape: An Open Source Platform for Complex Network Analysis and Visualization, "Accessed on: May 4, 2024. [Online]. Available: https://cytoscape.org"

[12] Agent-based Graph Applications in MASS Java and Comparison with Spark, "Accessed on: July 13, 2023. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/CarolineTsui_whitepaper.pdf"

[13] V. Mohan, A. Potturi and M. Fukuda, "Automated Agent Migration over Distributed Data Structures" in Proc. of the 15th International Conference on Agents and Artificial Intelligence pages 363-371, February 2023

[14] Repast Simphony, "Accessed on: May 5, 2024. [Online]. Available: https://repast.github.io"

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113.

[16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. Commun. ACM 59, 11 (November 2016), 56–65.

[17] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). Association for Computing Machinery, New York, NY, USA, 135–146.

[18] Giraph, "Accessed on: April 16, 2024. [Online]. Available: https://giraph.apache.org/literature.html"

[19] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: graph processing in a distributed dataflow framework. In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI'14). USENIX Association, USA, 599–613.

[20] Fitzpatrick, B. Distributed caching with memcached. Linux Journal, 124 (Aug. 2004), 72--78.

[21] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., & Venkataramani, V. (2013). Scaling Memcache at Facebook. *Symposium on Networked Systems Design and Implementation*.

[22] Oracle Coherence, "Accessed on: April 16, 2024. [Online]. Available: https://www.oracle.com/technetwork/middleware/coherence/learnmore/oracle-coherence-12c-deployment-3049521.pdf"

[23] Terracotta Ehcache, "Accessed on: April 16, 2024. [Online]. Available: https://www.ehcache.org/about/features.html"

[24] JSR107 JCache, the Java Caching API, "Accessed on: May 6, 2024. [Online]. Available: https://github.com/jsr107/jsr107spec"

[25] Redis Cluster Specification, "Accessed on: April 16, 2024. [Online]. Available: https://redis.io/docs/latest/operate/oss_and_stack/reference/cluster-spec/"

[26] Hazelcast, "Accessed on: May 6, 2024. [Online]. Available: https://hazelcast.com"

[27] Hazecast Distributed Map, "Accessed on: April 16, 2024. [Online]. Available: https://docs.hazelcast.com/hazelcast/latest/data-structures/map"

[28] Hazelcast vs. Redis Performance Benchmarks, "Accessed on: May 6, 2024. [Online]. Available: https://hazelcast.com/resources/hazelcast-vs-redis"

[29] JCraft JSch - Java Secure Channel, "Accessed on: April 28, 2024. [Online]. Available: http://www.jcraft.com/jsch/"

[30] Class MappedByteBuffer, "Accessed on: April 28, 2024. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/nio/MappedByteBuffer.html"

[31] Aeron, "Accessed on: April 28, 2024. [Online]. Available: https://aeron.io/"

[32] HashMap, "Accessed on: May 9, 2024. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html"

[33] Vector, "Accessed on: May 9, 2024. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html"

APPENDIX A

Table 1. Extra graph queries performance comparison.

Avg Time (milliseconds)	# Computing Nodes	DSG	Hazelcast
Get Neighbors	1	0.02	0.23
	4	1.03	0.94
	8	0.71	1.09
Get Parents	1	9.83	31.53
	4	6.39	29.65
	8	5.48	16.41
Get GrandParents	1	78.2	118.14
	4	28.49	101.73
	8	19.15	57.71
Get Min Edges	1	2.84	24.9
	4	2.03	22.96
	8	1.82	11.01
Get Highest Degree	1	5.7	54.05
	4	3.88	49.47
	8	3.42	24.6

APPENDIX B

DSG is located in the mass_java_core repository under the branch “chrisma/develop”. There is a README file that provides instructions on how to compile and run the code.

The new MASS GraphPlaces is located in the mass_java_core repository under the branch “chris/newGraphPlaces”. The new version is fully compatible with previous versions so all previous built applications should work with the new version.

To use the shared feature, since the new MASS detects multiple users by the username specified in the file “nodes.xml”, the username attribute needs to be added for all nodes including the master node. Listing 1 shows an example. Different users who want to share the same GraphPlaces need to have the “same” nodes.xml file. This means they need to start their programs from the same master node and use the same worker computing nodes. For example, User A and User B should all start their program on cssmpi1h with cssmpi2h-12h being remote nodes.

Listing 1. Example nodes.xml.

```
<nodes>
        <node>
                <master>true</master>
                <hostname>cssmpi1h.uwb.edu</hostname>
        <masshome>/home/NETID/yuanma/mass_java_appl/Graphs/TriangleCounting/
</masshome>
                <username>yuanma</username>
                <port>33333</port>
        </node>
        <node>
                <hostname>cssmpi2h.uwb.edu</hostname>
        <masshome>/home/NETID/yuanma/mass_java_appl/Graphs/TriangleCounting/
</masshome>
                <username>yuanma</username>
                <privatekey>~/.ssh/id_rsa</privatekey>
                <port>33333</port>
        </node>
</nodes>
```

One should create a GraphPlaces object by specifying a shared place name using the constructor:

```
public GraphPlaces(int handle, String className, String sharedPlaceName);
```

It should also be noted that all users need to specify the same sharedPlaceName to share the same GraphPlaces. After that, all modifications on the graph (e.g. add/remove vertices/edges) will be shared among multiple users.