An Implementation of Multi-User Distributed Shared Graph

Yuan Ma

Term Report

submitted in partial fulfillment of the

requirements of the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

December 13, 2023

Project Committee:

Professor Munehiro Fukuda, Committee Chair

Professor Kelvin Sung, Committee Member

Professor Robert Dimpsey, Committee Member

Project Overview

Multi-Agent Spatial Simulations Library (MASS) is a parallel-computing library for multi-agent and spatial simulation over a cluster of computing nodes. 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 [1].

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 database 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.

Project Goals

1. The project will get started with surveys on several platforms that facilitate shared space: Unix Shared Memory, Hazelcast [7], Redis [8], Oracle Coherence [9], and AWS SimSpace Weaver [10]. Particularly, we will check whether they can be used to create a distributed shared graph and compare their speed of running the same graph algorithm.
2. Based on the survey and prototyping, we will propose and implement a high-performance multi-user DSG.
3. Our implementation will replace MASS Places or MASS GraphPlaces and furthermore facilitate relevant methods.
4. After integrating DSG with MASS, we will complete verification and performance measurement for DSG-integrated MASS.

Progress

According to the project plan proposed, my first quarter should focus on implementing distributed shared graph using chosen platforms. Since we want to implement a high performance distributed shared graph without dependency on other platforms, we chose Unix Shared Memory as a basis for our own implementation. Because that we would like to use adjacency lists to represent graphs, we chose Hazelcast which offers high-performance key-value storage for comparison.

Our initial implementation based on Unix Shared Memory is shown in Fig.1 and Fig. 2. According to our definition of a distributed shared graph, the graph should be stored across a cluster in a distributed fashion and can be accessed by multiple users. For each graph vertex, when it is inserted, we pass its unique identifier (ID) to a hash function, and the output of the function indicates which node to store it. Each node will maintain a map that stores all pairs of vertex and corresponding adjacency list. As for accessing a vertex, we need to have the ID, and we pass it to the same hash function then we can know which node stores that vertex.

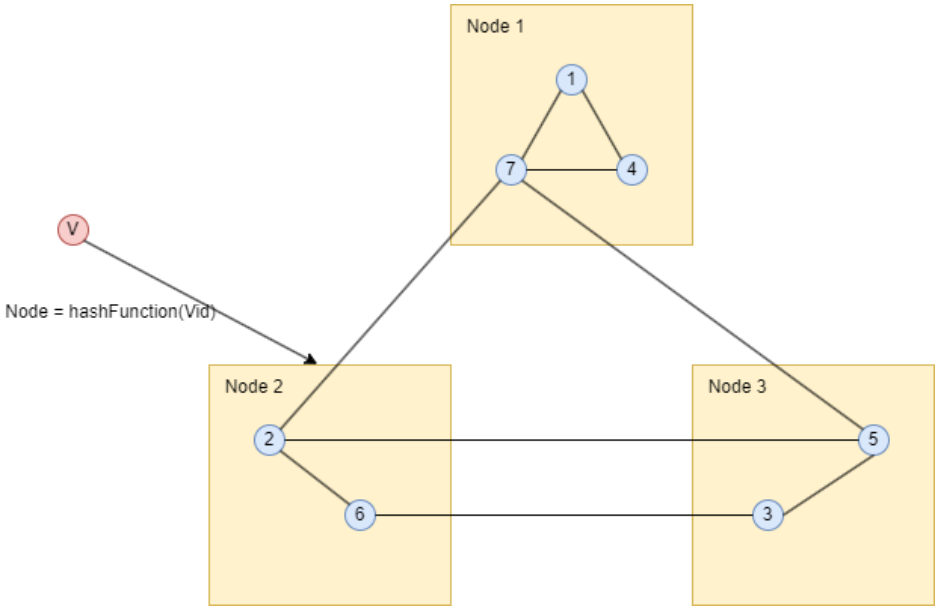

Fig. 1 Distributed Graph Storage

Unix Shared Memory is mainly used for the multi-user feature, and the details are shown in Fig.2. As I mentioned, each node maintains a map that stores vertices and adjacency lists, and we would like to store it in a file under the directory "/dev/shm". The reason is that "/dev/shm" in Linux is an implementation of the shared memory concept, and it is in the virtual memory which guarantees good computing performance. I tested that a file stored under "/dev/shm" can be accessed by multiple users using a MappedByteBuffer [11] in Java programming.

Overall, we have two scenarios. If a user is using the same computing node as the vertex he wants to access or insert, then just directly use MappedByteBuffer IO to access the map stored in "/dev/shm". If a user wants to access a vertex and after using hash function for computing we find it's in another node, then we initially planned to use either TCP or UDP to pass the data from or to the shared file in another node. This second scenario will contain two operations, one is the shared file access, and another is inter-process communication between processes running on different computing nodes.

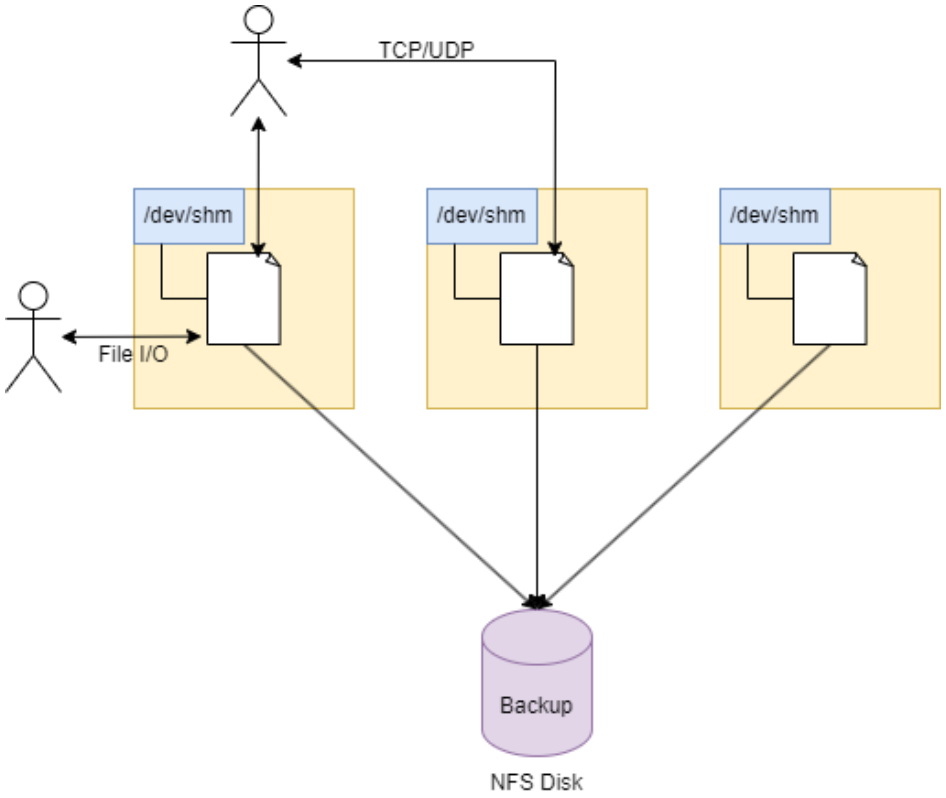

Fig.2 Initial Distributed Shared Graph Implementation

Since we really want to have a high-performance implementation, we need to make sure that the most basic operations in graphs: inserting vertex and accessing vertex are quick and reliable. Therefore, after I verified that Unix Shared Memory has the feasibility to build a distributed shared graph, I compared the performance with Hazelcast, the results are shown in Table.1 and Table.2.

Table.1 Initial Vertex Insertion vs. Hazelcast

#insert operations	MappedByteBuffer	MappedByteBuffer+TCP	Hazelcast
1000	165	185	300
10000	800	840	750
100000	4000	4230	3500

Table.2 Initial Vertex Access vs. Hazelcast

#access operations	MappedByteBuffer	MappedByteBuffer+TCP	Hazelcast
1000	300	320	100
10000	1400	1440	350
100000	6900	7130	2400

As I mentioned, the "MappedByteBuffer" column corresponds to the user and vertex on the same computing node scenario, while "MappedByteBuffer+TCP" corresponds to the second scenario in which I tested using TCP to send data between different computing nodes. As we can see from Table.1 and Table.2, the insertion operation is of nearly equal performance with Hazelcast, but the access operations are nearly 3-4 times slower.

According to the comparison results, we need to come up with a new design of the system in order to make our performance competitive. Therefore, I spent the rest of the quarter exploring ways to improve performance and refine the overall design of the system. I will present the results in the next section.

Solution & Results

Although the initial results are disappointing, but we find out that actually inter-process communications are quicker than shared memory operations, because we can see that the TCP communication time is minimal comparing to MappedByteBuffer operations. This is probably because that IO operations usually invokes the operating system so they are limited. I tried to use Java Native Interface (JNI) [12] to call the "shm_open()" [13] function in C/C++ to see if the shared memory operations performance can be better, but the improvement is minimal.

Therefore, to improve access operation performance, we come up with a new design based on the previous observations. We simply cache the adjacency lists in the process memory. In this way, vertex access becomes much quicker directly using process memory.

However, this new setting introduces another problem: when multiple users are present, how to access or insert vertex to the shared graph? The problem is now the most up-to-date data is stored in each user's process memory, but not in shared memory, so an insert operation in one process needs to invalidate or update the data in other processes running on the same computing node as well.

Since shared memory operations are slow, we tried to address this problem using inter-process communication. In the previous test I used TCP, it has

good performance and offers good reliability for communication between different computing nodes. Originally, we planned to form a cluster of TCP connection between each pair of nodes in the cluster, and this is already resource costing because each process needs a thread to listen to each connection it maintains to receive messages. Not to mention that after adding the cache mechanism, multiple users in the same node also need to be connected which needs more threads listening on TCP connections. Therefore, we started to explore group communication protocols and frameworks in which one process only needs to listen on one broadcast address. We explored and compared the performance and implementation difficulty, and the results are shown in Table.3. Here one operation typically represents data send from one computing node and received by another computing node.

Table.3 Inter-Process Communication Performance Comparison

#operations	TCP	UDP Group	JGroups	Aeron
1000	20	110	210	12
10000	40	850	2100	37
100000	230	3600	9400	116

UDP Group Communication is achieved by UDP broadcast operations. Since UDP does not guarantee reliability, during my tests only about 22% the packets can be delivered with a single send and receive. JGroups [14] is a toolkit for reliable messaging. It can be used to create clusters whose nodes can send or broadcast messages to each other. However, from the testing results we can see it is very slow. Lastly, we tried Aeron [15], which is a framework originally included in MASS for messaging between different computing nodes. In Aeron, publishers can send message to specific channels and subscribers can subscribe to the channel so that they receive those messages, this can be seen as a group communication. Given its good performance compared to TCP and its nature of group communication, we finally decided to proceed with Aeron.

After all the exploration and tests, we finalize the overall system design, and it is shown in Fig.3. For a vertex insertion or update operation, a process writes this modified vertex information through to the shared file in “/dev/shm” and at the same time broadcast this vertex information to all other processes running on same computing node through Aeron. As for vertex access, simply read from the cache in process memory. The above procedure is for the scenario that the vertex we want to access/insert is in the same node as the user’s process. If the vertex is stored in another node, we use Aeron again for messaging between different computing nodes and let the remote node perform local operations and send back the results.

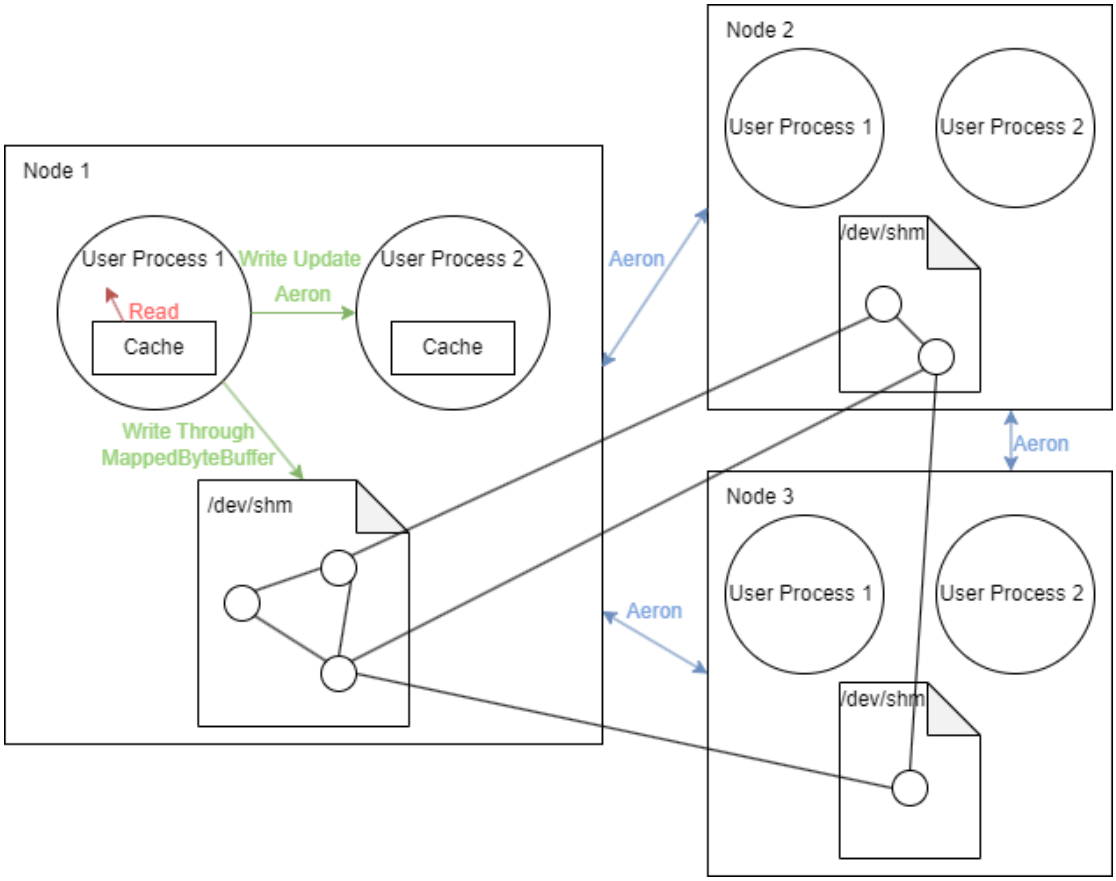

Fig.3 Finalized Distributed Shared Graph System Design

I compared again the performance of this system with Hazelcast, still focusing on the two basic operations, and the results are shown in Table.4 and Table.5. The four scenarios are defined as follows:

Local Insert: a process inserts/updates a vertex in its computing node, achieved by a write through to "/dev/shm" and a broadcast through Aeron.

Remote Insert: a process inserts/updates a vertex in other computing node, achieved by a message send to the remote node through Aeron, then the remote node performs Local Insert.

Local Access: a process accesses a vertex stored in its computing node, achieved by directly reading from the cache its process memory.

Remote Access: a process accesses a vertex stored in another computing node, achieved by a message sent to the remote node through Aeron, then the remote node performs Local Access and sends back the result through Aeron again.

Table.4 Vertex Insertion vs. Hazelcast

#insert operations	Local Insert	Remote Insert	Hazelcast
1000	177	189	300
10000	837	874	750
100000	4116	4232	3500

Table.5 Vertex Access vs. Hazelcast

#access operations	Local Access	Remote Access	Hazelcast
1000	0	24	100
10000	1	75	350
100000	9	241	2400

From the results, we can see that the insert operations are of similar performance with Hazelcast while access operations are much quicker. It should be noted that Hazelcast is a distributed key-value storage and the data here is recorded when only one Hazelcast instance is running, so it can be slower when multiple nodes running Hazelcast and the data is distributed. Also, this insertion operation can be further optimized by writing only to the process memory and use another thread to periodically write back to the shared memory.

Winter Quarter Plan

Based on the current progress of the project, I proposed the winter quarter plan as shown in Table.6.

Table.6 Winter Quarter Plan

Quarter	Week	Plan	Deliverables
Winter 2024	1-2	Implement the proposed DSG design	The proposed DSG implementation.
	3-4	Implement DSG using Hazelcast, functionality and performance tests of both implementations	A DSG implementation in Hazelcast, comparison of our implementation with Hazelcast.
	5-6	Preliminary MASS Places and GraphPlaces implementation on top of our DSG implementation	A preliminary version of MASS Places and GraphPlaces.
	7-8	Incremental implementation of MASS Places and GraphPlaces and performance tune-up.	An incremental version of MASS Places and GraphPlaces.
	9-10	Merging new MASS Places and GraphPlaces to the entire MASS library.	A new version of MASS with new implementation integrated.
	11	Write term report.	A term report submission to the committee.

Summary

Overall, during this quarter we encountered performance problems when comparing the initial version of DSG with Hazelcast but we managed to solve the problem and proposed a better designed system. Also, since for the first two weeks I read through the code base of MASS and I helped with the final project of CSS 534, I became more familiar with the MASS Java library. Although according to the original project plan I'm a little behind the schedule, I'm still confident that the goals can be met by devoting more effort to the project in the coming two quarters.

References

[1] "Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory", Accessed on: July 13, 2023. Available at: http://faculty.washington.edu/mfukuda/papers/biggraphs20.pdf
[2] "MapReduce", Accessed on: July 18, 2023. Available at: https://hadoop.apache.org/
[3] "Spark", Accessed on: July 18, 2023. Available at: https://spark.apache.org/
[4] "Storm", Accessed on: July 18, 2023. Available at: https://storm.apache.org/
[5] "Neo4j", Accessed on: July 18, 2023. Available at: https://neo4j.com/
[6] "Pipelining Graph Construction and Agent-based Computation over Distributed Memory", Accessed on: July 13, 2023. Available at: http://faculty.washington.edu/mfukuda/papers/biggraphs22.pdf
[7] "Hazelcast", Accessed on: July 18, 2023. Available at: https://hazelcast.com/
[8] "Redis", Accessed on: July 18, 2023. Available at: https://redis.com/
[9] "Oracle Coherence", Accessed on: July 18, 2023. Available at: https://www.oracle.com/java/coherence/
[10] "AWS SimSpace Weaver", Accessed on: July 18, 2023. Available at: https://aws.amazon.com/simspaceweaver/
[11] "Class MappedByteBuffer", Accessed on: July 18, 2023. Available at: https://docs.oracle.com/javase/8/docs/api/java/nio/MappedByteBuffer.html
[12] "Java Native Interface", Accessed on: December 11, 2023. Available at: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html
[13] *SHM_OPEN - Linux manual page*. Accessed on: December 11, 2023. Available at: https://man7.org/linux/man-pages/man3/shm_open.3.html
[14] *JGroups*. Accessed on: December 11, 2023. Available at: http://www.jgroups.org/
[15] *Aeron*. Accessed on: December 11, 2023. Available at: https://aeron.io/

Appendix

Since I haven't started working on the actual system, all my code this quarter are mainly testing the feasibility and performance. That's the reason I don't provide explanation on my code here. I uploaded all my testing code to mass_java_core bitbucket repository, under the branch "chrisma/develop". For code under directory "hazelcast", they can be compiled using the shell scripts. For code under directory "unixshm", they can be simply compiled by "javac *.java". As for the code under "unixshm/JNI", I created a README file in that directory which contains the step-by-step command to compile and run the code.

Link:

https://bitbucket.org/mass_library_developers/mass_java_core/src/ae13e6171709ca12bd2349438eeef14252756819/