The University of Washington Bothell
CSS-600: Independent Research

Distributed Graph Data Structures

Brian Luger
Professor: Dr. Munehiro Fukuda

10 June 2021

Introduction

The intent of this paper is to provide a summary of the work performed through the Spring quarter, 2021. The goal of this quarter was to research an approach to refactoring the GraphPlaces[1] feature within the MASS* core library and once found, begin iterating on the approach. The GraphPlaces library makes heavy use of a distributed graph data structure, hence the title of the paper. I addition to this, the team as a whole has decided to replace Hazelcast† with another messaging system. In doing so we will be removing the use of its distributed map implementation, and replacing it with a ground up implementation of which I was also tasked to tackle over the quarter.

While significant progress has been made relative to the number of credit hours committed to, the goal of completing the refactor of GraphPlaces and implementing the distributed map replacement was unfortunately not realized. Leaving the remaining work to be completed this upcoming Summer quarter, 2021. The Gantt‡ in Figure one below depicts the work that has been completed in contrast to what is still to be done. It is my belief that the reasons for being unable to complete the work as originally intended is due primarily to underestimating the scope involved in refactoring the existing GraphPlaces file support at a production level. Similarly, the effort required to implement a new distributed map that is reasonably performant for the DSLab§ use cases was considerably higher than originally expected. I will spend some time during the break leading into Summer quarter re-scoping the approach to completing this work in hopes of more accurately setting deadlines.

Another topic worth noting is performance measurements. A number of the features I worked on this quarter did not having working implementations when running over a cluster

*Multi-Agent Spatial Simulation
†https://hazelcast.com/products/in-memory-computing-platform/
‡https://en.wikipedia.org/wiki/Gantt$_chart$
§https://depts.washington.edu/dslab/

of nodes. Additionally, the refactor is not yet at a place where performance metrics would be meaningful. As such, benchmarks were not performed this quarter. Once complete, benchmarks will be done both in an attempt to compare and contrast what we can with the previous implementation, as well as to provide a baseline measurement and framework for future iterations. That said, in the coming sections the completed and remaining work will both be detailed.

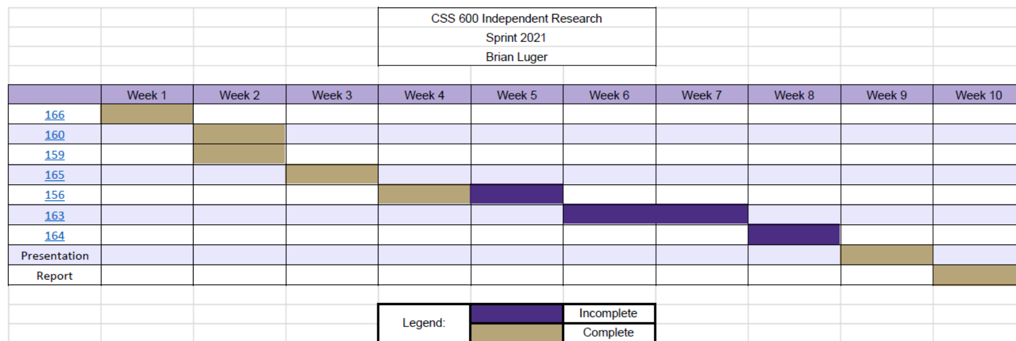

Figure 1: Gantt chart of work completed over the quarter.

Work Completed

This section will detail the work completed throughout the quarter, of which included implementing greater support for ad hoc CRUD[¶] operations over a cluster of nodes and a refactoring of the data structure and process with which vertices are stored and IDs are calculated. Additionally, a number of quality of life improvement were made. These included greater use of JavaDocs, more unit tests, and a greater separation of concerns between local and remote node logic. Lastly, wherever possible, refactors were made in a non-breaking fashion and proper use of deprecation annotations were utilized to provide intent to remove to any application developers that may be utilizing GraphPlaces now or in the future.

Vertex Storage

Prior to this quarter, vertices were stored in a multidimensional vector. As vertices were added, they would span across nodes in a cluster until the size of the vector hit a predetermined size set during initialization. Once this occurred, a new vector was created and placed on top of the previous one. See Figure two below for an example.

[¶]Create, Read, Update, and Delete

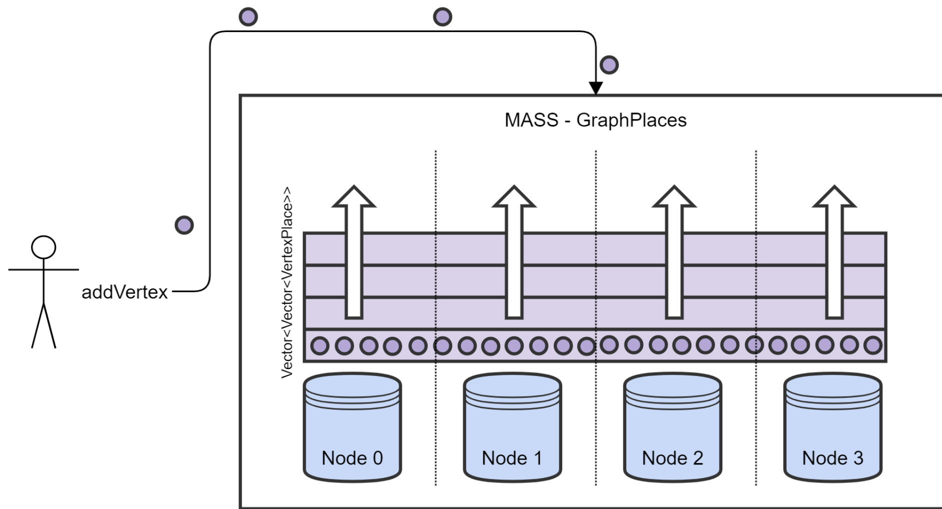

Figure 2: Vertex storage in GraphPlaces (MASS 1.3.0)

One issue with this approach had to do with dynamic creation of graphs. If a graph is initialized with a small initial size and grew dynamically to scale, it was possible for vertices to be stored in what effectively was column-major ordering. This occurred because the span of the nested vectors was set using the initial size. If it was small, new vectors would be created with each vertex, thus resulting in the structure growing vertically instead of horizontally.

Additionally, index calculation was a little complex, resulting in some minor bugs in determining the owner of a given vertex. To simplify the process, the storage of vertices was refactored to be a flat Vector that grew horizontally as more vertices were added. This is shown in Figure three below.

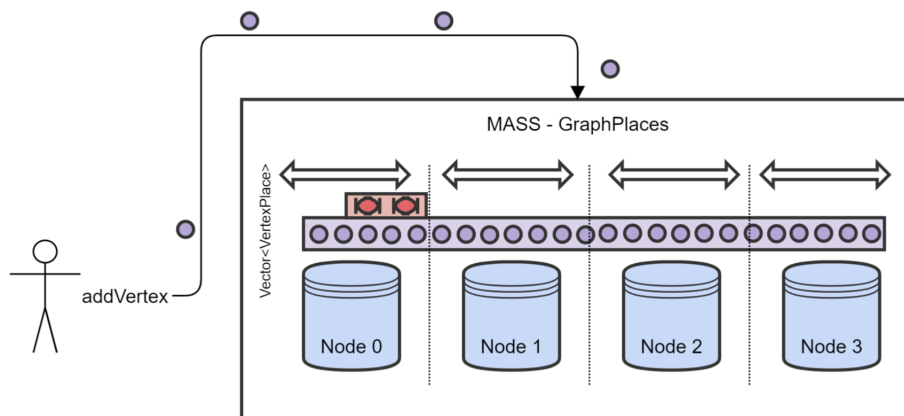

Figure 3: New vertex storage in GraphPlaces

As new vertices are added, the vector grows across the cluster. Removing vertices marks them as *empty* and queues for reuse by the next call to *addVertex*. This prevents the vector from shifting when removing vertices, an operation that would be quite expensive at scale. It also ensures that vertices are properly balanced across all nodes in the cluster, a feature that helps ensure parity with the previous implementation.

CRUD Operations

CRUD stands for *Create, Read, Update, and Delete* and in this context refers to the adding, removing, and retrieval of vertices and edges within GraphPlaces. Pre-existing implementations had a couple of issues that needed to be resolved prior to working on some of the usability features. Among these were *getVertex*, which did not work with remote nodes, and *removeVertex* which did not remove vertices from the property map, resulting in issues where previously removed vertices still appeared to exist. Retrieving these phantom vertices would result in an exception being thrown as they would no longer be in the vertex container. Lastly, a number of these operations had inconsistencies that prevented them from working as intended when using multi-node clusters.

Over the course of the quarter, I refactored all of the CRUD related methods to work off the new vertex container and performed manual testing to ensure that they all work as intended in both single and multi-node deployments. In addition to re-writing the methods, new remote method calls were added to *MProcess* to facilitate communication across nodes. Rather than introduce breaking changes to existing function, new ones were introduced and deprecation logs and annotations were added to the old ones to signal to users that support for them was ending.

While this is a major improvement for dynamic graph creation, graphs that are created from a file are still unable to take advantage of these changes. This is due to tight coupling between GraphPlaces and PlacesBase that primarily revolves around the constructors and generation of graphs from files. When initializing a graph this way, vertices do not utilize GraphPlaces containers, rather they use the containers in PlacesBase. This makes them completely disjoint from vertices that are added dynamically and prevents CRUD operations from within GraphPlaces from accessing them.

Quality of Life Improvements

At the beginning of the quarter, much of GraphPlaces lacked proper documentation and unit tests. As such, it was difficult at times to understand the reasoning behind previously made decisions in the class. This also resulted in a higher risk of regressions being introduced by the refactoring. To mitigate this, proprer JavaDoc comments were added to almost all methods in the class with best effort descriptions, and unit tests were added for existing methods. As work continues on GraphPlaces now and into next quarter, additional documentation and unit tests will continue to be added to further ensure the robustness of the library into the next release of MASS.

One area that is still lacking with no current approach to mitigation is testing in multi-node environments. It is my opinion that this may in part be because there does not currently exists a framework to help facilitate it. Standing up multiple nodes can require a lot of overhead and if the MASS library as a whole has any side effects or shared state, it can be difficult to stand up consistently for each test. This is unfortunate because it would seem that a number of the bugs that occur are a result of not testing on multi-node

deployments. Making this easier for developers to do should positively impact the quality of code being committed to the library.

Incomplete Work

This section will detail the work that was not completed, provide some insight into why that is, and briefly go over the remaining tasks.

File Support

This consists of implementing support for HIPPIE[‖], MATSIM[**], and a couple of couple of custom CSV[††] graph formats. Going into the quarter, it was my expectation that these could be simply ported, however upon further investigation it was determined that more effort would be needed. This is in part due to some oversights in some of the original implementations, tight coupling between GraphPlaces and PlacesBase, and lastly the newly introduced vertex container. As a result of these, most need to be fully re-implemented, rather than simply ported. Furthermore, it was decided to take these out of GraphPlaces in favor of implementing them within their own distinct classes that extend from it. This will make it easier to iterate on their implementations as doing so will not impact the underlying GraphPlaces logic. Greater support for other file versions could also be added without bloating the parent class. It is for these reasons that these tasks slipped into the upcoming quarter.

Having said that, some significant progress was made in adding support for HIPPIE graph data. Due to the nature of the file, it's not currently feasible to read discrete segments of the file, in parallel, across nodes. This is because vertices are not ordered and so it is very possible for multiple nodes to be consuming data for the same source node. Not having a means of sorting the file such that the reading node can take ownership of the vertices it consumes means that we would need to send vertex data to the owning node. Additionally, partial data that has yet to be completely consumed would need to be merged, thus resulting in significant overhead and complexity.

At alternative approach was determined that has the primary node parse the data and distribute vertices across the cluster. To limit message complexity, vertices are cached on the primary node and flushed once they've reached a reasonable size. To further increase performance and reduce the runtime and space complexity associated with caching vertices, the function calls are cached in place of the actual vertex objects. Lastly, to increase concurrency and overlap between IO operations, cached function calls are distributed across nodes in separate threads, allowing the primary node to continue working while it waits for other nodes to complete. A flow chart depicting the algorithm for doing this can be seen in Figure four below.

[‖] http://cbdm-01.zdv.uni-mainz.de/ mschaefer/hippie/information.php
[**] https://www.matsim.org/
[††] Comma Seperated Values

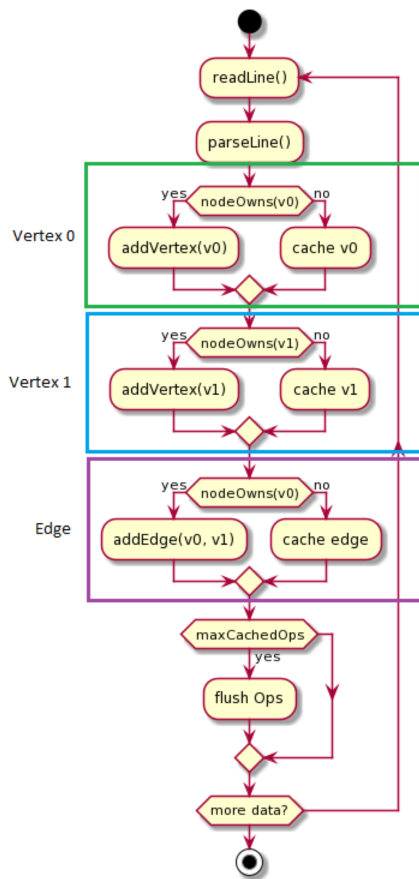

Figure 4: Function caching for HIPPIE file processing.

Distributed Map

The distributed map feature is intended to be a replacement for the Hazelcast distributed map. The approach decided at the beginning of the quarter still holds true today however, additional tasks have been identified that expand the scope of this feature, thus causing it to slip into next quarter. The approach to implementing this is to utilize a consistent hashing algorithm for determining key/value ownership in the map. Figure five below is an example of how consistent hashing works. In this diagram nodes A and B would be MASS nodes and the circles around the ring would be key/value pairs. The blue and red circles refer to their respectively colored nodes. Placement on the ring is based on the objects hash value and is uniformly distributed to ensure key are load balanced across nodes. The more node related items on the ring the greater the uniformity of keys across nodes.

This approach also facilitates a greater level elasticity as it reduces the number of key/-value pairs that need to be transferred when nodes are added and removed from the system. Though this isn't a feature that MASS currently can take advantage of, being able to scale simulations up or down dynamically may possibly be an ask in the future.

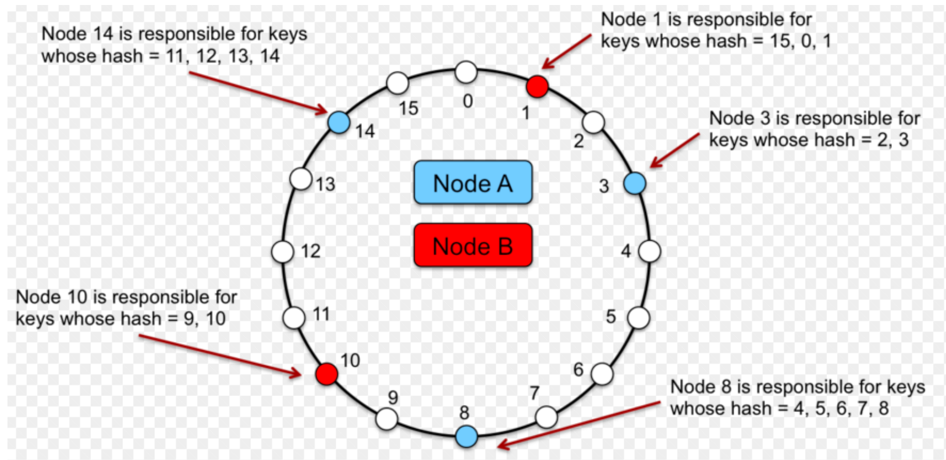

Figure 5: Example of a hash ring used in consistent hashing [2].

In addition to implementing consistent hashing for distribution of key/value pairs, integration with the new Aeron messaging system will also be required to ease communication between nodes. Support for asynchronous message passing between nodes will be needed to allow the map to be performant as sending and receiving in real time with *puts* and *gets*, on a single thread, would significantly impact performance and usability. Some of the open questions for this feature that have yet to find solutions have to do providing support for reverse lookups, as well as the ability for multiple core library utilities and/or applications to use the map at the same time without risk of key collisions.

Conclusion

To conclude, while a good amount of progress and work has been completed, there is still a sizable amount of work left to be done. I will continue iterating on it in the upcoming Summer quarter with the intent of completing the GraphPlaces refactor and implementing the distributed map replacement. I very much look forward to continuing this effort with the DSLab team under the guidance of Professor Munehiro Fukuda.

References

[1] "mass_library_developers / mass_java_core / src / main / java / edu / uw / bothell / css / dsl / MASS / GraphPlaces.java — Bitbucket." [Online]. Available: https://bitbucket.org/mass_library_developers/mass_java_core/src/1af4dd56d78f59285f3e774b700b36efb33277b7/src/main/java/edu/uw/bothell/css/dsl/MASS/GraphPlaces.java

[2] "Consistent Hashing Concepts - Databases, DHT," Oct. 2017. [Online]. Available: https://vitalflux.com/wtf-consistent-hashing-databases/