The University of Washington Bothell
CSS-600: Independent Research

Distributed Data Structures

Brian Luger
Professor: Dr. Munehiro Fukuda

20 August 2021

Introduction

The purpose of this paper is to provide a summary of the work performed over Summer quarter, 2021. This consisted of fully decoupling the GraphPlaces logic from its parent Places and PlacesBase classes, porting over support for MATSim[1] and HIPPIE[2] formatted graph files, as well as the UW-Bothell proprietary DSL and SAR formatted graph files. Lastly the callAll, exchangeAll, and agent migration functions across places and agents were all refactored to ensure they work properly with the new GraphPlaces data structures.

Background

GraphPlaces is an extension of the existing MASS Places object that adds capabilities for constructing graphs of places objects with which simulations can be run. Vertices within the graph are distributed across all nodes as shown in Figure 1 below. Vertices are distributed in a round robin fashion to ensure they are balanced across the entire cluster, and are persisted in a single vector on each node. Each vertex object within the graph contains a list of all its out edges, effectively making the container a distributed adjacency list.

[1] Multi-Agent Transport Simulation
[2] Human Integrated Protein-Protein Interaction rEference

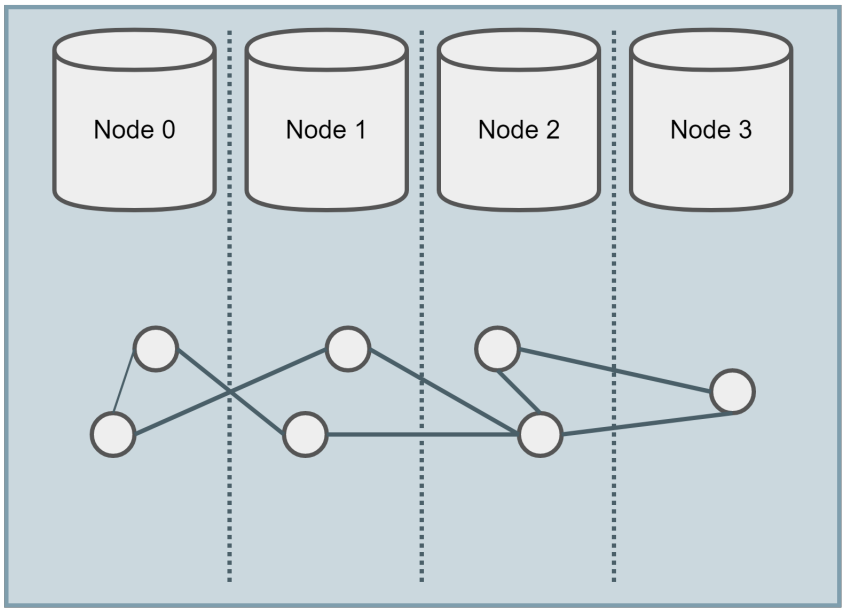

Figure 1: Visual representation of a distributed graph.

Once constructed, users have the option to load data in from a supported graph file format, or add vertices and edges manually through the appropriate *AddVertex* or *AddEdge* methods. Once constructed, users can further interact with a graph object by retrieving and/or removing vertices.

Agents can be deployed over the graph and contain logic for navigating and interacting with individual vertices. Unlike MASS Places, wherein agents migrate over neighboring cells in a matrix, agents constructed on GraphPlaces migrate over edges between vertices.

Motivation

The motivation for my work stems from a few high level issues. First and foremost was that vertex storage was inconsistent. If data was loaded from a file, vertices were stored in a Places matrix, however if they were added dynamically by the user then they were stored in a multi-dimensional vector within GraphPlaces. This inconsistency resulted in agents not being able to traverse the entire graph due to partitions that weren't representative of the actual data and should never have existed. Additionally, GraphPlaces functions utilized the GraphPlaces container and did not take into consideration that vertices could be stored in the Places matrix. This further exacerbated issues stemming from the inconsistent storage behavior.

Secondly, there were bugs that prevented GraphPlaces from operating as intended when using a multi-node cluster. For instance, if the size of the graph was not evenly divisible by the number of nodes in a cluster, exceptions would be thrown. Additionally, some functions like `getVertexPlace` and `removeVertex` did not work at all when using multiple nodes.

This was made worse by a lack of documentation describing the boundaries with which these functions should be expected work.

The last issue centered around quality of life complications. GraphPlaces was heavily coupled to its parent Places and PlacesBase objects, preventing developers from being able to extend it meaningfully without risk of breaking sibling classes. This also violated a number of the SOLID[3] principles. Documentation was also lacking, making it difficult to for developers to understand what GraphPlaces was doing. And lastly, there were very few tests for GraphPlaces and those that were written were marked disabled due to failures.

It was for these reasons I began looking into refactoring GraphPlaces.

Previous Work

I began working on GraphPlaces while volunteering with the distributed systems lab in Winter, 2021. Most of that time, however, was spent getting familiar with and understanding GraphPlaces and its role within MASS. Preliminary work was done on the approach to refactoring the existing vertex container but the real work didn't get started until Spring, 2021.

Throughout the Spring quarter my primary focus was two fold. First, I wanted to complete the refactor of the vertex container used in GraphPlaces such that it was less complex and more efficient with how vertices were persisted. This resulted in GraphPlaces moving from a container similar to that in Figure 2 to one that resembles that of Figure 3.

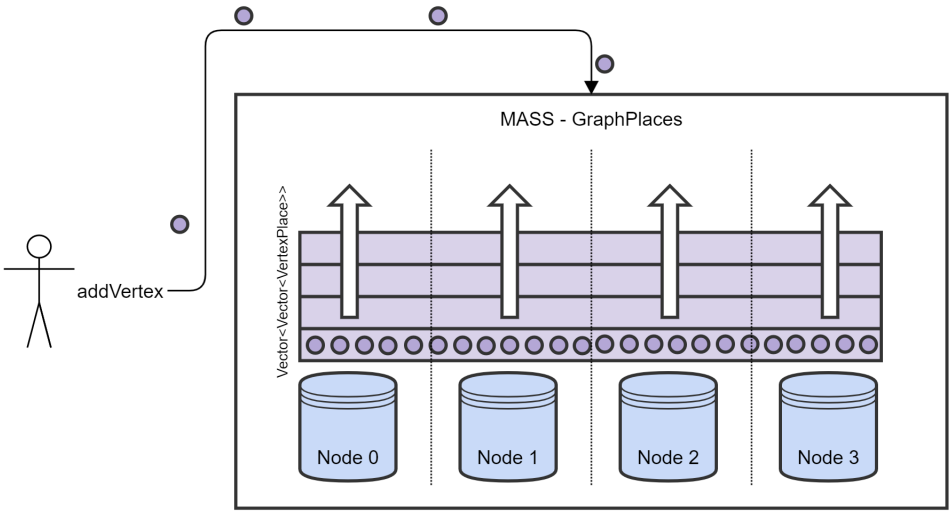

Figure 2: Vertex container and distribution as of MASS v1.3.0.

[3]https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

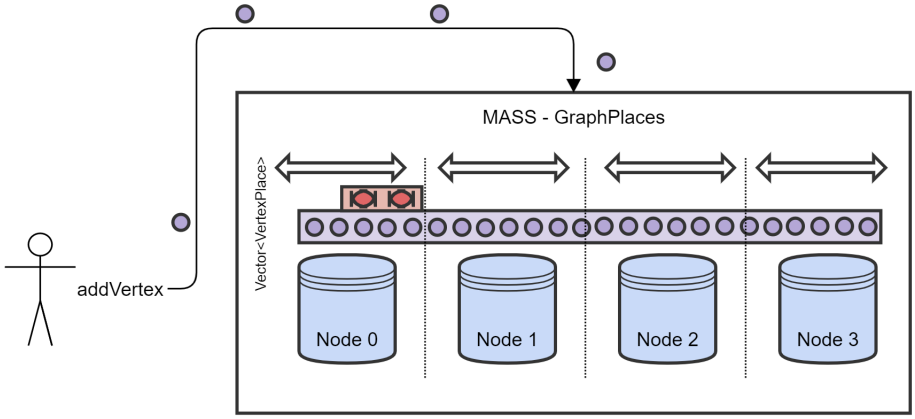

Figure 3: Current, refactored, vertex container and distribution.

It can be seen in Figure 2 that prior to this work, the vertex container closely resembled that of the Places matrix in that it utilized a multidimensional vector to store vertices. In this design, an initial graph size would be passed to the GraphPlaces constructor and used to determine the width of each vector. Adding vertices beyond this width would result in a new vector of the same width being created across the cluster to store the additional vertices.

In situations where the initial size is known at construction and just a couple of vertices are added post-construction, MASS would construct a new vector of the same sized width, resulting in unused memory. In cases where the size wasn't known at construction, single object vectors were created, in which added vertices would grow the container in column-major order, making traversals of the adjacency list inefficient due to cache thrashing.

In Figure 3 it can be seen that this was refactored such that it now uses a single vector that grows dynamically with the size of the graph. This prevents the need for users to specify the size of the graph at construction. Additionally, a removal queue (shown in Figure 3 as the red rectangle), was added to allow users to remove vertices without needing to shift elements within the container. Removed vertices are then added to the queue and recycled on the next call to `addVertex`.

The second item I worked on over the Spring quarter was the refactoring of existing GraphPlaces CRUD[4] functions such that they worked intuitively and regardless of the number of nodes in the cluster. This also included a cleaner separation of concerns between logic intended for local and remote nodes, as well as between global vertex IDs and named references stored in the distributed map.

File Support

File support in the previous GraphPlaces implementation was owned by the PlacesBase class and was responsible for much of the coupling between the two. Because it was owned by PlacesBase, it had no visibility into the vertex container within GraphPlaces resulting

[4]Create, Read, Update, and Delete

in the storage inconsistencies between vertices added by users and those added from files. To fix this, the logic responsible for importing graph data from each of the four supported formats had to be moved out of PlacesBase and into GraphPlaces. Additionally, it needed to be refactored to work within, and utilize the features of, GraphPlaces and not PlacesBase. This section details the work that went into doing this for each of the four formats: HIPPIE, MATSim, DSL, and SAR.

HIPPIE

HIPPIE stands for Human Integrated Protein to Protein Interaction Reference [1]. Graph data in this format is represented as a tab-delimited edge list. An excerpt of a HIPPIE graph file can be seen in Figure 4 below.

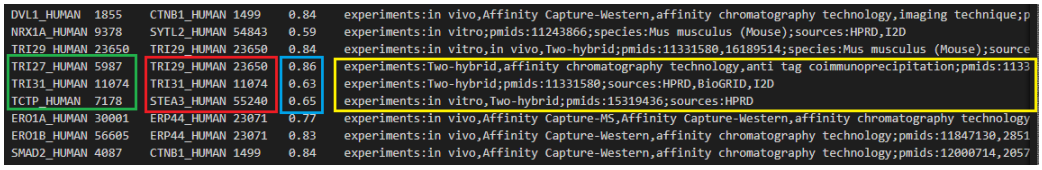

Figure 4: Excerpt from HIPPIE graph file.

Looking to Figure 4, the two values in the green rectangle are the protein key and ID respectively, and represent the source vertex for the edge. The red rectangle contains the interaction protein key and ID, and is the destination vertex. The blue rectangle contains the edge weight, and lastly, the yellow rectangle contains various metadata to be associated with the edge.

Prior support for HIPPIE files consisted strictly of creating source and destination vertices and linking them with an edge and the appropriate edge weight. The contextual data marked in yellow in the figure above was discarded. To improve on this, a new HIPPIE class was created within the *graph* package that extends from `GraphPlaces`. With it, are included `HIPPIEVertex` and `HIPPIEEdge` implementations that allow users to better represent HIPPIE data within MASS. Each `HIPPIEVertex` contains the protein key and ID, and each `HIPPIEEdge` contains the weight and all the provided contextual information from the graph file.

Providing this additional support for HIPPIE data enables users to construct agents that can leverage this information when performing simulations. Using it in decisions on which edges to traverse, when to terminate, and when to spawn new agents.

Implementation

The approach to implementing HIPPIE file support for GraphPlaces involved reading in the file data from the main node and distributing vertices to the appropriate owners as shown in Figure 5 below.

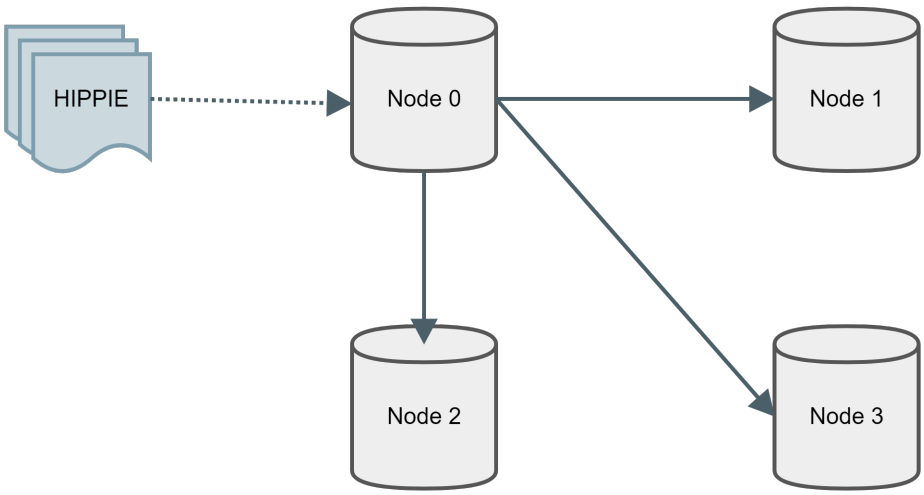

Figure 5: HIPPIE file content distribution.

In this approach, a HIPPIE file is read line-by-line by a single thread on the main node (Node 0). Each line is parsed, extracting the protein key and ID for each vertex, the edge weight, and the edge context. This data is then passed to the respective `addVertex` and `addEdge` methods to build the graph.

To minimize the number of messages required to communicate the created Vertex and Edge objects to remote nodes, the appropriate function calls for adding them, and thier arguments, are cached on the main node. Once the configured cache limit is reached, or we finish parsing the file, these functions and their arguments are communicated to the appropriate nodes where they are then constructed and persisted in memory.

It's worth noting that because the HIPPIE graph data is represented by an edge list, it would have been possible to have each node in the cluster process the data independently of one another. Consuming only the vertex and edge data for vertices it owns. However, because the data is not ordered and vertices are represented by named references (i.e., the protein key and ID), we are not able do this without a functioning distributed map to associate the named references with the correct vertex ID. A bug was found in the existing distributed map that restricted its use to only the main node, thus preventing this approach from being realized.

Benchmarks

Benchmarks of the HIPPIE file parsing algorithm were recorded with the largest HIPPIE file I could find at the time [2], which was approximately 49.53 MiB in size and contained 18,166 vertices. The time it took in seconds for MASS to parse this file was recorded as we scaled in the size of the cluster. Results can been in the chart below.

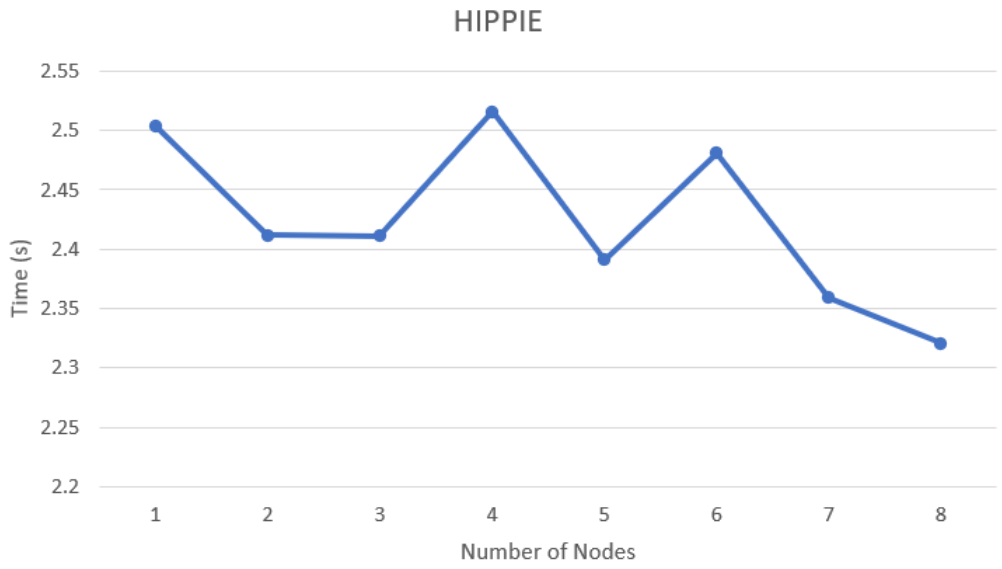

Figure 6: Execution time in seconds for parsing HIPPIE data.

Because the algorithm has to process the file data on a single node and distribute it across other nodes in the cluster, it would be expected that as more nodes get added to the cluster, and the size of the data grows, the time it took to process the file would increase. However, what we see in this chart is that as we add more nodes to the system the execution appears to trend downward, getting better.

It's worth noting that the difference between each data point is about a tenth of a second and these benchmarks were measured on a cluster shared by many other students. It's possible that external influences during the measurements are what resulted in it looking as though performance improved when in reality it remained steadily the same. It can also be argued that the size of the file is not sufficiently large enough to gather any meaningful data.

That said, there is some sense in the slight trend towards a faster execution time and that is that each of the worker nodes in the cluster are saved the overhead of having to read and parse the file themselves. In addition to this, they are sent just the data they need, meaning they aren't wasting resources discarding the data they do not. As the number of nodes in the cluster increased, the amount of data each node has to process decreases, thus resulting in a faster execution time for parsing the data. Data is processed concurrently between nodes with a barrier on the main node to wait for the others to complete.

Though this is possible, I still believe the amount of data used in the benchmark was not sufficient enough to be confident in these findings.

Large Scale Protein Data

An experiment was performed to see how CPU and Memory resources were utilized when ingesting and distributing a large scale graph file with MASS GraphPlaces across the entire 12 node CSSMPI cluster. To test this, the 69 GiB PP_Miner [3] data set was used and the execution time, CPU utilization, and memory consumption, was measured on the main

node as the file was parsed and distributed throughout the cluster. Additionally, a custom HIPPIE implementation was added that enabled each node in the cluster to process the file data independently of one another. To enable this, consistent hashing[5] was used to determine ownership of vertices based on their protein key. CPU and memory usage was sampled twice a second using the `ps`[6] UNIX command and outputted to a file where it could then be used to chart the results. The following script was used to automate this process:

```
1 while true; do (echo "%CPU %MEM ARGS $(date)" &&\
2     ps -e -o pcpu,pmem,args --sort=pcpu |\
3     grep bluger |\
4     cut -d" " -f1-5 |\
5     tail >> resource.data); sleep 0.5;\
6 done
```

In total, this file took 72 minutes to load into MASS. Each node in the CSSMPI cluster has a CPU with four cores, two hardware threads per core, and 16 GiB of memory. This enables MASS to utilize a maximum of eight threads in total for concurrency.

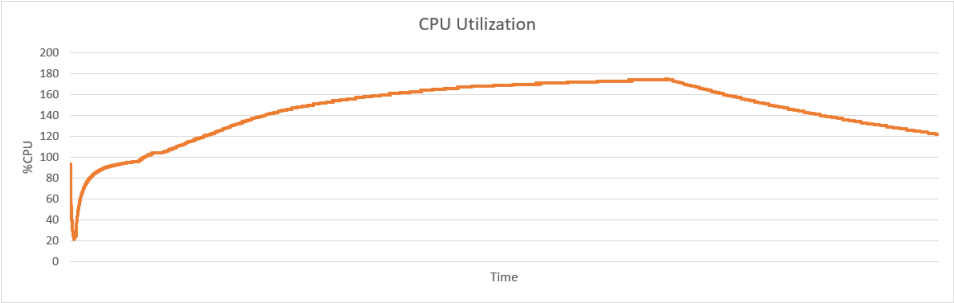

Figure 7: CPU usage of the main node while processing PP_Miner data set.

Looking at Figure 7 above, the first thing to note is that throughout the duration of processing this file a maximum of 178% of CPU was used. This isn't a complete surprise as the algorithm used to process this file, while distributed, was single threaded. However, the unused CPU resources indicate that there's potential for reducing our processing time should we be able to implement a concurrent algorithm for parsing the file.

Secondly, the inflection point about two-thirds of the way down the x-axis is also interesting. It's currently thought that this was the point in which the main node finished parsing the graph data and began waiting for the other nodes in the cluster to complete. However, if that's the case I would have expected to see a larger drop in CPU usage and it isn't currently clear why we did not.

[5] https://www.toptal.com/big-data/consistent-hashing
[6] process status

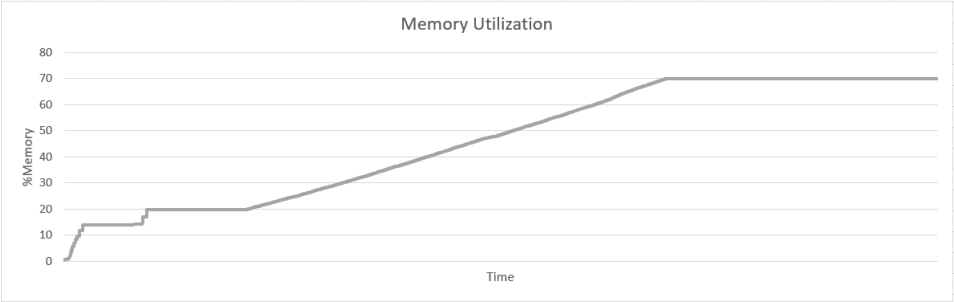

Figure 8: Memory usage of the main node while processing PP_Miner data set.

Figure 8 above charts the percent of memory usage over the period in which MASS was parsing the data set. It can be seen that we have slight steps up in memory usage as we begin consuming vertices owned by main. Following this is a linear increase in memory right up until it peaks at the area where the inflection point was seen in the CPU usage chart. This further reinforces the theory that the main node finished parsing the file at that time. In total, the main node used just shy of 70% of its system memory to consume its share of the PP_Miner data set. In monitoring the memory use of other nodes throughout this process, some reached as high as 92% usage of system memory.

The difference in memory usage is not believed to be an issue with balancing vertices across nodes, though this could be confirmed by increasing the number of points each node has on the hash ring. I believe the reason there is such a significant difference in memory usage between some nodes is because some nodes may contain vertices with significantly larger edge lists. This could be confirmed by counting the number of edges per node throughout the cluster, after the data has been consumed. Or possibly by using a similarly sized control dataset that gives an equal amount of edges to each vertex and then checking to see if the difference becomes smaller.

MATSim

MATSim stands for Multi-Agent Transport Simulation and it is an open-source framework for implementing large-scale agent-based transport simulations [4]. MATSim graph data comes formatted in XML and provides separate lists for vertices and edges. An excerpt of MATSim graph data can be seen in figures 9 and 10 below.

```
<node id="710347" x="4669462.0" y="5769109.5" type="0" />
<node id="710348" x="4657207.5" y="5734973.5" type="0" />
<node id="710349" x="4654386.0" y="5737502.0" type="0" />
<node id="710350" x="4658919.0" y="5734599.5" type="0" />
```

Figure 9: Vertex data in MATSim graph files.

```
<link id="200" from="1020" to="1002" length="639.0" freespeed="13.8888" capacity="11200.0" permlanes="1" oneway="1" origid="1207" />
<link id="201" from="1002" to="1027" length="547.0" freespeed="13.8888" capacity="22400.0" permlanes="2" oneway="1" origid="1208" />
<link id="202" from="1027" to="1002" length="547.0" freespeed="13.8888" capacity="22400.0" permlanes="2" oneway="1" origid="1208" />
<link id="203" from="1002" to="1003" length="207.0" freespeed="13.8888" capacity="22400.0" permlanes="2" oneway="1" origid="1209" />
<link id="204" from="1003" to="1002" length="207.0" freespeed="13.8888" capacity="22400.0" permlanes="2" oneway="1" origid="1209" />
```

Figure 10: Edge data in MATSim graph files.

As shown in Figure 9 above, vertices are represented by `node` tags and contain the vertex ID, x and y coordinates, and its type. Similarly shown in Figure 10, edges are represented by `link` tags and contain the edge ID, the source and destination vertex IDs (shown as `from` and `to` respectively in the XML data), and additional contextual information associated with the edge. Of which includes its `length`, `freespeed`, `capacity`, and `permlanes`, all of which could be considered an edge weight. And lastly its original ID and whether or not it's a directed edge (i.e., `oneway`).

Similarly to HIPPIE, original support for this file type was restricted to just the source and destination vertices. To improve upon this, a new `Matsim` class was created as a subclass of GraphPlaces and added to the graph package. Along with it included MatsimVertex, extended from VertexPlace, and MatsimEdge. MatsimVertex maintains the ID, x and y coordinates, and the node type. While MatsimEdge maintains all of its respective edge attributes.

Maintaining this contextual information enables developers to not only consume Matsim graph data, but to utilize it in transport-based simulations using MASS. Agents can be created to leverage this information in making decisions on how to traverse the graph, when to spawn more agents, or when to terminate.

Lastly, MATSim provides tag types for people, vehicles, and other agent-based models that could be represented in MASS through an extension to its own Agent class. It would not be a significant effort to add this functionality. Additionally, logic could be added on top of this to utilize the plan tags within MATSim for determining agent routes through the graph.

Implementation

Implementation for support of the MATSim graph XML file format follows the same approach as the HIPPIE implementation and for the same reasons. While MATSim does have a file format that would enable parsing it independently across the cluster, we are unable to leverage it due to a bug in the distributed map implementation that prevents its use in a distributed fashion. This would be needed to associate the named ID of each vertex with its global index within MASS.

Because this is currently not doable, we instead process the file on the main node, caching calls to addVertex and addEdge, and shipping the cached function calls to their respective nodes when we finish parsing the file or a cache limit is encountered.

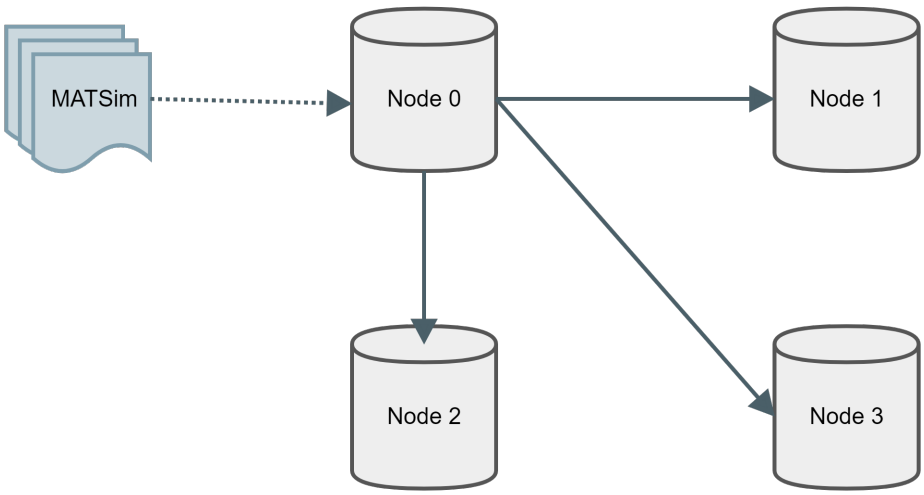

Figure 11: MATSim file content distribution.

Benchmarks

Benchmarks of the MATSim file parsing algorithm were recorded with the largest MATSim data set I could find at the time [5]. This data set was labeled as "benchmark" giving some indication of its intended purpose. However, due to its size, it's more likely that agent navigation was what was intended to be benchmarked and not necessarily graph scalability. The data set contains 11,566 vertices with a file size of 4.67 MiB.

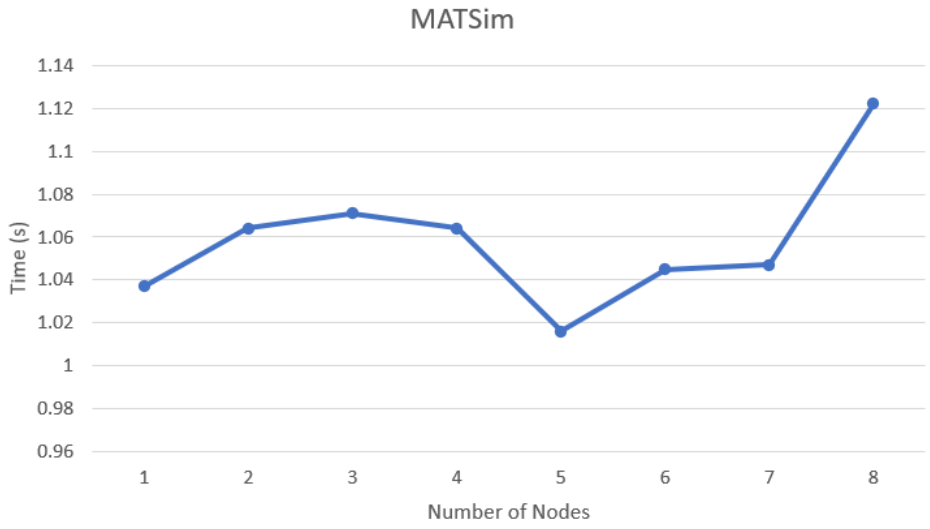

Figure 12: Execution time in seconds for parsing MATSim data.

It can be seen in Figure 12 above that execution time is somewhat all over the place. It looks to be increasing between one and three nodes, before dropping between three and five, and rising yet again between five and eight. Looking carefully at the difference in execution time between each point, it's in the order of hundredths of a second. It's because of this that I believe that the data simply wasn't large enough for us to glean any meaningful information from these metrics. However, because it follows the same algorithm as the HIPPIE parser, I expect its computational and message complexity to also be similar with slight constant variations based on the overhead associated with parsing the file content.

In the future, it may be beneficial to reach out to the MATSim maintainers to ask how best to generate sizable MATSim graphs for use with scalability testing, or possibly just create a graph generator of our own.

Proprietary File Formats

In addition to HIPPIE and MATSim files, two file formats proprietary to UW-Bothell were also ported to the new GraphPlaces implementation. These are the DSL and SAR file formats.

DSL

DSL stands for Distributed Systems Lab and it is the graph file format outputted by the GraphGen application used in Professor Fukuda's CSS-534 class for generating graphs. An excerpt of the file can be seen in Figure 13 below.

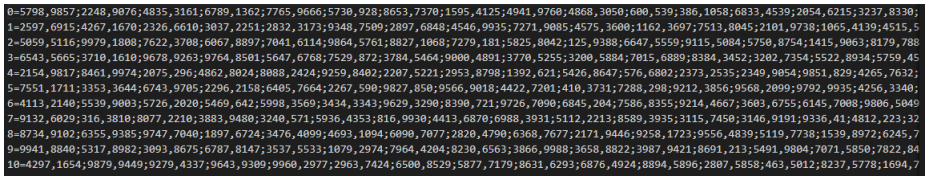

Figure 13: Excerpt from DSL formatted graph file.

As shown in the figure above, the format is representative of an adjacency list in that there is a single vertex per line, followed by an = sign and a semicolon delimited list of out-edges and their associated weights. In previous releases of MASS this format was labeled as CSV, however, to avoid confusion with actual CSV[7] formatted files, it was renamed to DSL.

Implementation

Unlike both HIPPIE and MATSim formatted graph data, DSL graph data is ordered with vertex IDs that are monotonically increasing by a value of one. Because of this, the vertex IDs within the file can be used as global indices for their associated vertex and the file as a whole can be parsed independently by each node in the cluster. Additionally, there is no contextual information associated with the vertex or edge that would require the use of a distributed map for storing references. It is because of this that DSL files do not need to be

[7] Comma Seperated Values

processed singularly on the main node and distributed to the workers. Instead each node in the cluster can parse the file and consume only the data it is responsible for. A high level diagram of this can be seen in Figure 14 below.

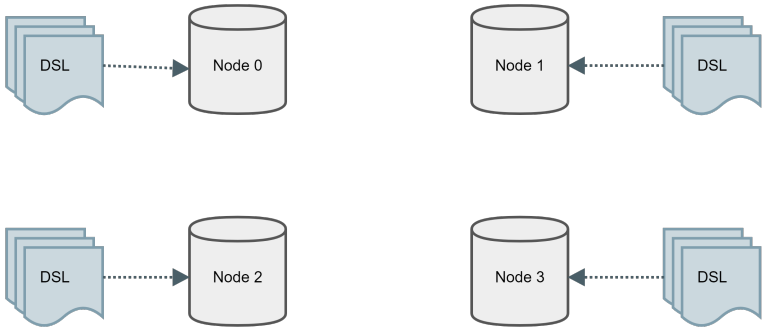

Figure 14: Parsing of DSL files in a cluster.

Furthermore, because the file is a purely raw representation of a graph with no contextual information, it made sense to add it as a format supported natively within GraphPlaces so that it could be used to generate basic graphs within it and any of its sub-classes.

Benchmarks

With regard to benchmarks, we had the privilege of being able to generate our own graphs and as such were not limited in the same way we were when benchmarking HIPPIE and MATSim. Taking advantage of this, a graph of 750K vertices, with 100 edges per vertex, and a file size of 1.03 GiB was generated to assist in testing how well it scales.

Looking at Figure 15 below, it can be clearly seen that as the number of nodes in the cluster increases, the time it takes to parse the file is significantly reduced. From approximately 36 seconds with a single node, to starting to plateau around 8 seconds with 8 nodes. Further improvements to execution time could likely be had by utilizing parallel file IO and additional threads for reading file content concurrently. It would also be interesting to measure CPU and memory use while processing DSL files to gain a better understanding of their resource utilization.

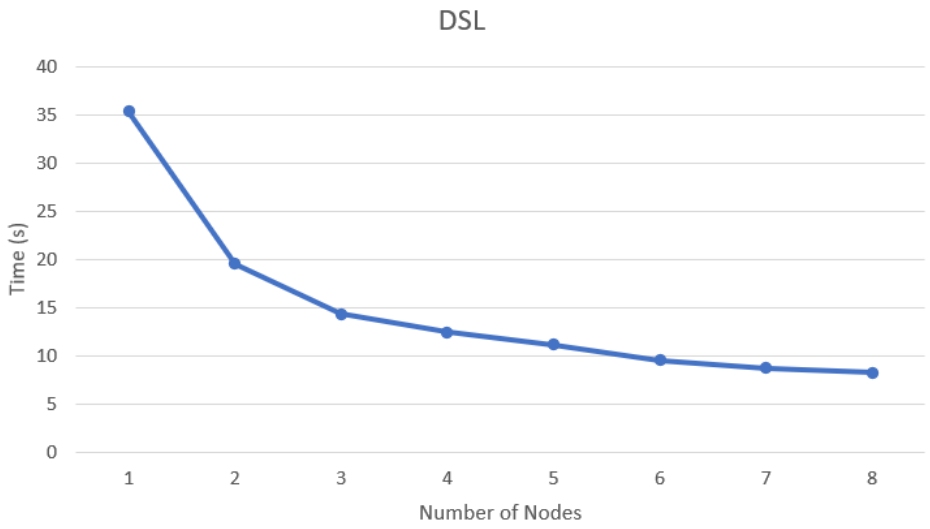

Figure 15: Execution time in seconds for parsing DSL data.

SAR

SAR is a novel, proprietary, graph file format also from the distributed systems lab at UW-Bothell. It was implemented by Justin Gilroy as a parallel I/O file format for graph data and named after the group members whom would be benefiting from its design. [6] An excerpt of the file format can be seen in Figure 16 below.

```
1   9,9,9,9,9,9,9,9,9,9
2   1  2  3  4  5  6  7  8  9
3   0  2  3  4  5  6  7  8  9
4   0  1  3  4  5  6  7  8  9
5   0  1  2  4  5  6  7  8  9
6   0  1  2  3  5  6  7  8  9
7   0  1  2  3  4  6  7  8  9
8   0  1  2  3  4  5  7  8  9
9   0  1  2  3  4  5  6  8  9
10  0  1  2  3  4  5  6  7  9
11  0  1  2  3  4  5  6  7  8
```

Figure 16: Excerpt from an SAR formatted graph file.

The first row of an SAR formatted graph file is comma delimited and indicates the number of neighbors belonging to the vertex in the respective position in the row. For example, the first value represents the number of neighbors in vertex 0, the second is associated with vertex 1, third, vertex 2, etc...

Immediately following this header information, beginning on line two, is the neighbor data, or adjacency list, associated with each vertex. The owning vertex ID is determined

by the line number. Line 2 belongs to vertex 0, line 3 to vertex 1, continuing to line N and vertex N-2.

In Justin's whitepaper [6] he makes the claim that using this format, vertex offsets into the file can be determined by summing the count of neighbors in the header up to the vertex with which the offset is needed. If accurate, this would allow nodes to precisely partition the graph data across the cluster, allowing each to read only its share of the file. Thus significantly reducing the time it takes to ingest the graph data.

He provides an example in Appendix D of his whitepaper demonstrating that with a header of `9,9,9,9,9,9`, the offset of vertex 3 can be calculated by summing the first three values in the header, multiplying by 10, and adding 3 (9 * 3 * 10 + 3 = 273). It is not clear from the whitepaper where the value 10 comes from though I speculate this is the number of newline characters that are introduced in the file and is equal to the number vertices therein. However, I'm completely unsure why 3 is being added.

Unfortunately, this formula does not work for its stated purpose. The first issue is that the size of the separators (the commas and tabs) are left unaccounted for. This would change the byte offset by up to a factor of two, assuming that vertex ID remains a single digit.

The second, more notable issue with the file format, is that it assumes single digit vertex IDs. It cannot be extended beyond ten vertices because then the offset could not be accurately determined by just the header data. This is because you could no longer assume all vertices have neighbors that can be represented by a single digit number and you couldn't confirm which edge lists contain them without parsing up to the vertices for which you want the offset. This is further exacerbated when you move to 100, 1000+ vertices.

Implementation

Implementation for support of this file format does not utilize Justin's proposed formula for the reasons stated above. Instead, it ignores the header line entirely and parses the adjacency list line by line. Associating vertices with IDs based on their line number. Fortunately, like the DSL file format, this format contains ordered vertex data with IDs that monotonically increase by one. Meaning that we can utilize the same approach to parsing the SAR file that we do with DSL files and it should get roughly the same performance benefit from scaling the number of nodes in the cluster. However, it also means that we are not able to partition the file data across nodes such that each node only has to parse a fraction of the data.

Though not implemented here, I think that if we wanted to partition file data evenly across all nodes we wouldn't need to do much else than divide the file size by the number of nodes in the cluster and then seek to the next newline. This would not uniformly distribute vertices across all nodes, however it should uniformly distribute memory use across nodes. This is because the length of edge lists aren't uniform. It's possible for a single vertex to contain the majority of edges and thus result in an imbalance of memory usage across nodes. We saw this demonstrated with the large PP_Miner data set. However, since we would be splitting the file based on size, and not vertex count, nodes would get only the vertices that encompass their share of memory.

Benchmarks

Unfortunately, there are no benchmarks for SAR files as I could not find an SAR graph generator and lacked the time to create my own.

CallAll & ExchangeAll

Aside from porting file formats over to GraphPlaces and decoupling it from its parent classes, the `callAll` and `exchangeAll` methods were also refactored to work properly. Fortunately, for callAll this required very little effort. Because PlacesBase simply iterates through all the places and calls their `callMethod` function with the appropriate arguments, all that needed to really change was to ensure that GraphPlaces was utilizing its vertex container for this and not the matrix within PlacesBase.

ExchangeAll however, required significantly more effort to implement as we couldn't simply update the original implementation. It was built to strictly work with matrices. To mitigate this, the `exchangeAll` method was overridden within GraphPlaces to allow it to run over a Graph. Additionally, the MASSBase Exchange Helper was needed for sending and receiving messages between nodes since MProcess could not be.

ExchangeHandler

To manage the receipt of messages from other nodes, an ExchangeHandler was implemented that constructs a thread for each node in the cluster as shown in Figure 17 below.

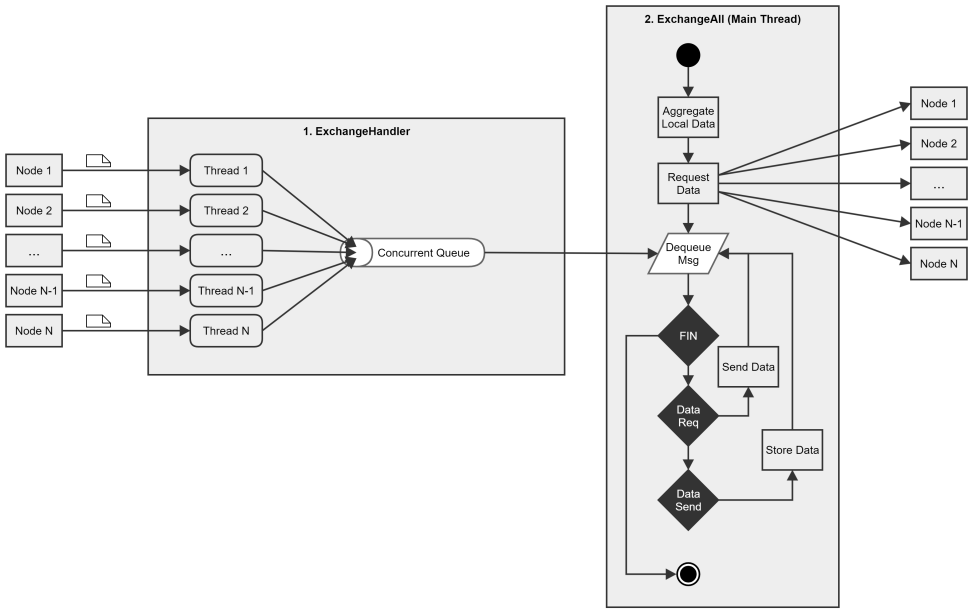

Figure 17: Flowchart for exchangeAll.

Upon receiving messages from nodes, the handler enqueues them into a concurrent linked blocking queue, where they reside until the main thread dequeues them for processing. Use of this queue enables each thread to place items on it without the worry of race conditions or need to manage locks. Additionally, because it is a blocking queue, the main thread can chose to block until the queue is no longer empty. This prevents the need to busy-wait by polling to see if new messages have arrived, or to implement the monitor pattern on an

already concurrent queue that doesn't need to be synchronized. Lastly, the exchange handler tracks the number of currently running threads through the use of an atomic counter. Before the last thread exits, it sends a poison pill[8] through the queue to notify the consumer that it will no longer be receiving messages and can safely discard the handler.

This is then utilized by the exchangeAll method to assist with distributing values returned as a result of calling the callAll method on each vertex place.

ExchangeAll

The ExchangeAll method, as shown in Figure 17 above, begins by first calling callAll on all vertex place objects owned by it and caching the aggregated return values. While doing this, it also enumerates each vertex neighbor and groups them by the nodes that own them. This forms a list of all of the vertices for which this node needs return values for but does not own.

Once this is done, requests are made via the ExchangeHelper to each remote node, asking for the return values of the vertices it does not own but that have in-edges from vertices it does. For example, if we have the edge `<A,B>` and node 0 owns A, and node 1 owns B. Node 0 will need to request the return value from Node 1's execution of callAll on vertex B.

Once these requests are made, the thread performing the exchangeAll method will begin processing messages from the exchange handler. If it receives a data request message, asking for return values for vertices it owns, it aggregates the appropriate values for the calling node and sends them to it. Alternatively, if it receives a data send message, it is receiving the return values that it previously requested from other nodes. Once all nodes have received the return values they need to process the exchangeAll, and have sent out the data other nodes need, a FIN message will come through the queue notifying the listener that this is done and it can now process the data for the exchangeAll call.

At this point, each node has all the results from calling the callAll method on every neighboring vertex. The local vertices are then iterated over once more, this time to store the results of the exchangeAll call. Once this is done, the method returns, and the call is complete.

Agent Migration

There were two major issues that were solved with Agent Migration. The first was that the AgentsBase class was utilizing old GraphPlaces methods and making assumptions about how global vertex indices were generated. This was updated to appropriately calculate indices and to call the correct methods. This and some minor refactoring was enough to get callAll functioning again across all agents associated with a GraphPlaces object.

The second issue was with the `map` function of the Agent class. This function determines whether an Agent should be instantiated based on the initial population and formulas used with the Places matrix, and not a graph. To resolve this issue, a `GraphAgent` class was introduced that extends from Agent and overrides the `map` function with one that appropriately maps the initial agent population over the graph. This function works properly in both single and multi-node deployments, ensuring that agents are evenly distributed in both instances.

[8] https://java-design-patterns.com/patterns/poison-pill/

Having made those changes, agent migration is now working as intended with GraphPlaces. That said, there are still some design issues with AgentsBase in the way of coupling to GraphPlaces. This, like Places and PlacesBase, violates the Open/Closed principle in that a class should be open for extension but closed for modification. Instead of modifying AgentsBase to be aware of GraphPlaces, we should consider extending it with a GraphAgents subclass that provides overrides and constructors for its respective functionality.

Conclusion

To conclude, the remaining feature work for GraphPlaces was completed this quarter. However, it could still use a little more documentation and better testing to prevent regressions and bugs from creeping in. I'll continue adding these over time as more tests, benchmarks, and applications are written to make use of GraphPlaces.

One initial goal of this quarter that wasn't realized was the implementation of the distributed map. That said, the approach to using consistent hashing was tested with the large PP_Miner data set. The work done there should help influence its use within the distributed map. Moving on to next quarter, I'll need to sync with Mathew Sell regarding his work on the Aeron messaging system so we can coordinate the development of the distributed map. Additionally, I'll be doing some preliminary research into parallel I/O in preparation for beginning my thesis work in the Spring.

References

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

[2] "HIPPIE Data Sets." [Online]. Available: http://cbdm-01.zdv.uni-mainz.de/~mschaefer/hippie/download.php

[3] "BioSNAP: Network datasets: Species-specific protein-protein association networks." [Online]. Available: https://snap.stanford.edu/biodata/datasets/10028/10028-PP-Miner.html

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

[5] "MATSim Data Set." [Online]. Available: https://matsim.org/files/benchmark/

[6] "Dynamic Graph Construction and Maintenance." [Online]. Available: http://depts.washington.edu/dslab/MASS/reports/JustinGilroy_whitepaper.pdf