MASS C++ Enhancement: Porting Parallel I/O and Graph from MASS Java to MASS C++

Prepared by: Elias Alabssie faculty advisor: Professor Fukuda, Munehiro

MASS Library Overview

MASS is a parallel-computing library for multi-agent and spatial simulation over a cluster of computing nodes. Agents and Places are the building blocks for MASS library. Places hold user defined Place object in a multi-dimensional grid with machine independent indices. Agents are execution instances that can reside inside places, migrate to other places either locally or on a remote machine in the simulation space. The scalability aspect of MASS is achieved by sharing the workload among the cluster nodes. MASS distributes and parallelizes the workload using native sharing algorithm unless otherwise the user specifies a different algorithm.

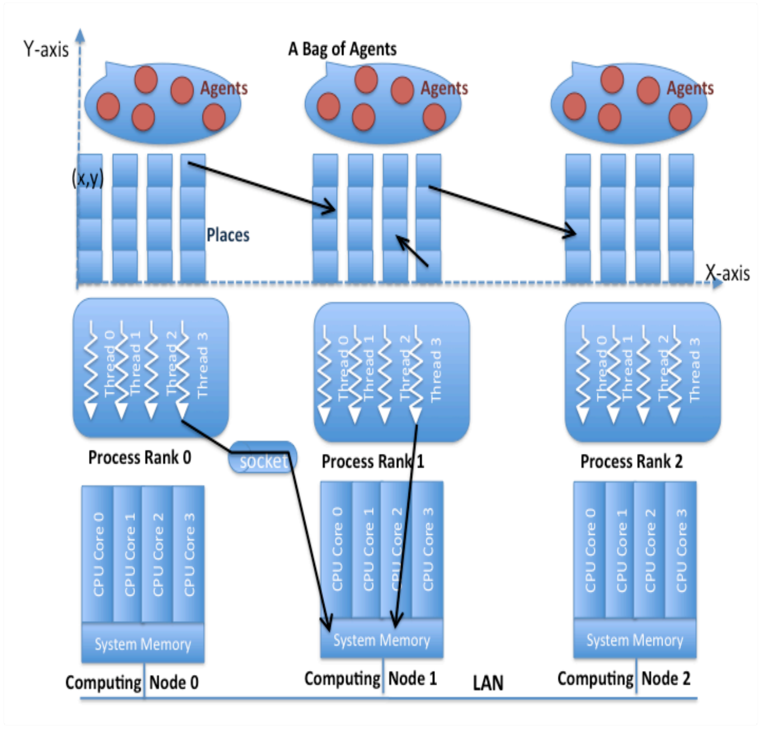

Figure 1. Agents, Places, cluster nodes (Processes) and parallelizing of places. [2]

Project Goals

MASS provides implementation in three computer languages: Java, C++ and CUDA. The MASS Java library has more features and functionalities than the MASS C++ library. To narrow this gap, I was assigned to port the MASS Java Parallel IO and the graph features from the MASS Java library into MASSC++ library.

Higher Level Class Design

Much of the design and the implementations are similar to the MASS Java library implementations. Generally, I used the same names for classes and their APIs in an attempt to save a MASS C++ library user from learning new APIs when they moved from MASS Java to MASS C++. However, I made changes to the class designs when I found advantages to do so. In addition, I've changed some of the method names to make it more intuitive for the library users and other developer who will maintain the library in future works.

One of the design changes I've made is, I've implemented a separate class that reads different input files (MATsim, HIPPIE, CSV, etc.) to get the vertices from input files. In MASS java, this functionality is embedded into 'PlacesBase' and 'VertexPlace' classes which I believe is a redundancy. This feature of a separate class implementation for parsing files helps to modify the file parsing logic without changing other implementations. Additional input file format parsing can be added to this class without modifying subsequent classes which uses the file parsing service.

The other major difference between MASS Java and MASS C++ is the way the master node propagates the map of 'vertices' and their Id (map<string vertex, int>) to the remote computing nodes after reading vertices and their relative position in the input file. MASS Java uses a 'HazelCast" library which automatically propagates any data structures from the master node into every other node in the cluster.

I abandoned this library for MASS C++ because it requires running the Cluster Manager component, which is a java application. A C++ application can access data structures from HazelCast cluster only as a client. Which intern means the MASS C++ library user have to have JVM installed and the HazelCast cluster manager program running before running the MASS C++. Requiring JVM installed doesn't seem too much to ask now a days, however requiring HazelCast cluster manager to run is a bit of inconvenient.

Due to this inconvenience of using HazelCast, the MASS C++ library uses the existing messaging infrastructure to propagate vertices from the master node to the remote computing peers. The map of vertices along with their global Ids are serialized and the remote peers will deserialize the message and build the map of <vertices, global Id> objects back.
On top of this, there are other smaller classes in MASS Java that I decided to squeezed them into other classes.

Apart from the differences discussed above, there is higher congruencies between MASS java and MASS C++ libraries. Here under is the partial UML diagram for MASS C++ library classes.

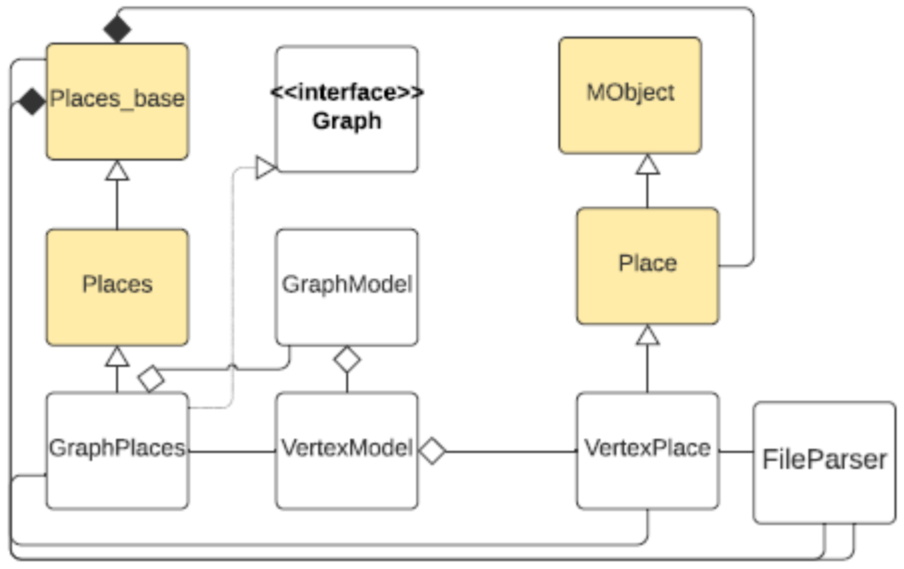

Figure 2. UML diagram of partial MASS C++ library classes

Class Implementation Details

1. Porting Parallel I/O

Parallel IO is a concurrent request from multiple processes of a parallel program for data stored in a file [1]. I/O performance, rather than CPU performance, is the key limiting factor in the performance of a computer. Specially in a distributed system, as multiple CPUs are working together, the IO overhead becomes the additive IO bottleneck of the individual computing nodes.

One solution to mitigate this IO bottleneck is enabling processes to access a file parallelly (hence the name parallel IO). Since MASS is meant to solve big data computation parallelly, having parallel I/O feature enhances the overall computation performance

MASS C++ implements the parallel I/O feature in the 'File' class and in the 'Place' class. The File class is an abstract class that defines interfaces for opening, deleting, and closing files. In addition, the class implements the algorithm of slicing the total file size across computing nodes in the cluster and among Places on the local machine.

Here, the same slicing algorithm used as in MASS Java. First, the total file size is shared for number of computing nodes. The share of each node's file size will be further sliced for the number of Place objects living on that node.

# slices = total file size/# cluster nodes ……………………………………………………………… [1]
# slice for each Place = #slices/#Places on local node ………………………………………….. [2]

For example, for a cluster of four nodes and 1000 bytes file size, and 100 Places on each node, the slicing algorithm works like this,

#slice for each node = 1000 bytes/4 = 2500 bytes ………………… [formula 1 above]
#slices for each place = 2500 bytes/100 = 25 bytes. …………….. [formula 2 above]

Therefore, according to the above slicing algorithm, each place is responsible reading 25 bytes worth of data on each computing node. If the number of bytes is not evenly divisible by the number of nodes, the node with a higher rank will take all the remainders in addition to its share.

If the file size is less than the number of nodes (unlikely scenario), only the master node has access to the file. All other nodes can not access the file via parallel I/O APIs. The programmer should use other C++ fstream methods to access the file. On the other hand, if the number of Places for a particular node is greater than the slice of N bytes file (another unlikely scenario), only the first 'N' place can access the file.

For example, if a node has 100 places and get 25 bytes file, only the first 25 places can access the file. The file is locked for the rest of the places. Again, the programmer has to use other C++ fstream routines to access this file from unauthorized 'place'. I chose not to use synchronization enforcement in these scenarios to avoid performance penalties. Denying access seems more performant.

A concrete class for parallel I/O should inherit the abstract class 'File' to benefit from these implementations. Currently, the 'TxtFile' class is implemented that works for a text file types.

Porting Graph Features

Porting the graph features from MASS Java was the bulk of this project. It took me a significant amount of time and effort.

1.1 Parsing files and Building Map of Vertices

To build the graph, first the vertices has to be extracted form the input file. The input file can be of different type. There are different file parsing implementations inside the

'FileParser' class depending on the file type. The file type should be passed as an instance of 'FILE_TYPE_ENUMS'. Then the class will call the appropriate parsing routine depending on the Enum type to read the vertices from.

```
enum FILE_TYPE_ENUMS {CSV, HIPPIE, MATSim, TXT, FILEGEN};
```

Figure 3. enum of file types.

1.1.1 File Types

As mentioned above, there are different file types to read vertices from. The enum above lists five file types. If additional file types needed, add the file type in the enum and implement the parsing algorithm. Every other logic will work fine without modification (no need of modification inside the 'Places_base' and 'VertexPlace' classes).

2.1.1.1 HIPPIE: HIPPIE (Human Integrated Protein Protein Interaction rEference) is a file type that contains the weight of protein -protein interaction. A core component of HIPPIE is the confidence scoring of interactions based on the amount and reliability of evidence supporting the interaction. This score is calculated as a weighted sum of the number of studies in which an interaction was detected, the number and quality of experimental techniques used to measure an interaction and the number of non-human organisms in which an interaction was reproduced [4].

HIPPIE is a tab separated file type. In this file type, individual proteins are vertices and the interaction weight between proteins is the weight of the edge that links the interaction.

```
……            ….        ….           …     ..    …..
SNX12_HUMAN   29934     FYN_HUMAN    2534  0.63  experiments:peptide
              array;pmids:17474147;sources:IntAct,MINT
ELK3_HUMAN    2004  GRB2_HUMAN   2885  0.63  experiments:peptide
              array;pmids:17474147;sources:IntAct,I2D
GRB2_HUMAN    2885  PRIC3_HUMAN  4007  0.63  experiments:peptide
              array;pmids:17474147;sources:IntAct,I2D
GRB2_HUMAN    2885  SP1_HUMAN 6667  0.63  experiments:peptide
              array;pmids:17474147;sources:IntAct
SNX12_HUMAN   29934     ELK3_HUMAN   2004  0.60  experiments:peptide
              array;pmids:17474147;sources:IntAct
```

Figure 4. Sample HIPPIE file format [4].

In figure 4, the yellow colored part at the beginning of each line are the primary proteins and the red colored once are also proteins that interact with the primary protein. The blue colored number are Ids for the proteins. The orange colored double number is the weight of the interaction. The rest of data is a meta data that the MASS C++ graph use case is not interested in. Each part except parts after the interaction weight, is separated by a tab.

In the above HIPPIE sample file, GRB2_HUMAN (Id = 2885) has two neighbors, SP1_HUMAN (Id = 6667) and PRIC3_HUMAN (Id =4007) with weight of 0.63 for both interactions.

Parsing the sample file above, results the following graph.

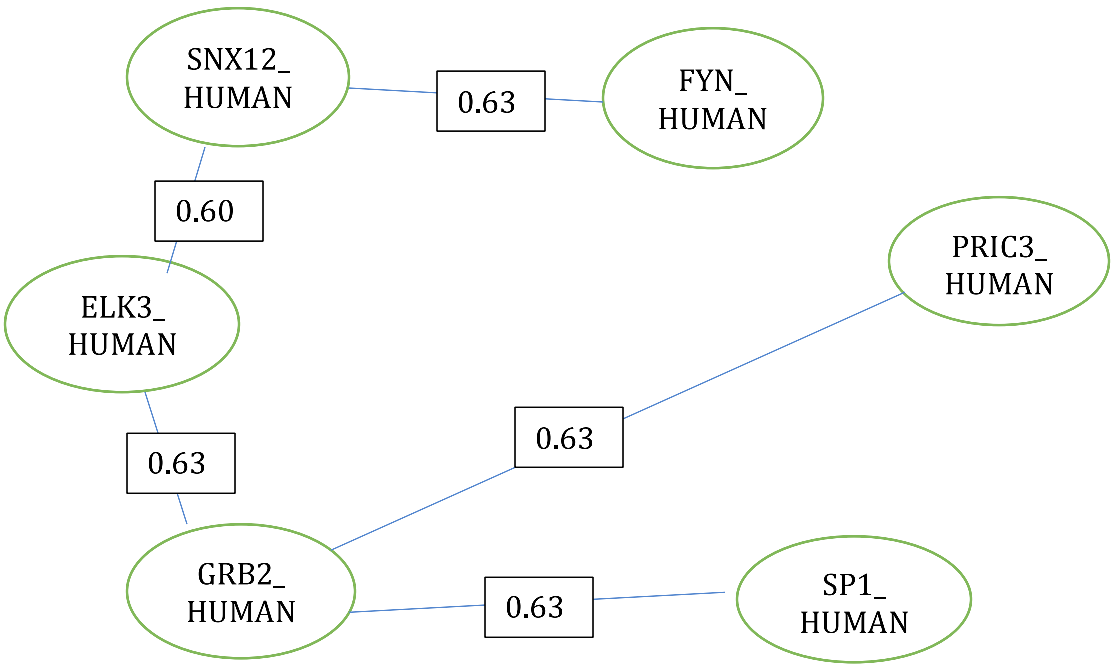

Figure 5. Graph Built from Parsing the HIPPIE Sample File.

the graph type for HIPPIE file type is a weighted and undirected graph.

To parse the HIPPIE file type, call the open method of the static FileParser class with the HIPPIE enum.

```
FileParser::open (string filepath, FILE_TYPE_ENUMS::HIPPIE, unorderd_map <string, int>* map )
```

Figure 6. calling the open method of FileParser class.

The method opens up the file, read the primary vertices into the map passed in as an argument. The content of the map is <string, int>. The string part is the name of the protein and the int part is their relative position in the file. The first protein is given an int value of 0(zero), the second one gets 1 etc. This integer value will play a role during slicing the number of vertices across cluster computing nodes.

Once the vertices are populated into the map along with their position based Id, FileParser::neighbor_init method will populate each vertex's neighbors along with the weight of the connection.

```
neighbor_init(unordered map<string, int> * , string filepath, FILE_TYPE_ENUMS type ,int
                    VertexIntId, vector<string> &neighbors, vector< double> &weight, string&)
```

Upon calling this method with the appropriate arguments, the class will call the right internal method to populate neighbors for each vertex. Here 'VertexIntId' is the value of the map for the vertex key, in map<vertex, int>, which is the position-based Id of the vertices.

2.1.1.2 MATSim: MATSim is an xml file for representing data for simulation of public transport. The simulation data contains networks with links available to public transport vehicles, a file describing the available public transport vehicles, a file containing the public transport schedule, and some specific settings in the configuration. Each link has a list of available transport modes. If no modes are specified, the simulation assumes that only "car" is allowed on such links [5].

Here is a same data for MATSim file type.

```
<network>
        <nodes>
                <node id="1" x="2000" y="1000" />
                <node id="2" x="4000" y="1500" />
                <node id="3" x="3000" y="3000" />
        </nodes>
        <links capperiod="01:00:00">
                <link id="1" from="1" to="2" length="3000.0" capacity="1800" freespeed="13.88"
                                                  permlanes="1" modes="car" />
                <link id="2" from="2" to="1" length="3000.0" capacity="1800" freespeed="13.88"
                                                           permlanes="1" modes="car" />
                <link id="3" from="2" to="3" length="5000.0" capacity="3500" freespeed="22.22"
                                        permlanes="2" modes="car" />
                <link id="4" from="3" to="2" length="5000.0" capacity="1800" freespeed="22.22"
                    permlanes="1" modes="car" />
        </links>
</network>
```

Figure 6. Sample MATSim traffic simulation data file [5].

In MATSim file type, all the vertices are inside <nodes> element of the xml structure. The associated neighbor and the cost of the link is given inside the <links> element.

Parsing the above MATSim sample file results the following graph.

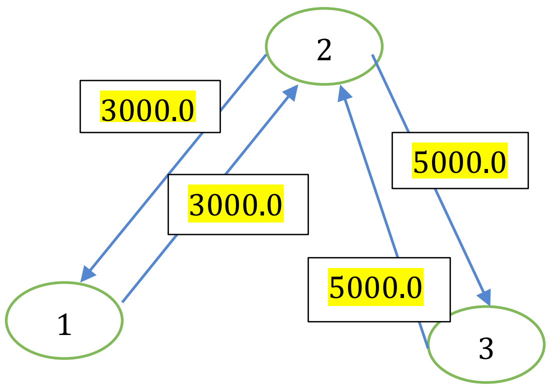

Figure 7. graph from sample MATSim file.

As indicated in figure 7, the graph resulted from parsing MATSim files is directed and weighted. This makes a logical sense since most roads are two way. The link weight between two vertices might be different even though the graph above doesn't show this behavior. The weight is not necessarily the same between two nodes in both directions. The cost of A to B might be for example 5000 and the cost from B to A might be 2000.

Because C++ don't have a native xml parser library, I used a third-party library called pugixml. I chose this library for its high performance among other available choices. More information can be found about pugixml and how to use it at [6].

Like the HIPPIE file type, calling the FileParser::neighbor_init(), with the right arguments will populate the neighbors for each vertex.

1.2 Implementing new classes for graph features

In addition to the file parser class discussed above, there are other new classes added to support the graph feature of MASS C++ library. The UML diagram in figure 2 shows how these classes interact each other and with the existing MASS C++ classes.

1.2.1 **VertexPlace**: this class is an abstract class that inherits from the Place class. User defined classes which wants to benefitted from MASS C++ graph features must inherit from this class. In addition to inheriting it, they should also define the following methods.

```
extern "C" Place *instantiate_from_file(string filename, FILE_TYPE_ENUMS type,
                                        int global, void *arg, unordered_map<string, int> *dist_map) {
   return new UserDifinedCLass(filename, type, global, arg, dist_map);
}

extern "C" Place *instantiate(void *argument) {
   return new UserDifinedCLass (argument);
}
extern "C" void destroy(Place *object) {
   delete object;
}
```

Figure 8. Method that should be defined by child classes of VertexPlace class.

In figure 8, "UserDifinedCLass" is a child class that inherits from VertexPlace. For instance, if a class called MyGraph inherits from VertexPlace, in place of UserDefinedClass, the programmer must put MyGraph.

On top of this, the user must implement the pure virtual method called callMetgod.
`void *callMethod(int functionId, void *argument) = 0;`

other public methods of VertexPlace class

Method name	Return type	Argument and what the function does
VertexPlace	Constructor	• String Filepath: the file path • FILE_TYPE_ENUMS: one of the file type enums • Int globalIndex: global Id of the vertex • void*: an argument for the user defined class • unordered_map<string, int> *: map pointer
VertexPlace	Constructor	• globalIndex: Id of the vertex as int • void*: argument for user defined class
VertexPlace	Constructor	void*: argument for user defined class
VertexPlace	Constructor	• string vertexName: the name of vertex • vector <string> &neighbors: self-explanatory • vector<double> &weights: self-explanatory
addNeighbor	void	Adds a neighbor to the existing neighbors • string neighborName: name of the neighbor

		• double weight: weight of the connection
removeNeighbor	void	removes a neighbor from the existing neighbors • string neighbor: name of the neighbor to be removed
clearAllNeighborsAndWeight	void	It clears all the neighbors. Take no argument
getNeighbors	Vector<string>	Return all the neighbors of the vertex. Takes no argument

2.2.2. **GraphPlaces:** this class is the bread and butter of the MASS C++ graph feature. It is the child class of Places class. When this class constructed, a multi-dimensional array of Place objects will be maintained inside the MASS_base::dllMap, an instance of map<int, Places_base *> . Unlike the existing MASS C++ library, the graph feature supports dynamically adding and removing Place (VertexPlace) objects once instantiated. Adding/removing Place objects after instantiating is not a trivial task. Before adding/removing, we need to know where the next vertex should go in the cluster. We must determine how to retrieve a particular vertex when we need it. This use case needs a reliable mathematical mapping algorithm to determine the next computing node to send/remove the vertex to.

Fortunately, there is a clever mapping algorithm implemented by Justine Gilroy [3] for the MASS Java library graph feature. MASS C++ library takes advantage of that algorithm.

Here a brief description of the algorithm.

Let's say we have read 100 vertices in one of file types and assume we have four members nodes in the cluster, nodes with rank 0, 1, 2, and 3.
Each node we have a share of 25 objects.

Strip size = #place/#computing node ……………………………………………[1]
Strip size = 100/4 = 25. ---> using formula 1.
Therefore,
place from 0-24 -----------goes to node with rank 0.
place from 25-49 ---------goes to node with rank 1
place from 50-74 ---------goes to node with rank 2
place from 75-99 ---------goes to node with rank 3

however, because of the dynamic nature of the graph feature, where should the next Place object should be created and how can we retrieve it if needed? This is where the Justin's mapping algorithm comes handy. For the algorithm to work as intended, the initial size (100 from the above example) should not be modified at any time after instantiating.

To determine which computing node the next vertex should go, divide the global Id of the vertex by the size (100 from the above example) and then take module of the initial strip size (25 from the above example). This gives you the computing node that this vertex should go.

The following example is taken from Justin's white paper [2], with little modification

Vertex Name: Hello
Global index: 362 ---> let's assume it is 362.
MASS Size: 100 ---> the initial size
Size of the cluster: 4
Stripe size: 25 ---> 100/4
Layer Index(raw): 3 --> (362 / 100) ---> the row
Next node rank: 2 --> (362 % 100 / 25)
Place index (column)to add the vertex into: 12 --> (362 % 25)

Even though this mapping algorithm solves the vertex allocation problem, it has a downside. It doesn't share the workload evenly. Let's take an initial size of 100, 000(which is not too big to assume in today's big data) and 10 commuting nodes. The algorithm allocates the next 10, 000 nodes for node 1(rank 0). Therefore, node with rank 0 is overburdened by an additional 10,000 jobs before the algorithm starts allocating vertices to the next computing node.

I attempted to generate the Id of the vertex randomly (as opposed to sequentially) to mitigate the vertex allocation skewness of the above algorithm. However, I realized that doing so will create holes in the simulation space. For example, if the random number generator produces 109 in the above example, the simulation space is hollow from the coordinate where vertex with Id =100 resides all the way to where vertex with id = 109 resides.

However, for lack of clear solution for now, I implemented the same algorithm for MASS C++ library. But this algorithm should be optimized for better job allocation in the future.

The above vertex allocation algorithm is implemented in GraphPlaces class for MASS C++ library.

GraphPlaces class, in addition to inheriting from Places classes, it also implements the Graph interface (a pure abstract class).

Public methods of GraphPlaces class.

public	GraphPlaces (int handle,string className,int boundary_width,int dimension, string filename, FILE_TYPE_ENUMS type, void* argument, int arg_size); ➢ calls the Place_base constructor, and the Place_base constructor calls the appropriate file parser routine indicated by FILE_TYPE_ENUMS. Then the vertices will be populated in the map<string,int> for further use. ➢ After populating the map, the master node instantiates Place objects and holds them inside MASS_base::dllMp ➢ When the Places_base constructor returns, GraphPlaces constructor calls the "init_master_base method of the Places class to send the map of vertices and arguments to the remote nodes.
Public	GraphPlaces(int handle, string className, int boundary_width, int dimension, void* argument, int argSize, int numberOfVertices); ➢ calls the Place_base constructor, then the Places_base constructor of the master node calls init_master method to instantiates Place objects and populate then in MASS_base::dllMp ➢ when the Places_base constructor returns, the GraphPlaces constructor calls the init_master_base method of the Places class to send arguments so that remote nodes can instantiate Place objects.
public GraphModel*	getGraphOnThisNode () ➢ returns all the GraphModel objects which lives on the master node
public GraphModel*	getAllGraphOnTheCluster ( ) ➢ return all GraphModel objects that live on the cluster
public void	setGraph(GraphModel &newGraph) ➢ adds the GraphModel to the value indicated by the argument
public bool	addEdge(string vertex, string neighbor, double); ➢ adds an edge between the vertex and the neighbor and set the weight of the link. If this process is happening on the master node, return

	true, sends addEdge message to remote nodes and return false
public bool	removeEdge(std::string vertexId, std::string neighborId); ➢ removes an edge between vertices. Return true of the process happens at the master node, sends removeEdge message to remote nodes and return false.
public int	addVertex(string vertexId); ➢ adds the vertex to the list of vertices and returns the index
public int	addVertex(string vertexId, void *argument, int arg_size) ➢ creates a VertexPlace object from the argument and from the class and adds it to the list of VertexPlace vector
public bool	removeVertex(std::string vertexId); ➢ removes the vertex from the cluster. Returns true if the vertex is on the master node, sends removeVertex message to remote nodes and return false
public bool	removeEdgeLocally(std::string vertexId, std::string neighborId); ➢ removes the edge between vertices and its neighbor on a local machine. If the edge found and removed, returns true, false otherwise.
public bool	addEdgeLocally(std::string vertexId, std::string neighborId, double weight); ➢ addes an edge between the vertices in the argument. If the vertices found and the edge formed, it returns true, false otherwise.
public int	addPlaceLocally(string vertexId, void* argument, int arg_size); ➢ create a Place object on the local machine from the argument and the class name and adds it to the VertexPlace vector and returns the index the object inserted. If the process couldn't succeed returns -1.
public bool	removeVertexLocally(string vertexId); ➢ removes the vertex on the local machine where the method is called. If found and remove, return true, false otherwise.
public int	addVertexPlace(string host, string vertexId, void* argument, int arg_size); ➢ calls addVertexLocally method if the host name the master's host name and returns the index the vertex inserted. If the hostname is not the master, the method send addVertexPlace message to the remote nodes and returns -1.
public void	merge(GraphModel &source, GraphModel &remoteGraphs); ➢ merges two GraphModel objects together

public void	callPlaceMethod(int functionId, void* argument); ➢ calls the Place's method callMethod for each vertex.
public void	callAllWithReturns(int functionId, vector<void*> &returns, vector<void*> &arguments); ➢ calls the Place's the function indicated with the function Id and keeps the return value into 'returns, the vector of void* in the argument.
public void	exchangeAll(int currentFunctionId, int handle); ➢ calls the function indicated by the currentFunctionId for the GraphPlace simulation object indicated by the handle. Then The vertexPlace will exchange information according to the implementation of the function indicated by the currentFunctionId.
public void*	exchangeNeighbor(int functionId, vector<int> neighbor, void*argument); ➢ the neighboring Place objects exchange data by calling the function indicated by the function Id.
public int	getTotalPlaceOnThisNode ➢ returns the count of vertices which lives on the master node.

There are other new classes created for the graph feature, but they are small and there are enough comment comments. Also, they are not usable for the user, they are there for the programmer.

1.3 Modifications on the Existing MASS C++ Classes

In addition to creating new classes, I have modified some of the existing classes.

2.3.1 Places_base:
there are additional utility methods added in this class to support the graph features. Upon calling the constructor from GraphPlaces class with the appropriate arguments, the constructor calls either 'init_all_graph' for the master node or 'init_all_graph_for_worker_nodes' for the worker nodes.

The difference between these methods is, init_all_graph method reads the vertices from the given file type by calling 'FileParser::open' method before creating and populating the MASS_base::dllMap with Place object. However, the remote nodes will get the vertex map (unordered_map<string, int>) from the master node. Therefore, remote nodes don't need to parse the file again. So they call 'init_all_graph_for_worker_nodes' method instead

On top of this, the init_all_graph and the init_all_graph_for_worker_nodes use the 'instantiate_from_file' object factory method in place of the usual 'instantiate' object factory method of DllClass.

In addition, there are some redundant methods that are also implemented in GraphPlaces class. The reasoned for this redundancy is, because of the way the classes compiled and linked in the Makefile, we cann't use either Places's or GraphPlaces's methods inside the MProcess class. Doing so will result in the 'undefined references' linker error. Therefore, the easiest solution I found is propagating some of the methods from GraphPlaces to Places_base class.

2.3.2 other modifications on the existing classes

Other modifications which I'm not covering in detail are also made inside the Message, MProcess, MObject header file, DllClass and Places classes.

There are additional elements added in the 'ACTION_TYPE' enum added for graph features. Serialization and deserialization routines are also implemented for these added enum elements in the Message class. These added ACTION_TYPE enum are also handled inside the MProcess class.

Here under is the diagrammatic view of sequence of actions when MASS C++ started with GraphPlaces.

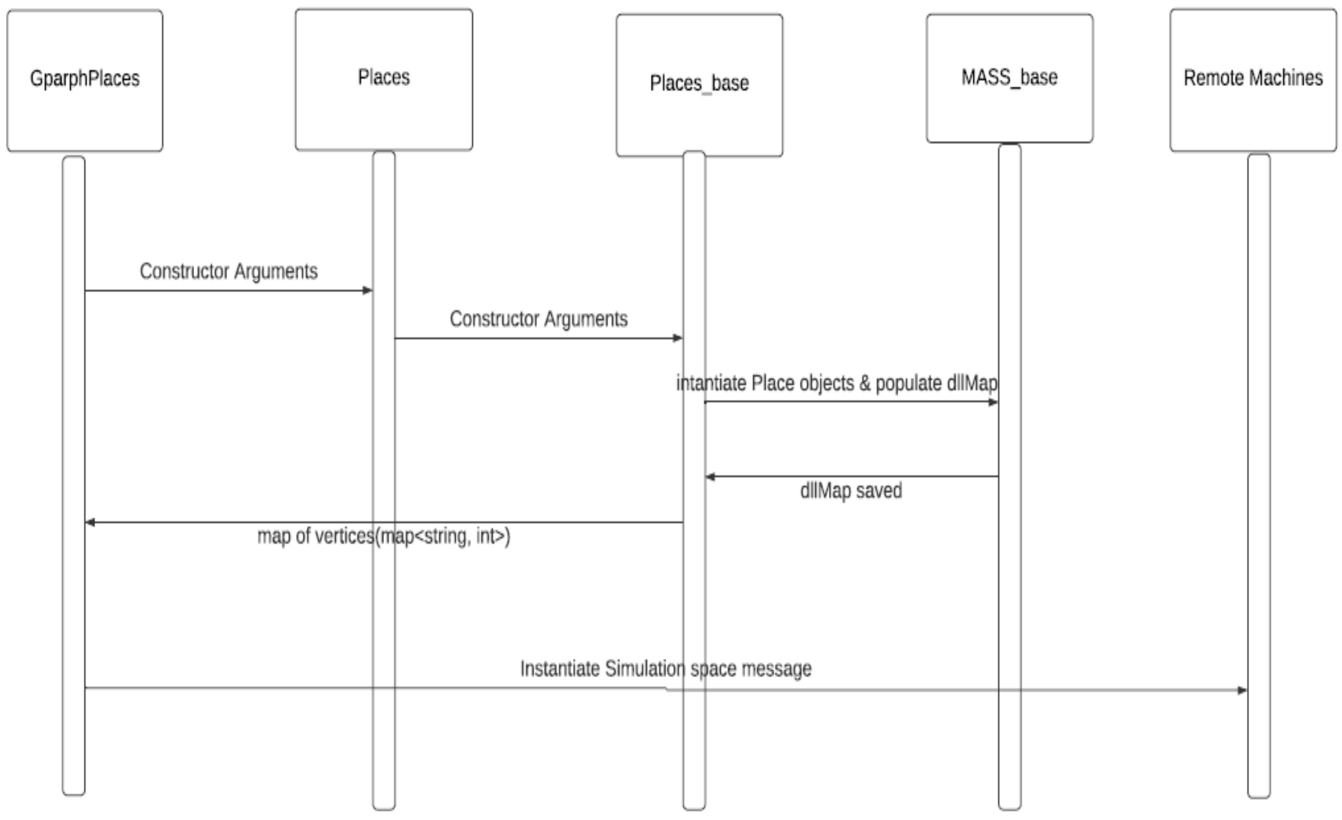
GparphPlaces
Places
Places_base
MASS_base
Remote Machines
Constructor Arguments
Constructor Arguments
intantiate Place objects & populate dllMap
dllMap saved
map of vertices(map<string, int>)
Instantiate Simulation space message

Figure 9. Sequence of Events when MASS started.

References

1. Thakur, Rajeev and Gropp, William. “Parallel I/O”. www.citeseerx.ist.psu.edu. *Accessed June 12, 2020*.
2. Fukuda, Munehiro. MASS C++: Parallel-Computing Library for Multi-Agent Spatial Simulation. University of Washington, August 4, 2015.
3. Gilroy, Justin. Dynamic Graph Construction and Maintenance. University of Washington, 2020.
4. “Human Integrated Protein-Protein Interaction rEference.” *www.cbdm-01.zdv.uni-mainz.de/~mschaefer/hippie/information.php.* Accessed July 12, 2020.
5. Rieser, Marcel. “matsim.atlassian.net/wiki/spaces/MATPUB/pages/83099693/Transit+Tutorial.” Last updated Jan 12, 2018. Accessed July 20, 2020.
6. “pugixml 1.10 quick start guide.” www.pugixml.org/docs/quickstart.html. Accessed Aug 2, 2020.