CSE 6230 Fall '14: Project Report

# Building a framework for executing parallel graph algorithms

Ankit Srivastava and Saminda Wijeratne

December 8, 2014

## 1 Introduction

Parallel graph algorithms are significantly different from other generic parallel applications in terms of computational requirements. Few dedicated libraries, like Parallel Boost Graph Library (PBGL), CGMGraph, etc., are available for parallel graph processing. However, these libraries suffer from inefficiencies when it comes to large scale processing [1]. Hence, a framework for processing graphs in parallel, which abstracts away all the parallel processing complexities, is required.

MapReduce [2] framework seems like an automatic pick for this purpose. However, the framework is inefficient when applied to classes of algorithms where data dependencies are involved. Even though design patterns for implementing some classes of graph algorithms using MapReduce have been proposed [3], iterative nature of graph algorithms renders the framework largely unsuitable for general graph algorithms.

## 2 Literature Review

### 2.1 Previous Work

Sarje and Aluru proposed a general framework for parallel computations on tree structures [4]. Much like MapReduce, the proposed framework requires implementation of two functions, *generate* and *combine*, which are combined to perform computations on trees. Our proposed framework will be based on implementing and extending this framework based on research done on existing scientific applications and support for graph data structures. Figure 1 depicts the framework proposed by Sarje and Aluru.

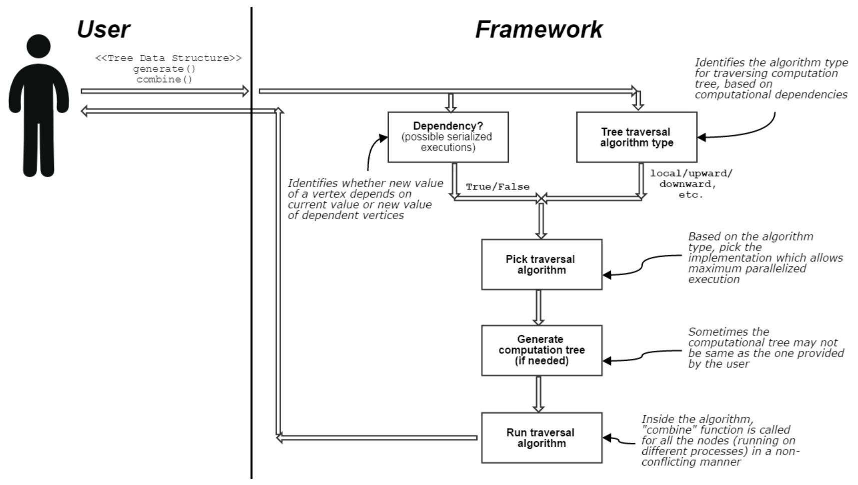
User
Framework
<<Tree Data Structure>>
generate()
combine()
Dependency? (possible serialized executions)
Tree traversal algorithm type
Identifies the algorithm type for traversing computation tree, based on computational dependencies
Identifies whether new value of a vertex depends on current value or new value of dependent vertices
True/False
local/upward/ downward, etc.
Pick traversal algorithm
Based on the algorithm type, pick the implementation which allows maximum parallelized execution
Generate computation tree (if needed)
Sometimes the computational tree may not be same as the one provided by the user
Run traversal algorithm
Inside the algorithm, "combine" function is called for all the nodes (running on different processes) in a non-conflicting manner

Figure 1: Schematic of the framework

The traversal algorithms themselves will have efficient way to run the *combine* in such a way maximum parallelism can be achieved. For example Sevilgen, Aluru, and Futamura [5] explain an efficient way to parallelise tree accumulation computations on a partitioned tree when performing upward accumulation with the possibility of achieving near $O(\log n)$ time complexity.

## 2.2 Existing Frameworks

### 2.2.1 GraphLab

GraphLab [6] is an asynchronous parallel graph processing framework. It is a vertex centric abstraction where each vertex can read and write to data on adjacent vertices and edges. Users provide sequential functions for performing relevant computations on data and the framework ensures consistent parallelism while executing these functions. The framework requires three inputs from the user: data graph, update function, and sync operation. It then distributes vertices in the data graph to different partitions, minimizing edges between partitions. One or more partitions would be distributed to each compute node. Two engines ensure consistency in execution among different nodes. The chromatic engine uses graph coloring to ensure consistency in executing the update functions in parallel while the distributed locking engine has more flexibility in the execution to allow prioritized serialized execution using read/write locks on vertex data.

One key difference between GraphLab and the proposed framework is that, instead of explicitly requiring the data graph, the proposed framework requires the user to provide a function for generating it. This approach offers significant advantage over GraphLab because users can work with same graph even if the computation dependencies of vertices change. Also, since data graph is built by the framework, it can be done efficiently for very large graphs.

### 2.2.2 Pregel

Pregel [7] is the generic parallel graph framework by Google which addresses the distributed processing of large scale graphs. Based on Valiant's Bulk Synchronous Parallel Model [8], it includes an API that is sufficiently flexible to express arbitrary graph algorithms. In Pregel computation model, the computation is done iteratively on each vertex in what is referred to as supersteps, at the end of which all vertices are synchronized. Each vertex can send/receive data and modify the state based on received messages during the superstep.

The proposed framework does dependency based computations which is fundamentally different from the iterative method used by Pregel. The communication pattern is also differnt as the proposed framework does need based fetching of data for vertices from other processes, using Remote Memory Access, instead of explicitly sending/receiving data as is done in Pregel.

### 2.2.3 HipG

Hierarchical Parallel Graph [9], or HipG, model is targeted towards divide-and-conquer graph algorithms. HipG parallel model consists of two basic elements: computations on graph nodes and orchestration of these computations by synchronizers, which are present on each node and manage the graph computations and distributed communication among the nodes.

HipG relies on asynchronous communication for remote method calls while the proposed framework, by its very nature, relies on synchronous communications. HipG expects a pre-partitioned graph as input and divides the nodes into equal sized chunks if an unpartitioned graph is provided. Ideally, good partitioning should minimize edges going across chunks for minimizing communication, however we have skipped implementing this in the current version and leave it for future work.

## 2.3 Scientific Problems Involving Graphs

We went through a list of scientific applications [10] for finding sample programs which can be parallelized through the proposed framework. The analysis of these applications shaped purpose of our framework significantly. While going through the application list, we encountered many variations of tree/graph problems which required more than just performing computations on each node in parallel. Therefore, we decided to explore these problems and try to find out how, if at all, our framework can be applied to them.

### 2.3.1 Data Point Computation Problems (eg: n-body, map-reduce)

This category of problems include calculating new values at data points, with or without considering the rest of the data points or other external factors. Since these problems require multiple computations to be done simultaneously, there exists potential to parallelize these computations. For example, n-body simulation is a problem which has many important applications of general relativity in fields of molecular dynamics, astrophysics and plasma physics. One of the approximation algorithms that can solve this problem efficiently is the fast multipole method (FMM). We will examine and evaluate FMM further in Section **??**. Page ranking problem is considered as a simplified version of n-body problem. Page ranking algorithm computes how important a certain data point is compared to other data points in the vicinity. Typically, the calculation is done using some heuristic value present in data point $x$ about data point $y$. Although this algorithm looks embarassingly parallel, the heuristic value may depend on the importance of data point $x$ itself, thus suggesting continuous improvement whenever the importance value of a data point is updated. Therefore, we found this class of problem ideal for the proposed framework.

### 2.3.2 Traversal Problems (eg: pattern search, pruning)

While these are highly interesting problems, it was not trivial to see how our framework can be applied to them. In fact, existing solutions for these kind of problems we found were in the form of DFS/BFS, cycle detections, etc. which, to our knowledge, have very little direct applications in efficient parallelism. However, a variation of the traversal problem is to discover multiple travesal starting points in the tree/graph, so that a serialized travesal algorithm can be applied node by node through the framework, while shared nodes are scheduled among processes accordingly. But this requires further research relating to precedence issues and, in some cases, unique deadlock conditions (when cycles exist), which we did not focus on in this project.

### 2.3.3 Comparison Problems (eg: longest common length, distance problems)

This catagory of tree/graph problems is somewhat related to our framework. Comparison problems, such as identification of genome mutations and common strands, can require multiple data structures with dynamic interdependencies. This presents an advance use case for our framework, which the current design can not handle. There are two main reasons for this. Firstly, it involves mutiple data structures, (current framework expects only one tree/graph data structure) and dynamic generation of data structures (*generate* function would have to be more sophisticated to spit-out data from tree/graph data structures which the original node is not connected to and the result of a *combine* function would lead to creation of new tree/graph data structures). Doing this in parallel will constitute a larger research problem which we did not focus on in this project.

#### 2.3.4 Convergence Problems (eg: threshold, decision, reorganization problems)

Some probelms require iterative calculation of new values for tree/graph nodes until the system obtains a set of expected results or comes to a set of acceptable margin of errors. At first, it looked like an obvious requirement from the framework itself. However, after analyzing different complexities involved in determining this specific state, it made more sense for user to make the decision as it would then be a single decision made for the whole data structure rather than for each node seperately with no parallelism being required here.

## 3 Scope and Domain Limitations

Determining the scope and the domain for the project took some time. Even after the initial project proposal we carried out few more literature studies in order to understand the full scope of the problem at hand in order to identify how to scale it down to the course project. In this section we will define the scope of the framework, the problem domain it handles and the reasons for doing so. Most of the decisions were influenced by the scientific problems we researched which involves trees/graphs and helped by guidance of our research advisor Dr. Srinivas Aluru.
We will tackle this section by answering a few questions regarding the requirements for the full project.

### 3.1 Deciding the supported problem domains

#### 3.1.1 Do we need to support bulk of the graph problems which have inherent parallel solutions?

As discussed in Section 2.3, there are different variety of graph problems with each of them having different complexities of implementation for the framework. Therefore, we decided to restrict ourselves with just single category of problems called "Data Point Computation Problems" (discussed in Section 2.3.1). We figured we'll be able to cover significant amount of requirements for that problem domain for solving that category of problems in parallel.

#### 3.1.2 Should the framework support creating a new tree/graph structure for computations other than what the user passes to the framework? Would it require a tree/graph repartitioning if the framework reuses the given data structure?

This was a main concern when we wanted to design the framework. In all practical use cases we came across it was evident the tree/graph data structure provided by the user resembles the same computation data structure which was required in the computation. Since we are aiming to achieve parallelism

with distributed computing our application of the framework is useful mostly when this data structure is very large such that single processor execution is infeasible due to insufficient resources or incurs too much overhead within a single compute-node.
Thus creating tree/graph data structures and syncing tree/graph nodes in the user given data structure with the new data structure creates considerable overhead to the whole process (alternative we considered was to repartition given tree/graph nodes to the new tree/graph nodes, but that would mean we are modifying users given node physical locations in the system without that being a primary objective of the users expectation from the framework). Therefore considering all these cases we decided our framework would support creating a new tree/graph data structure but for the purpose of this project it would only consider samples which the framework can reuse the user given data structures without modifying physical location of the data points. We figured for the moment we'll skip implementing the part where we create the new computation tree/graph if needed and pursue it nearing the end of the research project as time permits.

### 3.1.3 If we are reusing the data structure given by the user, how are we to determine existing partitioning of the nodes of the tree/graph in order to determine proper MPI process executions and message passing?

Given the above decision it was evident that this creates the dilemma of how the framework can identify the physical location of tree/graph nodes (i.e. the processor which it resides) in the data structure provided by the user. Our thought process was that either the user will have to provide an additional data map which provides the details of the tree/graph node locations or the framework should be aware of the data structure the user is passing and the data structure should contain the necessary details required the framework identify nodes in each processes local memory.
It is unfair for the user whom we are trying to refrain from having to go through MPI programming to perform additional work to pass a data map to the framework. Therefore we thought that it would be more prudent to allow users pass a data structure which the framework knows how to analyze. Since this data structure should support very large trees/graphs with efficient partitioning the best course of action we believe was to provide an abstract class which users can build their data structures upon. This abstract class will take care of handling the partitioning of the data and managing the mapping information users add to the data structure. To further ease the use, we can implement the abstract class and provide a library of known tree/graph data structures (binary trees, octrees, k-d trees etc.) for users to regardless of whether they'd want to use our framework or not.

## 3.2 Architecture Design Generalized to Acyclic Graphs

### 3.2.1 Since the structure of the graph represents the dependencies between the nodes in the graph resulting in the complexity of the framework who has to schedule all those variations of dependencies without any conflicts during computations, should we limit the scope of type of graphs our framework should support?

Our framework is mainly based on research done by authors Sarje and Aluru[4] which defines the support for only tree data structures. While we should atleast strive to achieve the support for any tree data structure, we decided that it would be interesting and useful if we can extend it to support the graphs as well. Eventhough this requires extensive research to determine how the scheduling/traversing algorithms should behaive for different graphs, we supposed our framework would be an ideal tool to test various solutions (because the work done in [4] can be indirectly extended to graphs with slight modifications in handling corner cases). However we decided it we'd limit ourselves for acyclic graphs (our intuition was that cyclic graphs exposes another set of problems of breaking cyclic dependencies for node computations)

## 3.3 Implementation Scope

### 3.3.1 What samples should we use to test our framework?

Although technically the framework should support any acyclic graph data structure, in order to keep the evaluation simple we decided to go ahead with tree data structure related problems.

### 3.3.2 Should we implement all traversal tree algorthms for the framework?

Due to limitation of time we would only implement the traversal algorithms required for the samples would create. However our goal is to design the framework in a such a way that it would be extensible to include new traversal algorithms. Thus a future researcher may use our work to create a better framework which would support broader set of problems.

# 4 Framework

## 4.1 Design and Architecture

The framework incorporates a repository/factory of classes capable of handling trees/graphs and traversing them while avoiding race conditions and deadlocks. This repository is what makes this framework flexible enough for different variations of computation graph structures. Thus it is defined in a such a way that makes the whole framework generic for different types of data structures and algorithms which is required to run on them.

Once the user invokes the framework, the framework performs 3 essential steps:

1. Generate the computation graph.
2. Analyze computation graph to retrieve the relevant traversal algorithms.
3. Perform the computations.

Step 1 returns the user provided graph if it is the same as the computation graph. Or else, it'll generate a new computation graph. The importance of step 1 is that when the computations are performed much of the relevant graph nodes required for a computation at each process will be available in the local memory. This will minimize the communication cost among processes (which is the bottleneck in most cases). The framework uses the `generate (...)` function in order to achieve this step.

Step 2 will analyze the computation graph structure and determine an algorithm that would traverse and compute the computation graph efficiently taking in to consideration that the graph is distributed among several processes. This takes in to mind 2 aspects of the computation:

1. Taking in to the structure of the graph what is the computation dependency between nodes?
2. Do the computations of the nodes dependent upon the new value of another node?

Note that given the answers for the above questions, there can be multiple solutions given by the factory for the same answer.

Step 3 then takes over running the actual computation on relevant data structures. Here the framework relies upon algorithm provided by the the repository/factory to carryout calling the `combine (...)` function in such a way that no race-conditions occur.

The pre-processing and post-processing steps are to setup and tear-down the relevant resources for the computation. After step 3, it can go to step 1 if another iteration is required. This is decided by the algorithm obtained at step 3. The algorithm will analyze the output of the `combine (...)` function and inform the framework whether or not to repeat the 3 steps.

Figure 2 depicts the architectural design for the framework.

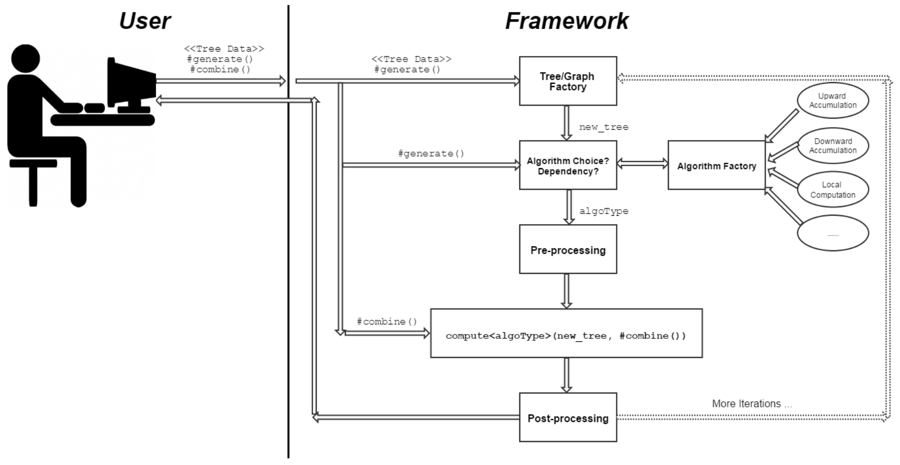

Figure 2: Architecture of the framework

## 4.2 Implementation

We implemented our framework in *C++11* and used *Open MPI* (v1.8.3) with *g++* (v4.8.2) for local compilation and testing.

### 4.2.1 Data Structures

The framework provides for two types of data structures for providing data on nodes in the graph. One is *Graph::Node*, which is the method used internally by the framework, and the other is using *DataPoint*, which is used for providing user data on nodes. The computation is always done on *Graph::Node*. Therefore, if data is provided in the form of *DataPoint* then it is converted to *Graph::Node* before computation and back to *DataPoint* at the end of the computation.

### 4.2.2 User Space

Figure 3 shows user space of framework in the form of a UML diagram. The user space consists of *GraphCompute* class which interacts with *Graph*, or classes derived from it, and implementations of interface classes *GenerateFunction* & *CombineFunction* for doing computations on the graph. *GenerateFunction* & *CombineFunction* are interfaces for functor classes that would implement *generate* & *combine* functions, for either a particular class of algorithms or general algorithms, as described earlier.

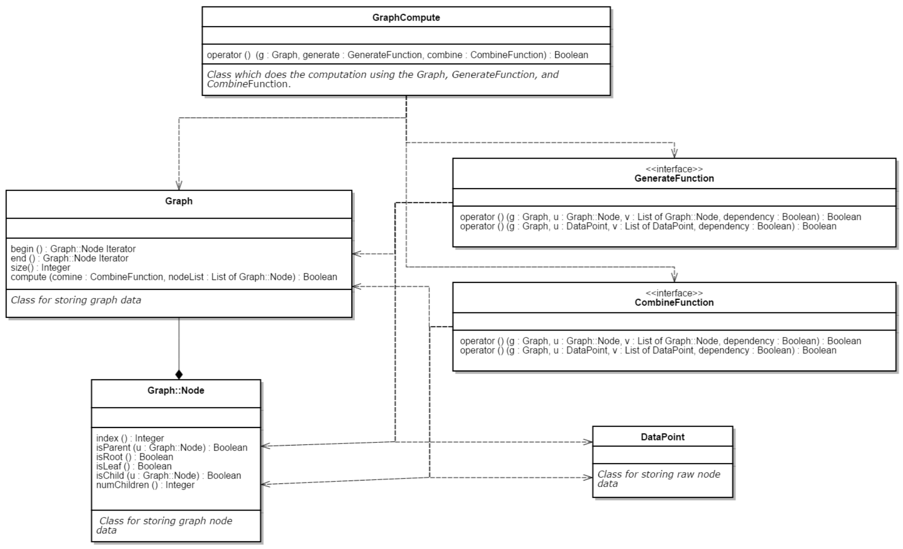
GraphCompute
operator () (g : Graph, generate : GenerateFunction, combine : CombineFunction) : Boolean
Class which does the computation using the Graph, GenerateFunction, and CombineFunction.
<<interface>>
GenerateFunction
operator () (g : Graph, u : Graph::Node, v : List of Graph::Node, dependency : Boolean) : Boolean
operator () (g : Graph, u : DataPoint, v : List of DataPoint, dependency : Boolean) : Boolean
Graph
begin () : Graph::Node Iterator
end () : Graph::Node Iterator
size() : Integer
compute (comine : CombineFunction, nodeList : List of Graph::Node) : Boolean
Class for storing graph data
<<interface>>
CombineFunction
operator () (g : Graph, u : Graph::Node, v : List of Graph::Node, dependency : Boolean) : Boolean
operator () (g : Graph, u : DataPoint, v : List of DataPoint, dependency : Boolean) : Boolean
Graph::Node
index () : Integer
isParent (u : Graph::Node) : Boolean
isRoot () : Boolean
isLeaf () : Boolean
isChild (u : Graph::Node) : Boolean
numChildren () : Integer
Class for storing graph node data
DataPoint
Class for storing raw node data

Figure 3: UML diagram of user space

### 4.2.3 Internal Structure

We discussed user space for the framework in Section 4.2.2 wherein user relies on *GraphCompute* to do computations on *Graph*. Figure 4 shows the internal structure of the framework, with only implementations specific to local computations shown.

Internally, *GraphCompute* first generates interaction set for all the nodes by calling *GenerateFunction* for each node. Using the generated interaction set, it then determines the combine case, or *AlgorithmChoice*. It then gets a traversal algorithm from *GraphAlgorithmFactory* based on the combine case and then use the traversal algorithm for traversing the graph nodes and applying *CombineFunction* to every node.

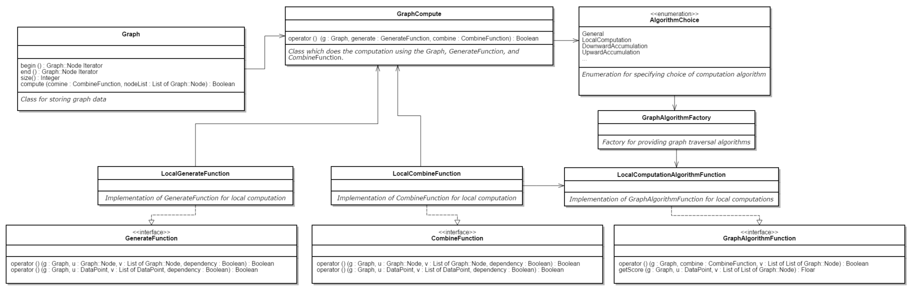
Graph
begin () : Graph::Node Iterator
end () : Graph::Node Iterator
size() : Integer
compute (comine : CombineFunction, nodeList : List of Graph::Node) : Boolean
Class for storing graph data
GraphCompute
operator () (g : Graph, generate : GenerateFunction, combine : CombineFunction) : Boolean
Class which does the computation using the Graph, GenerateFunction, and CombineFunction.
<<enumeration>>
AlgorithmChoice
General
LocalComputation
DownwardAccumulation
UpwardAccumulation
...
Enumeration for specifying choice of computation algorithm
GraphAlgorithmFactory
Factory for providing graph traversal algorithms
LocalGenerateFunction
Implementation of GenerateFunction for local computation
LocalCombineFunction
Implementation of CombineFunction for local computation
LocalComputationAlgorithmFunction
Implementation of GraphAlgorithmFunction for local computations
<<interface>>
GenerateFunction
operator () (g : Graph, u : Graph::Node, v : List of Graph::Node, dependency : Boolean) : Boolean
operator () (g : Graph, u : DataPoint, v : List of DataPoint, dependency : Boolean) : Boolean
<<interface>>
CombineFunction
operator () (g : Graph, u : Graph::Node, v : List of Graph::Node, dependency : Boolean) : Boolean
operator () (g : Graph, u : DataPoint, v : List of DataPoint, dependency : Boolean) : Boolean
<<interface>>
GraphAlgorithmFunction
operator () (g : Graph, combine : CombineFunction, v : List of List of Graph::Node) : Boolean
getScore (g : Graph, u : DataPoint, v : List of List of Graph::Node) : Floar

Figure 4: UML diagram of internal framework

#### 4.2.4 Algorithm Factory

*GraphAlgorithmFactory* is a factory where multiple traversal algorithms can be registered corresponding to each combine case. The framework can then decide the best algorithm for traversing an algorithm based on an objective score for each algorithm.

# 5 Discussion

## 5.1 Design and Implementation Challenges

### 5.1.1 Generic *generate* and *combine* Function Signatures

The different use cases involved forced us to think of specialized form of *generate* and *combine* functions which would be well suited for all the use cases. For example, for local computations the dependency flag returned from *generate* function is redundant and the *combine* function need only pass one node of the tree. For users who are implementing the *generate* and *combine* functions for this usecase could get confused due to redundant data needs to be calculated or additional parameters which they have no use of. In order to solve this issue what we thought was that rather than user introducing pointers to function implementations which has a specific signature he/she would introduce pointers to objects which has multiple versions of the *generate* and *combine* functions. Users will extend the base-class for these objects to create their own class containing the *generate* and *combine* functions. The base class will have the relevant variations for the *generate* and *combine* functions and users may choose to implement what deemed relevent for their problem. The framework should handle selecting the correct version of the function from the user provided objects. This approach makes it pretty useful if future patterns of *generate* and combine emerges, so that the developers will only have to update the base class and the relevent algorithmFactory function implementation without doing any changes to framework internals. One drawback however is that we are complicating the life of the user who now has to implement a whole class instead of just one function.

### 5.1.2 Supporing Multiple Implementations for Same Algorithm Choice of Traversal

Our abstraction for introducing traversal algorithms for different node dependency travel types allows multiple implementation for a single type i.e. developers can implement multiple implementations of the provider class which would contain different way of performing traversal for the given node dependencies. This causes a problem that who would get the precedance if multiple algorithms are possible. Choosing the correct algorithm is important because based on the problem at hand and/or how the tree/graph is structured different implementations may behave differently in the sense of amount of resources used. The obvious way to solve this was to introduce a scheduling algorithm to select the best algorithm out of all possible implementations. In order to this our criteria would be to introduce an abstract function for the base provider class which would return a numerical value representing how well it thinks it can perform for the given tree/graph.

### 5.1.3 Race Condition Issues while Calling *combine*

This was the usual scheduling issue we faced mainly because we have multiple processes where each has its set of nodes which it intends to process and the dependent nodes of each node in a process can potentially be present in other processes. Thus it was evident that process MPI communication needs to be scheduled among processes for sending values of ghost nodes once the values for those nodes are properly updated through computations. However, this can cause a massive amount of communications depending upon the number of levels and the branching factor of the tree/graph. Thus, in order to minimize the communication, one approach we took was to sort the nodes in a such a way that at the time of computation most of the dependent nodes of a node will be available locally in the process. (however this is tree/graph data structure dependent. Therefore this improvement is localized to the data structure library which we are providing). The 2nd way which we tackled the issue was to perform read first in uses cases which computations require the old value of the dependent nodes (i.e. to cache the values of the dependent nodes) and synchronize computations (using MPI communication) in use cases which computations require the new value of the dependent nodes. This is a tricky situation which meant that either the framework should manage the caching of dependent nodes or the provider implementations should. We decided the providers should do this because then they have more flexibility in handling any corner cases that may arise which could potentialy be bottlenecks for optimal execution performance.

### 5.1.4 Exposure of User Data v/s Structure Data

This problem arose when considering the target audience for this framework. On one hand we can assume sophisticated software that is developed by experience software developers would be pretty knowledgeable about complicated data structures such as octrees, k-d trees, etc. Thus some level of understanding about the data structure will be very useful to get over the learning curve of using the framework. On the other hand if developers such as scientists or research assistants who do not have a thorough computing background want to use this framework, they will be left branching in to learning concepts of understanding the context of different data structures and the importance of the stuctural representation of each domain data in an encapsulated object such as a node. While the latter catagory is certainly capable of learning the concepts, it would be a waste of their offort to spend time pursuing something which is not in their domain interest. While technically this does not create a problem for the framework, it however can cause alot of inconvinience for the user.

For example, incorrect interpretation of the data structure can lead to waste of resources, invalid results and debugging issues. Which would count on as a negative impression about the overall system. One solution we thought of was to introduce multiple function signatures based on different category of users. For example, for a certain *generate* function signature we can pass tree/graph level data structural information (tree, node, isLead/Root etc.) while another signature for the *generate* function could be just passing user level data which the framework would extract from the encapsulated data structural objects and later update back once returns from the *generate* function. While this is not an ideal solution for the problem, for the time being without any direct feedback we chose this as the implemented solution.

#### 5.1.5 Dynamic Addition of Algorithms

This was a trivial issue which we hid through the algorithms factory class. We decided a later implementation/improvement of the framework can handle this without any repurcussions to the rest of the framework. Therefore, right now introducing a new framework involves in hard coding to add the algorithm object to manually and recompiling the framework.

## References

[1] Andrew Lumsdaine et al. "Challenges in parallel graph processing". In: *Parallel Processing Letters* 17 (2007), pp. 5–20.

[2] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51 (Jan. 2008), pp. 107–113. ISSN: 0001-0782.

[3] Jimmy Lin and Michael Schatz. "Design patterns for efficient graph algorithms in MapReduce". In: *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. ACM, 2010, pp. 78–85.

[4] Abhinav Sarje and Srinivas Aluru. "A MapReduce Style Framework for Computations on Trees". In: *Proceedings of the 2010 39th International Conference on Parallel Processing*. ICPP '10. IEEE Computer Society, 2010, pp. 343–352. ISBN: 978-0-7695-4156-3.

[5] Fatih E. Sevilgen, Srinivas Aluru, and Natsuhiko Futamura. "Parallel algorithms for tree accumulations". In: *Journal of Parallel and Distributed Computing* 65.1 (2005), pp. 85–93. ISSN: 0743-7315.

[6] Yucheng Low et al. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proc. VLDB Endow.* 5.8 (Apr. 2012), pp. 716–727. ISSN: 2150-8097.

[7] Grzegorz Malewicz et al. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2.

[8] Leslie G. Valiant. “A Bridging Model for Parallel Computation”. In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782.

[9] Elzbieta Krepska et al. “HipG: Parallel Processing of Large-scale Graphs”. In: *SIGOPS Oper. Syst. Rev.* 45.2 (July 2011), pp. 3–13. ISSN: 0163-5980.

[10] *Scientific Applications.* `http://www.bcgsc.ca/platform/bioinfo/software`.