# Frequent Subgraph Mining by Giraph Distributed System

**Sadhana Priyadarshini, Sireesha Rodda**

***Abstract:** To overcome the challenges for managing the rapid growth of social graphs, massive Distributed Graph Mining Systems are developed, such as Pregel, GiraphHama, GraphLab, PowerLab, etc. The common approach to all systems is to divide the entire Graph Dataset into smaller divisions and use it as "think like a vertex", the programing model is to hold up a continual graph calculation. In this paper, we use the Optimized Frequent Subgraph Mining algorithm in the Giraph framework model and make a comparative study with existing different Distributed Systems. To enhance the flexibility and performance of the novel method, we carry out different optimization techniques associating it with updating different run time limits. We also investigate how the performance could be improved by Giraph Distribution System, which plays a vital role in social graphs such as LinkedIn, Twitter, Facebook, etc. The graph input, output, cluster set up and hardware configuration play vital roles in optimizing the performance of our proposed algorithm.*

***Keywords:** frequent subgraph mining, Graph Distribution, minimum support, Zoopkeeper.*

## I. INTRODUCTION

*Graph Mining Management* has become a vital research field in current generation as a result of fast growing in data in different areas such as bioinformatics, social network, semantic web, etc. For instance, by using *frequent subgraph patterns*, we can find out associations between different groups in social networks that help to understand the mechanism of social interaction and behavior. Similarly, researchers are able to investigate *protein-protein* interaction in case of bioinformatics. Generally we classified the entire large-scale frequent subgraph mining approaches into two parts, where the *frequent subgraph patterns* are used to generate set of subgraphs. In first part, one single graph data set consists of terabytes of data such as social networks. Second part, a large-scale collection of appropriate sized graphs such as bioinformatics [1].When we are deal with Big Database, the shared and centralized based systematic systems are need to transform into distributed decentralized architecture based systems. To overcome problems related to large-scale graph processing, the *vertex-centric program model* is developed, included into the distributed processing framework. In practical, we need the data that should be associated with its meaning and the value which can be retrieved from the main resource. So it is our continuous effort generating and analyzing them [1, 2]. Yahoo developed an open source project called as "Apache Hadoop" framework for rigorous distributed system.

The "*think like a vertex"* TLAV framework is the plan of action that repetitively carries out the program developed by the user over nodes of a graph. This approach is less expensive compared to the *conventional-graph omniscient algorithm*, and found better results in the field of *Distributed Systems*. It is a software which consists of a number of independent sections that need for program implementation and system production best. In the case of TLAV framework, the logic of vertex in the program is totally isolated from the timing and scheduling of its execution. The timing of a model is segregated to know how the active vertices are arranged by the scheduler for implementation. It can be a combination of synchronous and asynchronous or individual. The way data can be shared between the vertex programs is determined by communications in the TLAV framework. The message passing, shared, memory methods are used in both distributed systems and algorithms whereas active message methods are used to optimize the distributed message passing [3, 4, 5].

The model of implementation for vertex-centric programs describes the execution of vertex function and how data moves during computation. In case of vertex program implementation it can be one, two, or three phase-model. The vertex function can be implemented as an edge-centric function. It can be possible to separate the movement of facts related to vertex-program as data being sent and received [6]. The large-scale dataset should be classified up into number of slices so that it can easily fit in a distributed storage area. Distributed heuristics are decentralized techniques that need less or more centralized coordinates. For TLAV framework, *Steaming Partitioning* is a technique that uses a graph loader to load the graph from the disk to cluster. The vertex-cut is also another technique based on partition of edges instead of vertices. It uses a three-phase GAS model to overcome problems associated with it and increase its efficiency [3, 4]. There are chances of imbalances and growth in run time occurred during partitioning technique. It can be overcome by changing the number of active vertices in a given supersteps [7].

## II. RELATED WORK

Malewicz et al [8] proposed the first TLAV framework based on the BSP model. An order of iteration called as *supersteps that* are used to represent the programs, where message exchange between the vertices

Revised Manuscript Received on June 30, 2020.
* Correspondence Author

**Ms Sadhana Priyadarshini*,** Phd Scholar, Department of Computer Science and Engineering, GITAM (Deemed to be University), Visakhapatnam Andhra Pradesh India.
**Dr. Sireesha Rodda,** Professor, Department of Computer Science & Engineering, GITAM (Deemed to be University), Visakhapatnam Andhra Pradesh India.

and update individual state status can be possible from previous recurrence.

The model is applicable for any type of large-scale graph implementation and able to enhance its production quality, scalable, fault-tolerant execution. Kajdanowicz et al [9] developed the MapReduce technique that is based on map-*side join extension* and *Bulk Synchronous Parallel (BSP)*. The performance of MapReduce can be increased up to 10 times by the recurrence graph calculation of BSP processing models such as Apache Giraph, that uses disk storage support for large (larger than RAM memory) graphs.

Malewicz et al [10] designed a model appropriate for large-scale graph calculation and narration of its product quality, scalable, fault-tolerant execution. They also investigate the techniques for scaling to even bigger size graphs, such as moderating the model synchronously to skip the cost of faster workers who hold up often at inter superstep barriers.

Zhang et al [11] proposed an algorithm that performed well with accumulative changes by assembling the repetitive calculation. They developed Maiter that supports accumulative execution based on message exchange between hundreds of distributed systems.

Zhou et al [12] proposed a hybrid calculation model by Graph HP and used by P++ framework, which decreases the number of supersteps by reducing coupling intra-processor computation from inter-processor communication and synchronization. They use both boundary nodes and local nodes, during synchronous boundary nodes can exchange messages.

Dwarkadas et al [13] calculated performance of Pregel++, GPS, Giraph. GraphLab with respect to different graph characteristic algorithm categories, optimize technique. They found all systems are performed well depending on the parameters like datasets (i.e directed, undirected, weighted), threshold values, number of clusters, etc.

Protic et al [14] developed a model by merging advantage of shared- and distributed-memory approaches (DSM). The DSM systems give an option for variety scale multiprocessors and efficient building structure. This framework has the ability for both a clear and well organized programming platform for distributed and parallel application.

Tasci and Demirbas [15] developed an algorithm that execute sequential consistency without reducing the highly-parallel nature of BSP. It ensures that the coordination can be established between border vertices partitioned across workers.

Zheng et al [16] developed an engine to integrate with an SSD file system to achieve maximal performance, called FlashGraph which is a semi-external memory graph engine. It utilizes an asynchronous user-task I/O interface to lower the overhead related issues occurred during accessing data inside the file system and input/output operations. The sequence of processing vertices are scheduled to help to merge I/O requests and maximize the page cache hit rate. All of these designs are maximize performance for applications with different I/O access patterns.

Zhao et al [17] defined a mechanism for distributed system that search and eliminate the avoidable communication issues during the synchronization for abstraction. To achieve it, PowerGraph system implemented first and then communication overhead in distributed graph-parallel computation systems is reduced by LightGraph. An edge direction graph partitioning strategy is used to optimize the isolation of the outgoing edges from the incoming edges of a vertex during creation and distribution between replicas among different machines.

## III. APACHE GIRAPH OVERVIEW

The new apparatus of Google's Pregel is called *Apache Giraph*. The message is sent and received in a graph through a *Bulk Synchronous Parallel* programming model called "*Think like a vertex*" .It behaves like a map-only job on a *Hadoop Distributed System*. It has an in-memory scalable system to enhance the performance of handling huge numbers of vertices and messages in larger problem systems.

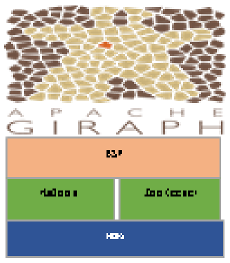


**Fig. 1. Layered Architecture of Giraph**

Giraph is a distributed system that helps the execution of graph mining processing on huge database. It implements Hadoop's MapReduce for processing graphs as shown in fig.1. In real case, the link between the Pregel System and Giraph Apache is similar to the link between Google MapReduce framework and Apache Hadoop project. Currently Giraph is used in Ranking (i.e.Popularity ,importance,etc), LabelPropagation(i.e. Location, school, gender, etc.) and Community(i.e. Groups, interest). Giraph implements all its HDFS (Hadoop Distributed File System) for processing tasks as map-only jobs on Hadoop and uses for graph datasets of as input and output.There are set of machines (workers) present to process large graph datasets[15].

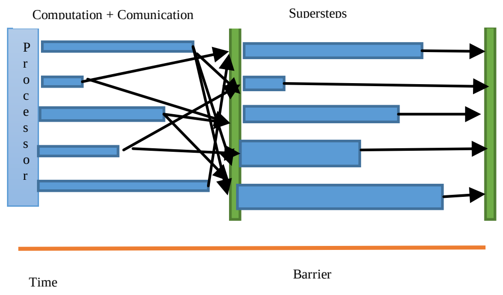


**Fig. 2. Supersteps Execution in BSP**

One of them is selected as a master of all other slave workers. The main task of the master is to perform global synchronization, error grasping, and assigning partitions to workers. The features of Pregels

(i.e. shared aggregators, edge-oriented input, etc) input are added in Giraph.
Built-in from Hadoop architecture, Giraph implements Apache Zookeeper for containing check pointing, maintaining configuration information, and distributed synchronization (fig.1).

### *A.Bulk Synchronous Parallel Model*

In BSP, a similar set of programs executes over a set of virtual processes and implements as a sequence of parallel supersteps difference by barrier synchronizations. In our proposed model, the superstep consists of three ordered phases as shown in fig.2. First one, *Global Communication*, where the vertex acquire its updated values from previous supersteps (iteration).Second one, *Local Communication* ,where the vertex value is exchanged between processes according to the request generated during local commutation. Third one, a *Barrier Synchronization* where vertex sends its new value to its adjacent vertex that will be available to them in next supersteps. At each iteration (supersteps) each vertex can be active or inactive. In our implementation, we specify the frequent subgraph mining computation on the graph vertices and transmission.

### B. How Giraph Differ from MapReduce?

The MapReduce framework uses key-value pair maps to provide the index of graph dataset as shown in fig 2. In real cases, MapReduce tasks have two problems, first from one iteration to another the data remains the same, which must be reloaded and reprocessed at each iteration that makes longer time for input/output operation, bigger network bandwidth, and more processor resources. Second, the endpoint of iteration requires more mapReduce tasks on each iteration, leads to more terms of scheduling extra tasks, reading more data from disk, and moving data accesses the network. Therefore we need more resources[17]. Apache Giraph, is a "Think like a Vertex" framework that supports synchronous timing model which is based on BSP as shown in table. In Giraph, each vertex is connected with a unique ID associated with related information. Giraph follows "think like a vertex", not like MapReduce, a key-value pair (Table: 1).

**Table-I**

| **MapReduce:** | **Giraph:** |
|---|---|
| Public class Mapper<KEYIN,VALUEOUT ,VALUEOUT>{void map(KEYIN key, VALUEIN value, Context context) throws IOException, InterruptedException;} | public class Vertex< I extends WritableComparable, V extends Writable, E extends Writable,M extends Writable>{Void compute(iterator <M> msgIterator);} |

## IV. OVERVIEW

*Graph Mining* is the process of extracting frequent subgraph from a Graph Database on the basis of user-given threshold value. To make faster execution we initially did it with a parallel system which now converted to a distributed system. In this paper , we implemented our OFSM(Optimized Frequent Subgraph Mining) algorithm in different distributed systems such as Giraph, Pregel, Graphlab, Hama and found out Giraph generates better results with our proposed model, and we use a number of optimized algorithms to make performance better.

### A. *Giraph in MapReduce*

In 2010, Google developed a Pregel system executed in C/C++ which supports BSP that is given a native API specifically programming graph method using a *"Think Like a Vertex "* computing paradigm where each vertex can have value associated with messages. At each supersteps, each vertex can aquaire messages, change its value and the message passing of the system is deputed in fig. 3.

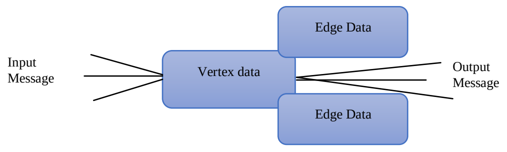


**Fig.3. State of Message Passing**

Hence, it leads the communication overheads. To overcome this problem, Pregel in Giraph conserves data locality by ensuring computation on locality stored data. The graph vertices are distributed over different processors of the cluster where each vertex and its connected set of neighbors are allocated to the same node. All the graph processing methods are represented as supersteps. Each step explains what each participating vertex has to compute. The edges between vertices represent communication channels for transmitting computation results from one vertex to another. In our proposed model, at each iteration, a vertex implements Optimized Frequent Subgraph Mining (OFSM), sends and receives messages (support value) to its neighbour (or any vertices with known ID), and changes active state to inactive. In Pregel all iteration are synchronous. In our proposed model, at each iteration, a vertex implements Optimized Frequent Subgraph Mining (OFSM), sends and receives messages (support value) to its neighbour (or any vertices with known ID), and changes its state from active to inactive. In Pregel all supersteps are synchronous.

### B. Giraph Components

Giraph becomes a popular graph framework due to growing in developers worldwide and community of users. Nowadays Facebook, Twitter, LinkedIn built its graph search service using Giraph. The basic principle graph processing task executes as map-only task on Hadoop and implements Hadoop Distributed File System (HDFS) to read and write the data. There are so many machines available to execute OFSM with different threshold values.

- *Master*

Anyone machine selected as Master which plays *Application Coordinator* with all remaining machines. There should be one active master at a time.

It is responsibility of master to assign the partition owners as workers before performing the supersteps and maintain the synchronizations.

- *Worker*

After selection of master all carry on processor treat as Worker to Computation and messaging among the vertices. It Loads the graph from input splits from the main graph datasets. They perform the communication or messaging of its assigned partitions[15].

- *Zookeeper*

The Zookeeper plays a vital role to achieve the synchronization between the distributed system, configuration of information, check pointing and failure recovery system, global application state. During high demand of clients, if any call has been failed, then immediate alternate ZooKeeper is active to perform all operations.

- *Input and Output of data format*

Generally Giraph supports three types of input formats (i.e. vertex input, edge input, and mapping input format. In the case of the first method, VertexInputFormat abstract class is used to read vertex values .If an edge information is to be study, either it can be inserted in the corresponding vertex or a separated Edge Input Format can be used. In the case of the second method, the EdgeInputFormat of the abstract class used to study graphs stored in the configuration of edges. Here it is not required to have all edges related to a vertex next to each other. The main problems arise that we can't read vertex values. Therefore, to overcome all problems, we use mapping input format to read graph datasets in the proposed approach. The desired result can be generated by either using Vertex OutputFormat (i.e.vertex per line) or EdgeOutputFormat (i.e. edge per line). In our implementation we use SimpleText Vertex OutputFormat and IdWithValueTextOutputFormat to produce results [12-16].

## C. Apache Giraph Data Flow

The Apache Giraph provides two interfaces to computing the entire life-cycle of a Giraph (i.e. MasterObserver and WorkerObserver) and three phases (i.e. Loading Phase, Compute/Iterate Phase, Offloading Phase).In this paper, we implement these interfaces and utilise for different tasks.

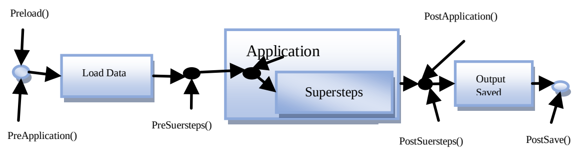


**Fig.4. Life-cycle of Giraph Application**

The fig.4 represents the life-cycle of Giraph application. The black colour circle shows the different functions are executed during the entire life-cycle. Before the application starts, the input data is stored in memory. Any number supersteps are implemented within the application [18]. The entire application completes and stores the output destination after the last supersteps complete its operations. In our experiment we use the methods for computation and superstep is given in Table II.

**Table-II: Methods for Giraph API**

| **Methods available to compute()<br>Immediate effect/access** |
|---|
| I getVertexId() |
| V getVertexValue() |
| Void setVertexValue(V vertexValue) |
| Iterator<I> iterator() |
| E getEdgeValue(I targetVertexId) |
| Boolean hasEdge(I targetVertexId) |
| Boolean addEdge(I targetVertexId,E edgeValue) |
| E removeEdge(I targetVertexId) |
| Void voteToHalt() |
| Boolean isHalted() |
| **Next superstep** |
| void sendMsg(I id, M msg) |
| void sendMsgToAllEdges(M msg) |
| void addVertexRequest(BasicVertex<I,V,E,M>vertex) |
| void removeVertexRequest( I vertexId) |
| void addedgeRequest(I sourceVertexId,Edge,I,E>edge) |
| void removeEdgeRequest(  I sourceVertexId ,I destVertexId) |

- *MasterObserver*

The source code for MasterObserver interface represented in fig.5.The method of interfaces are invoked only once i.e. at master node .Then the code is written write to global scope i.e. connected to the whole application. If the application fails, then the applicationFailed() is called. In our novel approach, we override the method and send an email to the application owner [12]. Then we check the number of supersteps already executed and put a trigger of action after each superstep(fig.5).

```
public interface MasterObserver extends
ImmutableClassesGiraphConfigurable
{
void preApplication () ;
void postApplication () ;
void applicationFailed ( Exception e ) ;
void preSuperstep ( long superstep ) ;
void postSuperstep ( long superstep ) ;
}
```

Fig.5. source code for MasterObserver intereface

- *WorkerObserver*

Due to the presence of a number of workers, any number of instances of workerObserver can be generated during Giraph application implementation. Therefore no need of global scope to the method of interface for writing the code. The methods can be implemented to keep track of individual workers. The source code is written for the WorkerObserver interface is shown in fig.6.

```
public interface WorkerObserver extends
ImmutableClassesGiraphConfigurable{
void preLoad () ;
void preSuperstep ( long superstep ) ;
void postSuperstep ( long superstep ) ;
void postSave () ;
}
```

**Fig .6: the source code for the WorkerObserver interface**

- *The Computation Interface*

It represents the calculation to be applied on all the alive vertices in each superstep. During each iteration, there can be any number of instances of this interface, each doing calculation on each partition of the graph's vertices. In our model we access the global data (like data from WorkContext), as an object of this interface alive for a single iteration [18]. There is one method compulsory to write is compute () as we need to calculate frequent subgraph mining based on the user-defined threshold value. We implement the method for our proposed model in fig.7.

```
public interface Vertex < I extends WritableComparable , V extends
Writable , E extends Writable > extends
ImmutableClassesGiraphConfigurable {
 void initialize ( I id , V value , Iterable < Edge > edges );
void initialize ( I id , V value );
I getId ();
V getValue ();
void setValue ( V value );
void voteToHalt ();
int getNumEdges ();
Iterable < Edge > getEdges ();
void setEdges ( Iterable < Edge > edges );
Iterable < MutableEdge > getMutableEdges ();
E getEdgeValue ( I targetVertexId );
void setEdgeValue ( I targetVertexId , E edgeValue );
Iterable getAllEdgeValues ( final I targetVertexId );
void addEdge ( Edge edge );
void removeEdges ( I targetVertexId );
void unwrapMutableEdges ();
void wakeUp ();
boolean isHalted ();
}
```

**Fig.7. Source code for Immutable Classes Giraph Configurable**

## V. GRAPH DISTRIBUTION

The Distributed Graph Mining platforms such as Giraph have substantially simplified the design and development of certain classes of distributed graph methods. The methods should be simple to implement and the executing infrastructures should not require major hardware and software investment. We use Giraph, a well-liked framework for distributing computing, that based on vertex-centric design.

### A. Master Graph Partitioner

It is used to generate initial partitions and then produce different partition that changes between supersteps. In our suggested framework, we use GraphPartitionFactoryInterface that defines the partitioning framework for this application (fig.8). Then the MasterGraphPartitioner interface, determines how to divide the graph into partitions, how to operate partitions and then how to allocate those partitions to workers. After the workers stats have been merged to a single list, the master can use this information to send commands to workers for any partition changes. Therefore, the Dynamic Partitioning needs to make balance during implementation by migrating vertices between workers when needed. The *master-slave architecture* is used as a TLAV framework, a master node starts off the slave worker, manages planning amongst workers, monitors the execution, stores global values such as a combiners. Each worker performs a copy of the program on local particulars and informs the master of runtime status [13].

### B. Worker Graph Partitioner

It determines which partition a vertex belongs to and creates/modifies the partition states (can split/merge partitions). *Worker Graph Partitioner* Stores the PartitionOwner objects from the master and provides the mapping of vertex to PartitionOwner(fig.8).

## VI. OPTIMIZATION TEXHNIQUES

In order to improve the flexibility of OFSM algorithm in Giraph and enhance its performance, we used different optimized algorithm with different datasets shown in Fig.8.

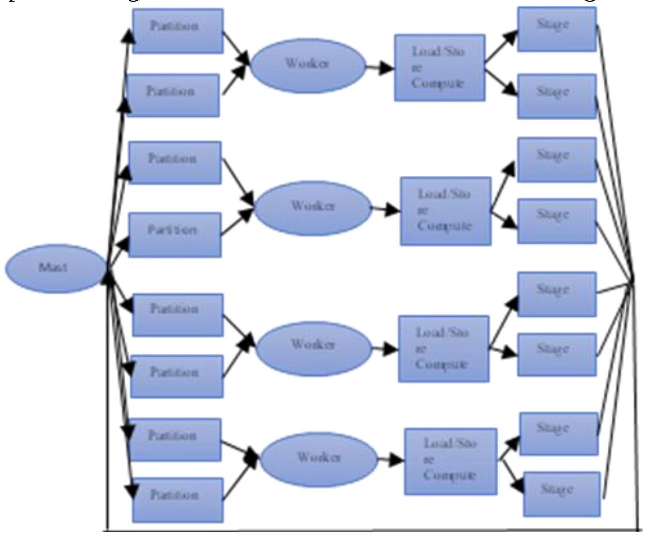


**Fig.8: Graph Distribution**

### A. Data Sharing Across Nodes

There are three way of data sharing across Giraph nodes: Broadcast, ReduceOperation and Aggregator. By using a String Identifier, individual data sharing approach has to be initialized in MasterComputer. In our implementation, we use *Aggregator* to perform data sharing (i.e. threshold value). Both master and worker nodes can use and change the value of aggregator function. This function needs to be initialized by master node and it has to be communicative and associative. After we initialize the aggregator, Giraph transmits a copy of the aggregator objects to each worker. Then the worker will have read-only access to current aggregator value that has been set by the master. By applying the function aggregator (String) work, apply a value change to its object, when all changes will be synchronized and applied on the aggregator object by the end of superstep. Both MasterCompute of the next iteration will have a compatible outlook of the latest update in the aggregator object that was modified by the worker of the last iteration. If the MasterCompute of new superstep does not change the value of aggregator, the worker would also be able to access the latest change in the aggregator object that was updated by last iteration (superstep) [15].

### B. MasterCompute

The *MasterCompute* is a non-compulsory phase that carries out the centralized computation in Giraph system before starting the procedure. We implement the OFSM at the worker nodes, we need to execute MasterCompute class in master node at beginning of each superstep. We use this stage to modify graph calculation class, such as message combiner and vertex compute class. During execution, we use this class to enable data sharing across the different vertices at the beginning of each supersteps. Then we use Giraph Configuration.setMaster ComputeClass() or property giraph. masterComputeClass to register its MasterCompute class[17].

### A. Coarse-grain processing

We use coarse-gain on each worker so that all workers have right access to globally defined aggregators, reducers and broadcasted values. The implementation of coarse-grain processing is consist of four methods: preApplication(), preSuperstep(), postApplication() and postSuperstep(). Before and after of each iteration and message exchange between workers, individual worker implements a set of four methods. The preApplication() is used as an initialization step that is implemented before starting the first superstep in Giraph, while postApplication()is implemented after the last superstep. During implementation, preSuperstep() is executed before beginning of the next superstep while postSuperstep() is executed after finishing the current superstep[11].

### B. Combiner

The basic goal of a combiner is to lessen the size of incoming messages to a vertex. We make it by notifying their values with help of the message combiner on incoming messages for each vertex [11]. The method computes over a list of incoming messages and generates one or more messages. We make the combiner function communicative and associative to confirm correctness of incoming messages to vertex. In Message combiner, the user collects a vertex ID, original messages and a new message. The PageRank, Shortest Path and Connected Component algorithm is used to enhance the performance.

## VII. PERFORMANCE OPTIMIXATION

During analyzing of our proposed algorithms, we use different optimization techniques by changing its run time parameters. All the optimization we performed fully depends on the cluster set up, hardware, graph input and output.

- **Out-of-core processing**

The Giraph purely depends on the operating system for providing more memory through *operating system disk swapping* method. If the cluster memory can't assemble the message generated by the user's algorithm regarding the input graph data, then there is a chance of failure or even to make the system slow[12]. Therefore, we use out-of-core process (explicit swapping) to efficiently split the least utilize graph partitions on cache the algorithm's messages temporally into disk.

- **Controlling the OutEdges class**

The ArrayListEdges, ByteArrayEdges, HashMapEdges, HashMultimapEdges data structure used to store the out-degree edges. In our designed model, we use ArrayListEdges to make balance between storage area and performance of edge computation.

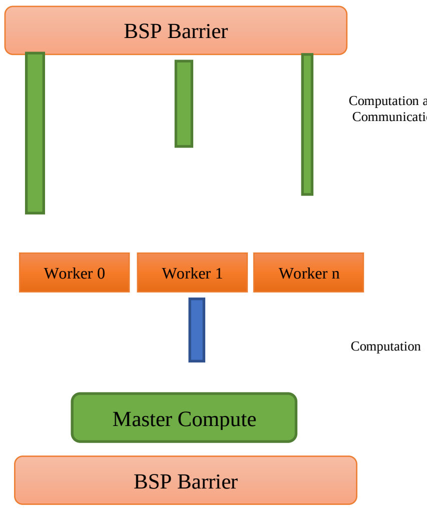


**Fig.8. MasterCompute**

- **Multithreading**

In case of Multithreading technique, the performance of the algorithm is enhanced by increasing the number of threads for individual. We use the multithreading to enable parallel computation for Giraph's supersteps, so it is easy to control the number of threads computations through property *giraph.numComputeThreads* In our proposed framework, Giraph's reading efficiency is expanded by increasing number of threads as per individual worker for each partitions of input graph dataset. To overcome the network overhead, we use it's *giraph.useInputSplitLocality* by forcing Giraph workers to prioritized reading local input splits before remote ones. By help of it's another property called *giraph.numOutput Threads*, which we make the output for multithreading purpose [13].

- **Message exchange tuning**

To optimize the message exchange in absence of combiner, Giraph supports three techniques (i.e byte_array_per_ partition, extract_bytearray_per_partition and pointer_list_ per_vertex). We use pointer_list_per_vertex technique to store the messages in a byte-array data format and generate pointers to them for destination vertices. It permits access to messages in byte-array data structure despite of higher overhead in storage area. We use the giraph.messageEncode AndStoreType property for control the message encoding[12].

## VIII. REECOVERY MANAGEMENT

We implement frequency checkpointing and put back the graph calculation in case of any failure occurred. The state of graph is check in each checkpoint and the graph check properly used to control the frequency of checkpointing(fig.8).

If any failure occurred, the current active master becomes idle and the next master becomes spare of it .

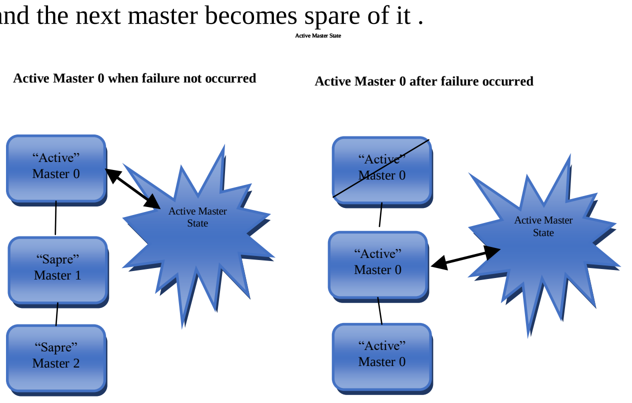


**Fig.8. Master Thread fault tolerance**

There are three ways to achieve the customer fault tolerance. First, if alive master dies then a new one will automatically takes the old master place (i.e in case of multiple master threads).Second, if a worker thread dies, the application rollback to previously checkpointed supersteps. The next superstep will start with succeeding of workers. Third, if a zookeeper server is not alive, then the application can be able to proceed till quorum is continue as shown in fig.9 [12-17].

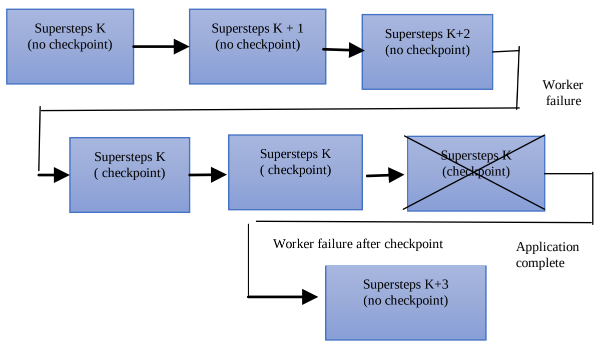

Supersteps K (no checkpoint)
Supersteps K + 1 (no checkpoint)
Supersteps K+2 (no checkpoint)
Worker failure
Supersteps K ( checkpoint)
Supersteps K ( checkpoint)
Supersteps K (checkpoint)
Worker failure after checkpoint
Application complete
Supersteps K+3 (no checkpoint)

**Fig.9. Worker thread fault tolerance**

## IX. EXPERIMENTAL IMPLEMENTATION AND ANALYSIS

In this section, we experimentally analysis our proposed approach for generating frequent subgraph using OFSM with different distributed system such as Pregel, Giraph, Hama, GraphLab. Some of the systems support synchronous and other asynchronous timing, we observe that the performance of individual system depends upon requirements its own optimized algorithms, but overall Giraph gives better result with OFSM algorithm. We utilise four real graph datasets as shown in below:

**Amazon Network**

It is a dataset that was gathered in 2003 from an Amazon website based on a creeping concept i.e. a customer who bought one item should take another fixed item. If a product *x* is frequently co-purchased with product y, the graph contains a directed edge from vertex *x* to vertex y.

**Facebook social sities**

It is a social dataset where vertices are represented by the profile of peoples and edges are represented by the messages sent and received between them.

**Google web graph**

The dataset is organized with a set of vertices (i.e. web pages) and directed edges (i.e. hyperlinks between them), that was collected in 2002 as a part of Google Programming Contest.

**Texas road network**

The dataset is a collection of road map of Texas where each node is used to represent the intersection and endpoints of road, the undirected edges are used to figure out the roads connecting these intersections and endpoints.

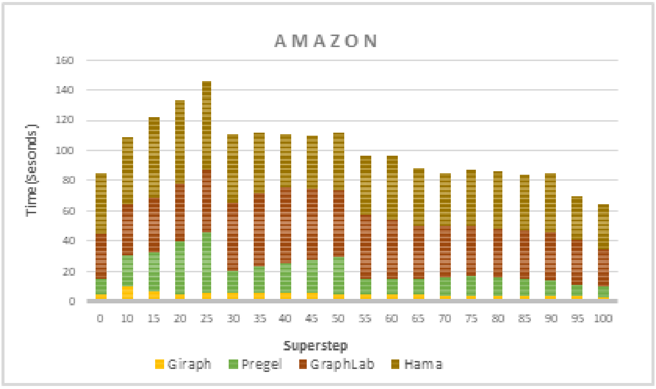
We consider all directed graphs and converted these into undirected graphs when needed in some implementation. We use 15 clusters, each server runs on Windows operating system and Java 1.2.Giraph, Pregel, Hama, GraphLab were implemented based on version download in March 2020 accordingly. In Giraph, each server was configured to run upto 10 workers simultaneously. When we use any mode of distribution, the OFSM algorithm runs on the same input with similar type of overhead cost such as reading input, setting up the algorithm, closing of the task and writing into the final format of output.

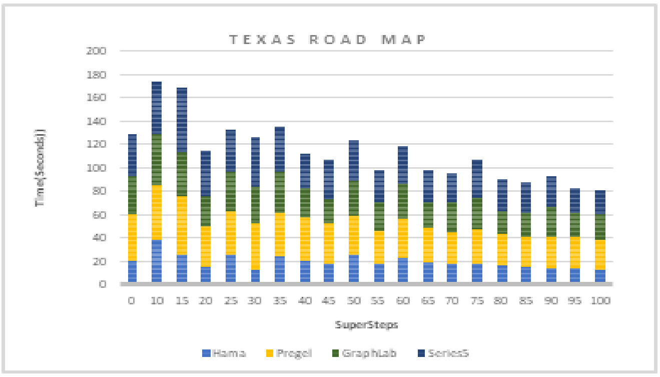
**Table-III: Datasets characteristics**

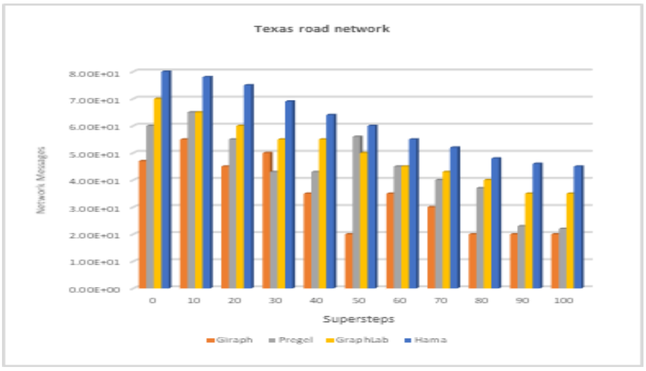
| dataset | #nodes | #directed edges | #Undirected edges |
|---|---|---|---|
| Amazon | 262111 | 1234877 | 1042012 |
| Texas | 1379917 | 1921660 | 1510530 |
| Google | 875713 | 5105039 | 4504321 |
| Facebook | 4039 | 88234 | 70123 |

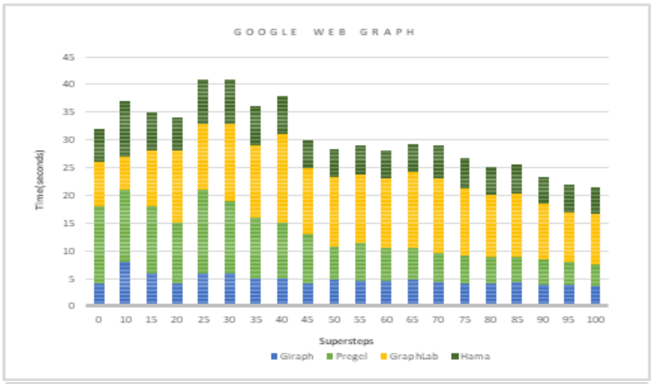
In our proposed model, the Graph Partitioning algorithm plays a vital role in a Distributed System. We use different supersteps ranging from 10 to 100. In case of no supersteps, we combine mastercompute and coarse-grain processing to optimize our algorithms. Adding the multithreading and message passing algorithm in superstep from 40 to 80 in different datasets, the performance is totally changed (fig.10). We conclude that Giraph has better performance than Pregel, GraphLab, Hama distributed system. The main reason behind it is that we use OFSM which gives the result fully dependent on user threshold value.

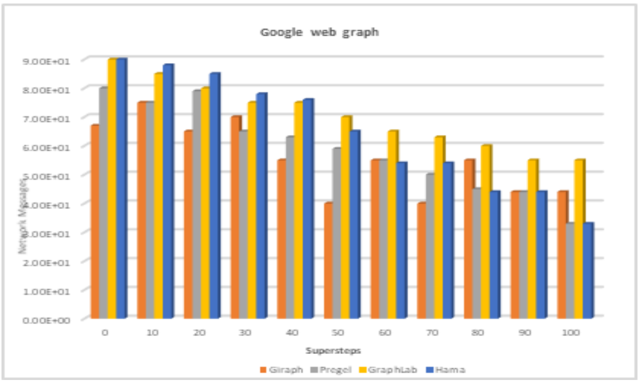
In our proposed model, the message exchange with multithreading is used and we compare how the network messages are affected with different values of supersteps(Fig.11).











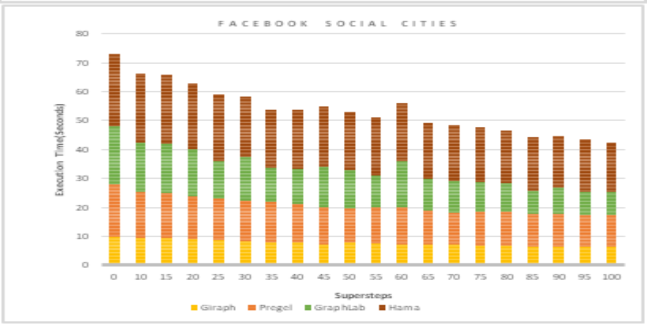


Fig.10. The execution time with different number of supersteps in four datasets.

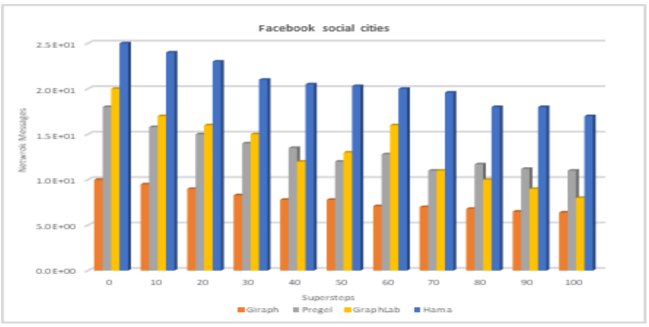
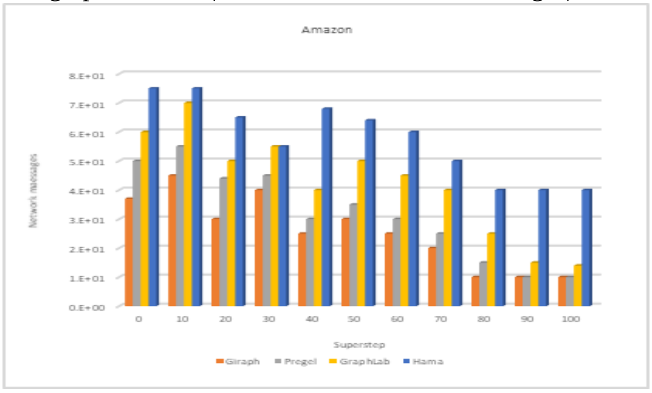


Fig.11. The network messages with different number of supersteps in four datasets

In amazon dataset, for certain values of supersteps we get better results (i.e. 80, 90,100). For Texas road network datasets, the same result we also get from 50, 70,100 respectively. But in the case of Google web graph, we found drastic changes in the results (i.e. 50, 70, and 90,100). In the case of facebook social sites, it is over all the same for any value of supersteps. Therefore, we concluded from our dataset that network message overhead fully depends on input size graph datasets (i.e. number of vertices and edges).



Giraph is based on one master and any number of workers (i.e. Master-Slave architecture). In our experiment, we use different ranges of workers and do a good comparative study with all four graph datasets (Fig.12). We found, in some cases less worker better results, that's reflected in how to utilize network congestion and optimization techniques.

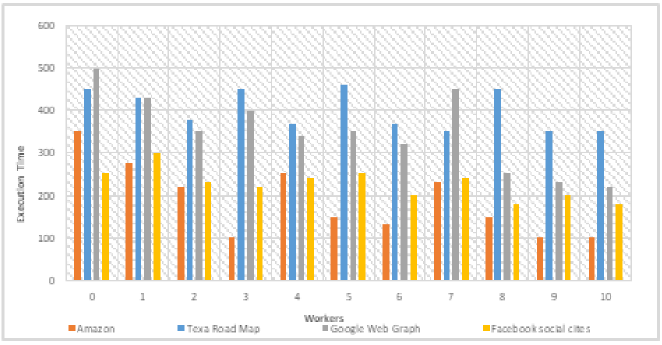


Fig.12: Execution time with numbers of workers of four Graph Datasets

*Published By:*
*Blue Eyes Intelligence Engineering*
*& Sciences Publication*
*© Copyright: All rights reserved.*

## X. CONCLUSION AND FUTURE WORK

In summary, TLAV framework fully overcomes the challenges of large graph database processing. Our proposed model is based on *OFSM* and *vertex-centric processing* for distributed graph mining which we can use different timing, communication, execution mode and graph partitioning algorithm to get the better results. By implementing optimized techniques (i.e. Data sharing across no des, MasterCompute, Coarse-grain processing, Combiner), and Performance Optimization (i.e. Out-of-core processing, Controlling the OutEdges class, Multithreading, Message exchange tuning), we are able to get faster result with fewer worker in Giraph system. Throughout our experimental study, we observed that performance wise improvement of OFSM is with fixed threshold values. In future, we focus on to elaborate our study with different threshold values and distributed systems in-depth.

## REFERENCES

1. Amr Ahmed, Nino Shervashidze, Shravan Narayana murthy, Vanja Josifovski, and Alexander J. Smola. 2013. Distributed Large-scale Natural Graph Factorization. In Proceedings of the 22Nd International Conference on World Wide Web (WWW '13). International World Wide Web Conferences Steering Committee, Geneva, Switzerland, 37–48.
2. Una Benlic and Jin-Kao Hao. 2013. Breakout Local Search for the Vertex Separator Problem. In Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI '13). AAAI Press, 461–467.
3. Nguyen Thien Bao and Toyotaro Suzumura. 2013. Towards Highly Scalable Pregel-based Graph Processing Platform with x10. In Proceedings of the 22Nd International Conference on World Wide Web Companion (WWW '13 Companion). International World Wide Web Conferences Steering Committee, Geneva, Switzerland, 501–508.
4. Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. 2010. Twister: A Runtime for Iterative MapReduce. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10). ACM, New York, NY, USA, 810–818.
5. William W Hager, James T Hungerford, and Ilya Safro. 2014. A Multilevel Bilinear Programming Algorithm For the Vertex Separator Problem. arXiv preprint arXiv:1410.4885 (2014).
6. Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Ozsu, Xingfang Wang, and Tianqi Jin. 2014. An Experimental Comparison of Pregellike Graph Processing Systems. Proceedings of the VLDB Endowment 7, 12 (2014), 1047–1058.
7. Borislav Iordanov. 2010. HyperGraphDB: A Generalized Graph Database. In Proceedings of the 2010 International Conference on Web-age Information Management. Springer-Verlag, Berlin, Heidelberg, 25–36.
8. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10). ACM, New York, NY, USA, 135– 146.
9. Tomasz Kajdanowicz, Przemyslaw Kazienko, and Wojciech Indyk. 2014. Parallel Processing of Large Graphs. Future Gener. Comput. Syst. 32 (March 2014), 324–337. DOI:http://dx.doi.org/ 10.1016/j.future.2013.08.007.
10. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10). ACM, New York, NY, USA, 135– 146. DOI:http:/ /dx.doi.org/10.1145/ 1807167.1807184.
11. Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2012a. Accelerate Large-scale Iterative Computation Through Asynchronous Accumulative Updates. (2012), 13–22.
12. Xianke Zhou, Pengfei Chang, and Gang Chen. 2014. An Efficient Graph Processing System. In Web Technologies and Applications, Lei Chen, Yan Jia, Timos Sellis, and Guanfeng Liu (Eds.). Lecture Notes in Computer Science, Vol. 8709. Springer International Publishing, 401– 412. DOI:http://dx.doi.org/10.1007/ 978-3-319-11116-2_35.
13. Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. 1995. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (Supercomputing '95). ACM, New York, NY, USA, Article 37.
14. Jelica Protic, Milo Tomasevic, and Veljko Milutinovic (Eds.). 1997. Distributed Shared Memory: Concepts and Systems (1st ed.). IEEE Computer Society Press, Los Alamitos, CA, USA.
15. Serafettin Tasci and Murat Demirbas. 2013. Giraphx: Parallel Yet Serializable Large-scale Graph Processing. In Proceedings of the 19th International Conference on Parallel Processing. Springer-Verlag, Berlin, Heidelberg, 458– 469.
16. Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billionnode Graphs on an Array of Commodity SSDs. In Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15). USENIX Association, Berkeley.
17. Yue Zhao, Kenji Yoshigoe, Mengjun Xie, Suijian Zhou, Remzi Seker, and Jiang Bian. 2014. LightGraph: Lighten Communication in Distributed Graph-Parallel Processing. In Proceedings of the 2014 IEEE International Congress on Big Data (BIGDATACONGRESS '14). IEEE Computer Society, Washington, DC, USA.
18. Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12). USENIX Association, Berkeley, CA, USA, 17– 30.

## AUTHORS PROFILE

**Ms Sadhana Priyadarshini** is a Phd scholar in Department of Computer Science and Engineering at GITAM (Deemed to be University), Vishakhapatnam, India She completed MTech(CSE) from SQA University in 2010.Her research interests in field of Data Mining.

**Dr. Sireesha Rodda** is a Professor in the Department of Computer Science & Engineering, GITAM (Deemed to be University). She has 17 years of research experience in the fields of Artificial Intelligence, Data Mining and Machine Learning. She has more than 30 papers published in referred journals