SE256 Project:Comparison of performance of common algorithms when using Map Reduce and Giraph

Abhilash Sharma
Department of Computational
and Data Sciences
Indian Institute of Science
Email: abhilash@grads.cds.iisc.ac.in

Prateeksha Varshney
Department of Computational
and Data Sciences
Indian Institute of Science
Email: prateeksha@grads.cds.iisc.ac.in

***Abstract*—Hadoop and Giraph are some of the frameworks for processing large scale graphs, so it would be interesting to get a comparative performance of Hadoop and Giraph for some algorithms that are commonly used in social networks . On one hand map reduce views world as key , values whereas Giraph has Think like a vertex approach. Most graph algorithms are iterative in nature and involve repeatedly iterating over the graph states. Both Hadoop map-reduce and Giraph use different approach for storing the graph states. So it would be interesting to know what kind of leverages can we get when we use one over the other for such algorithms.**

Keywords*—*Map Reduce, Giraph

I. Introduction

According to Jon Bruner, ”Big Data is the result of collecting information at its most granular level its what you get when you instrument a system and keep all of the data that your instrumentation is able to gather.” When data generated by instruments becomes so large that we cannot do analytical processing using traditional methods, there was a need for Big Data frameworks. Google introduced Map Reduce framework that processes data as key value pairs, it gave a new method for processing big data, but it was not efficient in processing graph structured data, graph algorithms were more often than not iterative in nature, which required map reduce to repeatedly pass state from one map reduce job to another by writing data to disk. Giraph was introduced which was iterative map only framework which addressed most of the limitations of map reduce model when processing graph structured data, it uses messages to pass states thus avoiding the need to read from disk between iterations.In this project, we aim to compare the performance of common social network algorithms and find out how much leverage does giraph provide(if any) while processing graph structured data.

Our deliverables in this project are:

- Common social network algorithms implemented in Hadoop and Giraph.
- A systematic comparison of Hadoop and Giraph to study how their different approaches impact their performance for different algorithms.
- Using metrics like running time, memory usage, network traffic together with different datasets and algorithms, to explain the performance of Hadoop and Giraph.
- Analysis of benefits and limitations of each framework for social network graph data processing.

A. Hadoop Map Reduce

Hadoop MapReduce is an Apache implementation of Map Reduce [3]. Map reduce Framework enables expressing algorithms involving processing or generation of data into two functions. Input file is read from HaDoop File System(HDFS), apache’s implementation of Google File System(GFS) and passes it as $< key, value >$ to Mapper. Map function runs for each $< key, value >$ and may transforms it into another set of $< key1, value1 >$ and passes it to Reducer. Reduce function gets the key and list of values $< key1, < value1, value2... >>$ and runs for every key, it may process the list of values and may emit one or more key values as output. This is simple overview of map reduce paradigm, though it explains the programming paradigm, it leaves key architectural components of map reduce out. The Combiner if specified may run to reduce the communication traffic between mapper and reducer. Shuffle does a groupBy of keys and Sort component sorts the $< key, value >$ pairs at each reducer,partitioner which takes $< key, value >$ as input and returns the partition(reducer) to which this key belongs to. Making good use of these components is imperative as most of the time one of these becomes bottleneck of the Job.

B. Apache Giraph

Apache Giraph[2] is an open-source alternative to the proprietary Pregel which is a vertex-centric model and uses iterative BSP computation. Giraph uses HDFS for data input and output. It runs workers as map-only jobs on Hadoop. For the purpose of checkpointing, coordination and failure recovery, Giraph uses Apache ZooKeeper. Giraph supports various input formats for graphs. The computation logic is written from the perspective of a vertex (”think like a vertex approach”). Computation proceeds as a sequence of iterations, called supersteps in BSP. In the superstep 0, every vertex is active. In each superstep each active vertex invokes the

Compute method provided by the user.
The Compute method:
1.receives messages sent to the vertex in the previous superstep,
2.computes using the messages, and the vertex and outgoing edge values, which may result in modifications to the values, and
3.may send messages to other vertices.
There is a barrier between consecutive supersteps. This means that the messages sent in any current superstep get delivered to the destination vertices only in the next superstep, and vertices start computing the next superstep after every vertex has completed computing the current superstep.There is no guaranteed message delivery order. Each message is delivered exactly once. A vertex can send messages to any node, though typically to neighbours. Values of vertices are retained across barriers. Any vertex can deactivate itself after any superstep. The vertex simply declares that it does not want to be active anymore.This can be done by calling the *voteToHalt()* method. However, any incoming message will make the vertex active again.
The computation halts after all vertices have voted to halt and there are no messages in flight. Each vertex outputs some local information, which usually amounts to the final vertex value. Giraph also has some additional features like *combiner* and *aggregators*. It is costly to send a message to another vertex that exists on a different machine. User can specify a way to reduce many messages into one value by overriding the *combine()* method. It is the same as *combiners* in *Hadoop*. *Aggregators* enable global computation. During a superstep, vertices provide values to aggregators. These values get aggregated by the system and the results become available to all vertices in the following superstep. There are two types of aggregators in Giraph : *regular* and *persistent* aggregators. The value of a regular aggregator will be reset to the initial value in each superstep, whereas the value of persistent aggregator will be available through the application.

II. Algorithmic Workloads

Algorithms considered for evaluating both the frameworks are PageRank,Triangle Counting, HashTag Aggregation,Transitive Closure and Local Clustering Coefficient. These algorithms are few of the common algorithms that are used widely for graph analytics. These algorithms cover both iterative and non-iterative workloads.In this section, overview of algorithms used along with the implementation strategy in both Map Reduce and Giraph is described(Excluding HashTag generation which is done only for Map Reduce to generate hash tag data for 10 timesteps using epidemiological model).

A. PageRank

PageRank is used to measure the importance of website pages(vertices), it uses the intuition that a page is more important if many pages have links(edges) to this page, initially it gives equal ranks to all webpages. At iteration of page rank calculation, a particular page distributes its page rank to pages that it has links to equally, it calculates its own page rank as sum of the PageRanks that it receives from pages that has a link to this page. This is an iterative algorithm which can be ran until convergence or predefined number of iterations. We evaluate PageRank algorithm on LiveJournal graph.
In Map Reduce this is implemented by taking the adjacency list(web graph) as input, and a set which stores all vertices of the web graph. In a map phase, each vertex uses its adjacency list to distribute the page rank. It also sends its adjacency list to the reduce phase so that the output contains the adjacency list which is used as input in the next iteration. In reduce phase, each vertex sums up the values to calculate its current page rank as 0.15/(Total number of vertices) + 0.85*(sum of page ranks received) and attaches it with the adjacency list and writes it to HDFS which is read by next iteration as input. To handle dangling vertices, after every 10 iterations, the PageRank of dangling vertices is summed up in the Mapper Task and distributed among all the vertices in the graph during cleanup. We ran our implementation of PageRank algorithm for 30 iterations for all our experiments.

In giraph, page rank is calculated using 30 supersteps where each superstep represents an iteration of page rank. In *superstep 0* each vertex has its initial value set to 0. Each vertex sends value/(no. of edges) to all its neighbours. In all further supersteps >=1 , each vertex sets its value equal to 0.15 / (total no. of vertices) + 0.85 * sum where sum = sum of the messages received by the vertex and sends message to all its neighbours where message = value/(no. of edges). This process continues till superstep 30. At superstep 30, all vertices vote to halt and the job terminates.

B. Triangle Counting

Triangle counting[5] is widely used for finding clustering coefficient of a graph.Global clustering coefficient is based on number of closed triplets in the graph. It measures degree to which graph tend to cluster together. We evaluate triangle counting on Live Journal Graph.
We implemented triangle counting in two sequential Map Reduce Jobs, first job finds the two length paths in the graph, while the second job finds the number of closed triplets in the two length paths. To elaborate this, the input to the first job is edge list of graph, for edge list entry u, v,the map phase emits source as key and sink as value, in reduce phase source has its adjacency list entry in values, to find all two length paths from this, we find combinations of neighbours, for a key value-list $u, < v1, v2, ... >$, $v1 - u - v2$ is example of two length path. The reducer emits all such pairs $< key = uvalue = v1, v2 >$. The second job takes original edge list as input as well as output of the previous job, mapper emits $< key = u, v \;\; value = \$ >$ for every edge, for an entry $< key = u \;\; value = v1, v2 >$ it swaps the key and value and emits it. In reduce phase test for triangle is done, if for key u, v we find $ as value then there exists a triangle with vertices in key and each of the vertices in values except $. This is counted and emitted by reducer.

Triangle counting in Giraph is done in 3 supersteps. The input to the giraph job is the **Live Journal** graph in JSON input format. For this a seperate job is written to convert the edge list into JSON format for giraph. In *superstep 0* each vertex propogates its id to all of its neighbours which have a higher id. In *superstep 1* each vertex forwards the received message, containing the ID of neighboring vertices with lower IDs to all vertices that have higher ID than the vertex that received the message. In *superstep 2* it is detected whether a triangle has closed after a round of propagating and forwarding vertex IDs. At this point, if the current vertex receives a message with an ID that corresponds to one of its neighbors, it means that the current vertex participates in a triangle with the vertex with the specific ID. For each such message, the current vertex increases a counter. The final value of the counter is the number of triangles in which the current vertex has the maximum ID. As a final result, each vertex will be having a value which indicates the number of triangles in which it has the maximum id. Thus if we sum up the values of all the vertices, we get the total number of triangles present in the input graph.

C. HashTag Aggregation

HashTag aggregation[7] is widely used in data analytics, for finding subject popularity, political polarization, subject attention spikes, user interests etc. In Map Reduce, Hash Tag aggregation is done in one job. A timestep contains hashtags generated by all vertices in that timestep.Input to Map Reduce Job is HashTags Generated in all timesteps. Map method in Mapper task gets vertex and list of hashtags, it searches for a particular hash tag and emits $< HashTag, 1 >$ for successful search as well as $< VertexID, 1 >$ for each occurrence. The reducer aggregates the total count of Hashtag as well as count per vertexID. Combiner is used to reduce the network traffic by performing map side aggregation.

In giraph, the live journal graph is given in JSON format where value of each vertex is equal to the list of hashtags separated by $ tweeted by the user in a given timestep. Hashtag aggregation is to be done across multiple timesteps, so each timestep file is given input to giraph sequentially. For each vertex, its list of hashtags is searched for a particular hashtag ”x”,and the vertex sets its value equal to the count of that hashtag. Also a persistent aggregator is used to sum up the counts of all the vertices. Before processing the second timestep file, a map reduce job is run to perform a join operation between the output of previous file and the next file. This is done to facilitate state passing across timesteps in giraph which otherwise would not have been possible. Thus the output of the join will be <v count$list-of-hashtags>where count is the count of the hashtag of vertex v in the previous timestep and list of hashtags are the hashtags separated by $ which are tweeted by vertex v in the next timestep. Now this output of join is given as input to giraph job. The giraph job simply counts the occurence of hashtag ”x” among the list of hashtags and adds it to count of previous timestep. This process is repeated for all timesteps. In other words, giraph job and map reduce job are executed alternatively where giraph job performs counting and map reduce job passes the states of vertices to next timestep. At last, we get statistical summary of hashtag ”x” across all the timesteps and also for each vertex.

D. Transitive Closure

Transitive closure is often used to find connectivity or reachability from source vertex. We evaluate Transitive closure for CitPatents graph. This is implemented in Map only iterative job, each job takes a list of vertices and reads them in a hashmap, it then performs one step of Breadth First Search algorithm using vertices in hashmap as source vertices and emits a list of output vertices after performing one step of Breadth First Search. These set of vertices are read in hashmap in next iteration, this goes on upto a certain depth specified. Elaborating the above algorithm further in map reduce terms, map method in Mapper task gets a vertex and its adjacency list as input, it checks if vertex is in the hashmap, if it is then it simply emits its neighbours as output. The next iteration takes output of previous iteration and reads them in the hashmap and this goes on. In case of finding reachability between two patents, a target vertex is also specified.Instead of performing BFS upto a depth, algorithm terminates when it finds the target vertex.

In giraph, a source vertex id ”x” is specified from which we wish to find the reachability of another vertex ”y”. So in *superstep 0* each vertex checks whether its id is equal to x, if yes then it sends message to all of its neighbours and votes to halt. This is done to perform one step of BFS as all the neighbours will become active due to receipt of messages. In all the further supersteps, each vertex checks whether its id is equal to ”y” or not. If yes then the job halts else the vertex sends message to all of its neighbours and votes to halt. This process is continued until we find the vertex ”y” or we reach a predefined number of maximum supersteps.

E. Local Clustering

Local clustering coefficient is a metric that measures how close a particular vertex and its neighbour are to form a clique or complete subgraph. Local Clustering coefficient is calculate as ratio of number of edges between a vertex and its neighbours to Total Number of edges possible. We evaluate Local Clustering on truncated LiveJournal graph that consists of users that used a particular hashtag in a timestep.

In Map Reduce, Local Clustering coefficient for each vertex is calculated in one Map Only Job. The input to this job is a adjacency list . To do it in one Map Only Job, we read the adjacency list in a HashMap also, which helps in finding number of edges between a vertex and its neighbours. In Mapper, for each vertex and its adjacency list, we put vertex and its neighbours in a set S, we iterate over S and read the adjacency list of each vertex in S and count the number of edges in which both source and sink both belong to S, at the end we emit the clustering coefficient as per the definition.

Local clustering is implemented in giraph using 3 supersteps. Input to the giraph job is the Live Journal graph in JSON input format where value of each vertex is a list of hashtags tweeted by that vertex at a given instant of time. For giving the value as Text, a seperate input format *JsonLongTextFloatDouble* is implemented. The hashtag data is obtained by the *hashtag generation job*. In *superstep 0* each vertex checks its value to see whether it has a particular hashtag ”x” in its hashtags list. If it is present, then the vertex sends its ID to all of its neighbours as message. In *superstep 1* each vertex iterates through all the messages and concatenates them to form an adjacency list seperated by comma. It then sets its value equal to this adjacency list. At the end of this superstep, each vertex will be having an adjacency list which will have only those neighbouring vertices which have the hashtag ”x”. Also each vertex send its adjacency list as message to all of its neighbours. In *superstep 2* each vertex adds all of the vertices of its adjacency list and itself into a hashset S. Then we iterate over S and read the adjacency list of each vertex in S (obtained as messages from the previous superstep) and count the number of edges in which both source and sink both belong to S and clustering coefficient is calculated as per the definition. At the end each vertex sets its value equal to the clustering coefficient and votes to halt.

F. HashTag Generation

To generate hash tag using epidemiological model, at each instant, map reduce job takes input three inputs, first is the adjacency list of social network graph, second is the set of previous hashtags of each user or vertices which is loaded by each mapper in setup, third a set of immutable list(in this timestep) of new hash tags. We refer an instant of hashtag generation as a timestep. At any timestep, a particular user have three choices, generate a new hashtag, choose from his set of old hashtags or choose a hashtag from one of the neighbours. We model this as three probabilities, finding these three probabilities to match real world require actual datasets, hence this is out of scope of this project. We considered these three probabilities as hyper parameters. We set probability of generating new hashtag as 0.3, choose old hashtag as 0.2 and choosing a hashtag from a friend as 0.5 while generating dataset for each timestep. At any timestep, upper bound on number of hashtags generated by a user is set to 5. For a particular user, a random number between 1 to 5 is chosen to set the number of hashtag N that is generated by this user in this timestep. To choose the type of a Hash Tag(i.e new,old,friend) for each of N hashtags, the probabilities of the types are used. This gives the type of each hashtag, if the type of hash tag to be generated is new then we use immutable list of new hash tags to choose. If the type of hash tag is old, then we use the set of old hashtags and select according to probability that is inversely proportional to difference between current timestep number and timestep number of that hashtag. If hashtag to be generated is from one of the neighbours, then one of the neighbours from the adjacency list is selected randomly and a hashtag is chosen from neighbours set of old hashtags(according to probability for selecting own hashtag). We generated hashtags for 10 timesteps.

III. Performance Metrics

The performance Metrics used to evaluate both Frameworks subject to workloads described above are:

- MakeSpan:End to End run time of each of the algorithms.
- Scalability:Using synthetic graphs of different sizes, we measured the scalability of each of the algorithms, we ran our algorithm implementations on 7 synthetic graphs and 1 original graph.
- Resource Utilization:We captured CPU utilization, Memory Utilization and Network Traffic during the makespan of the algorithm, this helped in analysis of performance of each of the algorithms.

IV. Experimental Setup and Evaluation

We used the SSL cluster for our experiments.

A. Hardware configuration

- SSL Cluster configuration
  - Nodes: 11 nodes
  - CPU Configuration: AMD Opteron(tm) Processor 3380
  - RAM: 32GB, DDR3

B. Software configuration

SSL cluster is configured to run in Java 1.7 environment.

SSL cluster has Hadoop 2.6.0 installed along with Giraph 1.1.0

C. Profiling Tools

We used Ganglia that is deployed on SSL cluster to collect Resource Usage data. It is made sure that no other jobs(except daemons and other essential processes) are running so that resource usage data collected by Ganglia is as accurate as possible.Monitoring daemon are configured in each node while Meta daemon is configured on head node.

D. Datasets

Dataset	Number of vertices	Number of edges
Patent citation graph	3774768	16518948
Live journal	3,997,962	34,681,189

Also we have generated synthetic graphs of 7 different sizes(keeping the edge to vertex ratio constant) using R-MAT along with original graphs for scalability experiments.

E. Evaluation

- Makespan: Table 1 on the next page lists the makespan of the algorithms on both Hadoop and Giraph. For each algorithm, the table shows setup time (once all the machines are allocated, how much time it takes before starting to read input), total time of each superstep(in ms), total time. For Hadoop, table shows the time taken by each job, map and reduce phase times(in ms), and the total time. Analysis of the makespan of all the algorithms is given below :

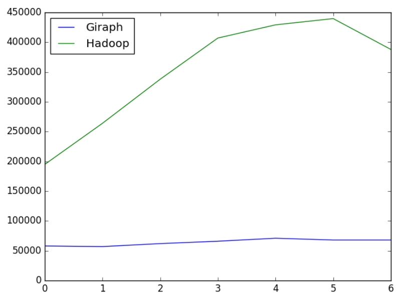

Fig. 1. Page Rank Weak Scaling

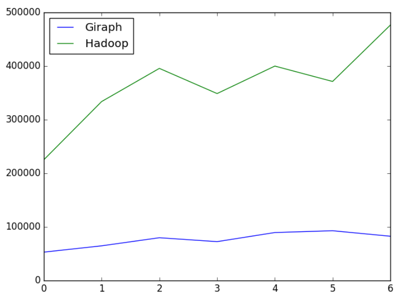

Fig. 2. Transitive Closure Weak Scaling

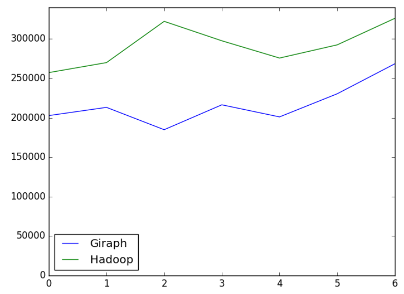

Fig. 3. Triangle Counting Weak Scaling

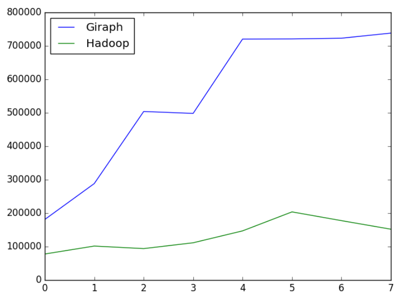

Fig. 4. HashTag Aggregation Weak Scaling

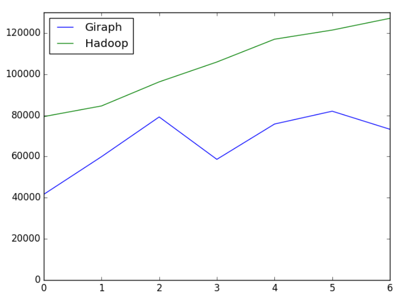

Fig. 5. Local Clustering Weak Scaling

TABLE I. MAKESPAN OF DIFFERENT ALGORITHMS (IN MILLISECONDS)

Algorithm	Giraph					Hadoop						
	Setup	Superstep 0	Superstep 1	Superstep 2	Total	Job 1		Job 2		Job 3		Total
						Map	Reduce	Map	Reduce	Map	Reduce	
Clustering	3282	10373	40642	9627	95146	22988	25509	28612	17401	25113	32073	151698
Page rank	3269	4152	2012	2247	62000	258502	170520	114468	32605	113368	35021	573262
Triangle counting	3007	22594	212169	7342	268602	48892	45882	120019	111447			326242
Hashtag aggregation	31516	89130			738280	90334	61682					152016
Transitive closure	3107	8264	7993	8096	98055	35606	26472	36313	14648	35988	20259	282136

Clustering: For Giraph, superstep 1 takes the maximum time among the 3 supersteps. This is because in superstep 1 (as discussed above in the algorithm description) each vertex iterates through all the messages and concatenates them to form an adjacency list seperated by comma. It then sets its value equal to this adjacency list. At the end of this superstep, each vertex will be having an adjacency list which will have only those neighbouring vertices which have the hashtag x. Also each vertex send its adjacency list as message to all of its neighbours. This superstep involves 2 time consuming tasks as compared to only 1 task in the other 2 supersteps.
From the table below it is visible that Hadoop takes much more time for clustering as compared to giraph. As Clustering is done in 3 sequential jobs, which require output from previous is read as input to the current iteration.

Triangle counting: For Giraph, superstep 1 takes the maximum time among the 3 supersteps. This is because in superstep 1 (as discussed above in the algorithm description) each vertex iterates through all the messages and for each message it iterates on all its edges to check whether to forward the message to given edge or not. Thus this superstep involves 2 loops(one inside other) as compared to only 1 loop in both the other supersteps.
Triangle counting is done in two sequential map reduce jobs, apart from the fact that the output is passed between jobs by writing to disk, it also requires more computing because it finds two length paths in reducer of first job which is proportional to square of number of neighbours.

Hashtag aggregation: We have run hashtag aggregation algorithm for 10 timesteps. For giraph, the table lists the setup time and the time of superstep 0(because hashtag aggregation takes only 1 superstep.) Setup time for this algorithm is the sum of the setup times for the 10 timesteps. The time for superstep 0 is the sum of the superstep 0 time for all the 10 timesteps.
From the table it can be seen that Hadoop takes much less time as compared to Giraph for hashtag aggregation. This is primarily because hashtag aggregation internally does just word count and does not require graph kind of structure. Giraph processes each time step file sequentially, one after the other, passing the state from one timestep to other and finally summing it up. Hadoop on the other hand takes

all the time step files together as input and simply runs word count and outputs the value. Also since Giraph processes each time step file sequentially, it reads each time step file from hdfs processes it and writes to hdfs whereas Hadoop reads all the files at once, processes them and writes to hdfs finally only once. All these factors make Hadoop better for hashtag aggregation.

Transitive closure: For transitive closure, the table lists the times taken for the first 3 supersteps in Giraph and first 3 jobs in Hadoop. This is because transitive closure involves multiple supersteps/iterations and therefore we have listed only the first 3 superstep/iteration times as these times are approximately same and involve the same operations. Hadoop takes much more time as compared to Giraph as expected. This is because finding closure is an iterative job. To find the reachability of a vertex ”y” from a source vertex ”x” requires as many iterations as the depth of the vertex ”y” from ”x” in Hadoop. While in Giraph this is done in multiple supersteps instead of iterations. In Giraph all the computations are done in memory once the file is loaded while each Hadoop iteration involves loading the file, processing it, and writing the output. In subsequent iterations, in addition to loading the file, the output from previous iteration is also loaded into the hashmap(as discussed above in the algorithm description). All these factors increase the time taken by Hadoop.

Page rank: For page rank, the table lists the times taken for the first 3 supersteps in Giraph and first 3 jobs in Hadoop. This is because page rank involves 30 supersteps/iterations and therefore we have listed only the first 3 superstep/iteration times as these times are approximately same and involve the same operations. Hadoop takes much more time as compared to Giraph. This is because page rank is an iterative algorithm. Finding the page rank requires 30 iterations in Hadoop while in Giraph this is done in 30 supersteps instead of iterations. In Giraph all the computations are done in memory once the file is loaded while each Hadoop iteration involves loading the file, processing it, and writing the output. In subsequent iterations, the output of the previous iteration is loaded as input in the next iteration(as discussed above in the algorithm description). Page rank requires the knowledge of graph structure and requires retaining the graph structure across the iterations along with page rank calculation. In other words, the adjacency list of a vertex has to be passed to reducer along with the calculated page rank. Thus in Hadoop we dont get any advantage of the graph structure, while in Giraph we only need to send the page rank as messages without requiring to pass the graph structure. All these factors increase the time taken by Hadoop.

- Scalability: X-axis shows the file size(accordingly increasing the processors) and Y-axis shows the time taken in ms.

Clustering: For clustering Giraph almost shows weak scaling behaviour with a slight increase in the time. This increase can be attributed to the fact that superstep 1 is the most time consuming superstep out of the 3 and involves each vertex iterating through all the messages and concatenating them to form an adjacency list seperated by comma. At the end of this each vertex send its adjacency list to all its neighbours. So as we increase the size of file the number of messages and the size of messages also increase and thus increase the overall time. Hadoop does not scales weakly for clustering which is done in three jobs, this can amounted to two reasons, first reason is passing of states(output) between jobs and other reason is size of hashmap required to find Closed form of the adjacency list found in first job, which is observed to increase linearly.

Triangle Counting: Both show weak scaling initially, but gradually weak scaling breaks. Makespan of Hadoop Job is higher than Giraph because of passing state by writing output of first job to disk while giraph performs everything in memory in 3 supersteps. From figure 3 we can see that weak scaling breaks at the end for giraph, because number of messages exchanged between workers start to dominate Synchronization time between supersteps. As workers are increased, the number of messages communicated between workers also increases.

HashTag aggregation: As discussed in the makespan section, Giraph takes more time as compared to Hadoop for hashtag aggregation. In the weak scaling plot it can be seen that time taken by Giraph for increasing file sizes increase and does not show weak scaling behaviour. This is because the input superstep time(the time for reading the input (vertices and/or edges as provided) into memory and assigning vertices to partitions and partitions to workers) dominates the overall time of the algorithm. Hashtag aggregation involves only 1 superstep and the time involved in the actual computation is very less as compared to input superstep time. Also the 10 timestep files are loaded and processed sequentially for each size. This input superstep time goes on increasing as we increase the size of files and as a result we get an increase in the overall time of giraph. In case of Hadoop we get a weak scaling behaviour. This is because hashtag aggregation is simply a wordcount job and as expected map reduce performs better and exhibits weak scaling.

Transitive Closure: For transitive closure Giraph exhibits weak scaling behaviour. This algorithm involves performing BFS till we find the target vertex from a given source vertex. It involves simple message passing. For Hadoop the behaviour is not a perfect weak scaling one. From the plot it is visible that the time increases. This is because transitive closure in Hadoop involves loading the output of previous iteration into a hashmap. As we increase the size of file the output also increases and the time taken to load the vertices into hashmap also increases.

Page Rank: In Page Rank, Hadoop's makespan is higher than Giraph for all sizes because hadoop repeatedly writes output of each iteration to HDFS, which is read in subsequent iterations while Giraph performs each iteration as a superstep, so everything is performed in memory. Other point that is to be noted is that hadoop doesn't scale as the data size increases. This is due to increase in HashMap, that is used to distribute page rank to all vertices to handle dangling vertex.As data size increases, total number of vertices increases and hence size of hashmap increases. This increases the sequential portion of Page Rank Algorithm across 30 iterations(HashMap is read at start of each Mapper task)

- Resource Utilization:We captured CPU utilization, Memory Utilization, Network Traffic during the makespan of the algorithm, this helped in analysis of performance of each of the algorithms. X-axis shows the time and Y-axis shows CPU usage percentage, memory usage in giga bytes, network usage in mega bytes.
  Clustering: Fig.30 and Fig.33 show that CPU utilization of Hadoop is more compared to Giraph. The reason being that computing clustering coefficient in Hadoop involves string manipulation whereas in Giraph there is much lesser operations as it takes advantage of the graph structure. Also the spikes in Fig.30 correspond to map-reduce tasks within the 3 Hadoop jobs. Fig.33 shows the spikes for the supersteps involved. Fig.32 and Fig.35 show that network usage of Hadoop is much more compared to Giraph. The initial spike in Fig.32 corresponds to the 1st job which loads all the vertices having a particular hashtag into a hashmap. And due to this we can see that network usage of 1st job is high. In case of Giraph network usage is low because the input file is read once and further supersteps involve only message passing.
  Triangle Counting: In Fig.12 and Fig.15 it is visible that the CPU utilisation of Giraph is much less compared to Hadoop. In Giraph triangle counting involves only message passing, or counting the messages whereas in Hadoop edge list taken as input and string manipulation is done at reducer for first job to find two length paths which is proportional to square of the number of values(neighbours) of current key(vertex) considered, while in reducer of second job, all values are iterated until $ is seen which indicates presence of triangle. Giraph uses message passing during its three supersteps while in Hadoop, finding two length path requires storing all the neighbours of a vertex in memory which supports the memory utilization plot in Fig.13 and Fig.16. Peak Network utilization is high for Hadoop because it emits the entire edge list in map method of first job while giraph sends relatively small number of messages during its makespan(Fig.17). The networks utilization also supports our analysis about CPU utilization being high in Hadoop(Low N/W utilization between two big spikes in Fig.14) which indicates reducer of first job spending time in computing.
  HashTag aggregation: As visible from Fig.24 and Fig.27 the CPU utilization curve in Hadoop shows a single spike while in Giraph there are multiple. This is because Hadoop does hashtag aggregation in a single pass while in Giraph it requires 10 Giraph jobs which run sequentially to process the 10 timestep files. The smaller spikes correspond to Giraph jobs while the larger ones correspond to Hadoop jobs which pass vertex states to next timestep. For the Fig.25 and Fig.28 it is observed that the peak memory usage is almost same for both Hadoop and Giraph. This is because there are Hadoop jobs which run alternatively after each Giraph job which takes the same input as the Hadoop job which does hashtag aggregation and thus the memory usage is almost same. In Fig.26 and Fig.29 it can be seen that the peak network usage is much more for Hadoop as compared to Giraph. This is because Hadoop takes all the timestep files at once as input and emits the count for each vertex whereas Giraph processes one timestep file at a time and thus the number of messages in Giraph are very less.
  Transitive Closure: From Fig.18 and Fig.21 it is observed that the CPU utilization of Hadoop is around 4 times more than that of Giraph. This is because finding transitive closure in Giraph involves only passing messages whereas in Hadoop one has to read the output of previous iteration, load the hashmap and emit the reachable vertices in each iteration. Also the 5 spikes correspond to the 5 iterations/supersteps involved. In Fig.19 memory usage plot of Hadoop shows 5 spikes which illustrates reading the input file before each iteration and loading the output of previous iteration into the hashmap. Fig.22 shows memory usage for Giraph in which the intial spike shows the input superstep. In Fig.20 and Fig.23 it is visible that the peak network usage of Giraph is less than of Hadoop. This is because Hadoop involves reading the output from hdfs each time and loading the hashmap whereas in Giraph the file is loaded at once and further supersteps involve only message passing. Also the 5 spikes in Fig.18 correspond to 5 iterations involved. Fig.21 shows that the network usage keeps on increasing in each superstep. This is because after each superstep the number of messages increase because the number of reachable vertices keep on increasing.
  Page Rank: It is visible from the Fig.6 and Fig.9 that the CPU utilisation of Giraph is much less compared to Hadoop. This is because in Giraph only messages are sent and they are summed up in the next superstep whereas in Hadoop page rank requires string manipulation i.e splitting the adjacency list and emitting each vertex with its rank to the reducer as well as the graph structure. Also we can see that there are around 30 spikes which correspond to the 30 supersteps/iterations. From Fig.7 and Fig.10 it can be seen that memory usage of Hadoop is almost double than that for Giraph. Also there are spikes for Hadoop which is due to reading and writing to hdfs after each

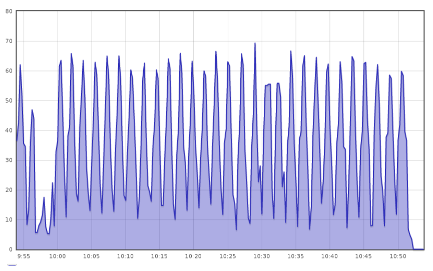

Fig. 6. Page Rank CPU Utilization Hadoop

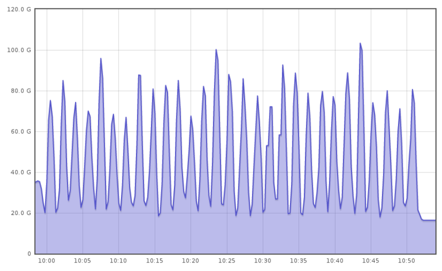

Fig. 7. Page Rank Memory Utilization Hadoop

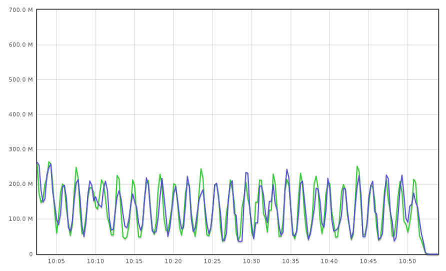

Fig. 8. Page Rank Network Utilization Hadoop

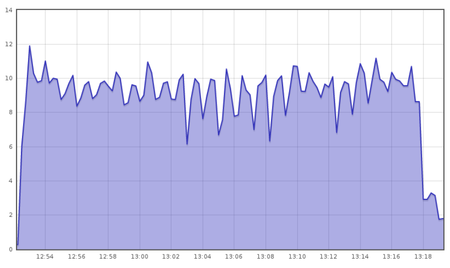

Fig. 9. Page Rank CPU Utilization Giraph

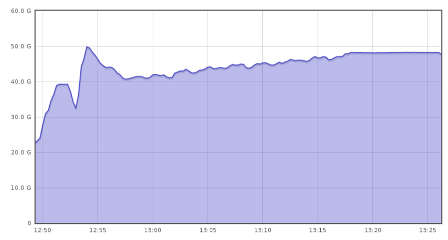

Fig. 10. Page Rank Memory Utilization Giraph

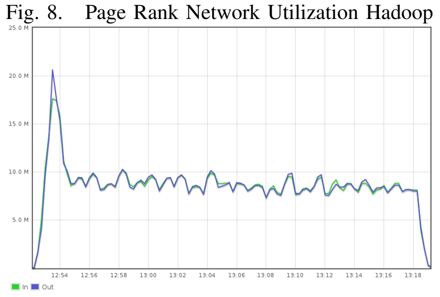

Fig. 11. Page Rank Network Utilization Giraph

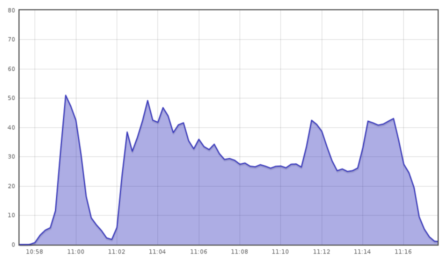

Fig. 12. Triangle Counting CPU Utilization Hadoop

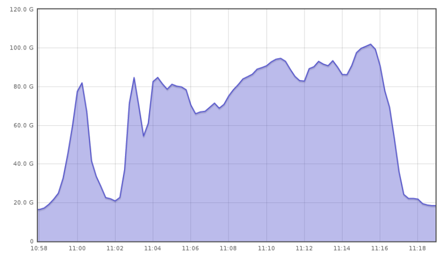

Fig. 13. Triangle Counting Memory Utilization Hadoop

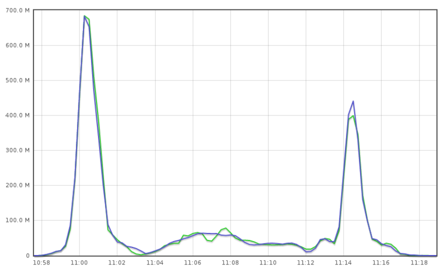

Fig. 14. Triangle Counting Network Utilization Hadoop

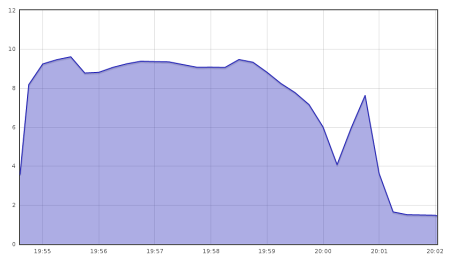

Fig. 15. Triangle Counting CPU Utilization Giraph

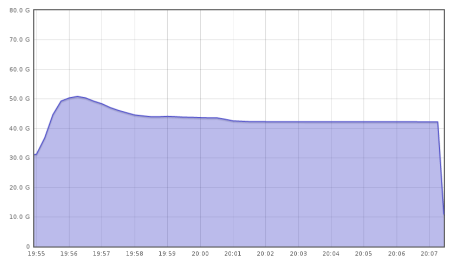

Fig. 16. Triangle Counting Memory Utilization Giraph

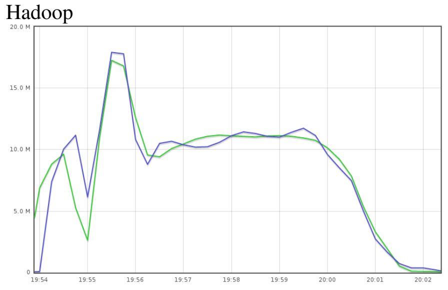

Fig. 17. Triangle Counting Network Utilization Giraph

iteration and also loading the hashmap. In case of Giraph the initial high spike corresponds to the input superstep phase and after that the memory usage is constant because the entire algorithm involves only message passing and thus memory usage is constant. From Fig.8 and Fig.11 it is visible that the network usage for Hadoop is much higher as compared to Giraph. This is because in Giraph we only send the page rank values to the neighbouring vertices while in Hadoop we not only emit the page ranks but also pass the graph structure. Thus size of the messages is increased. Also after each iteration entire file has to loaded which is not so in Giraph. The initial high spike in Giraph corresponds to loading the input file.

V. Conclusion

In this project, we analyzed common algorithms using different performance metrics such as makespan,scalability and resource utilization. It is observed peak utilization of CPU is higher in hadoop while memory usage was higher in giraph, peak network utilization is higher in map reduce.Makespan of algorithms that uses graph structure(iterative) is higher in hadoop while algorithms that require simple string processing like word count that do not get any advantage from passing graph structure as found while running HashTag Aggregation.

.

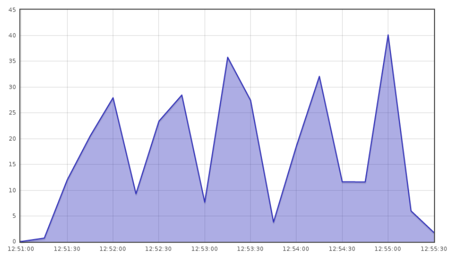
Fig. 18. Transitive Closure CPU Utilization Hadoop

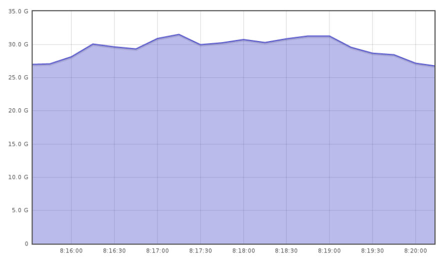
Fig. 19. Transitive Closure Memory Utilization Hadoop

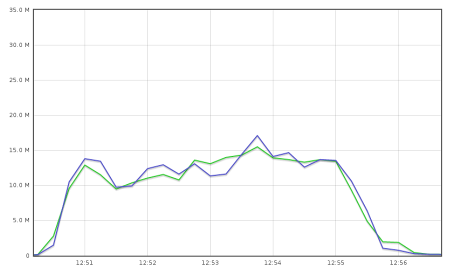
Fig. 20. Transitive Closure Network Utilization Hadoop

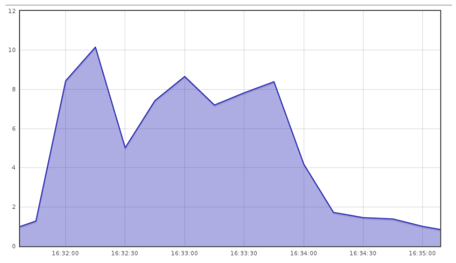
Fig. 21. Transitive Closure CPU Utilization Giraph

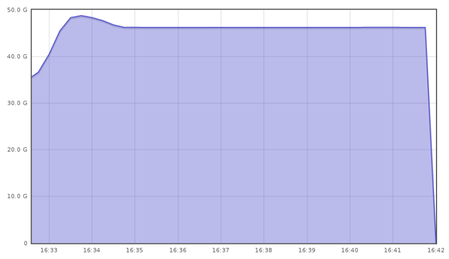
Fig. 22. Transitive Closure Memory Utilization Giraph

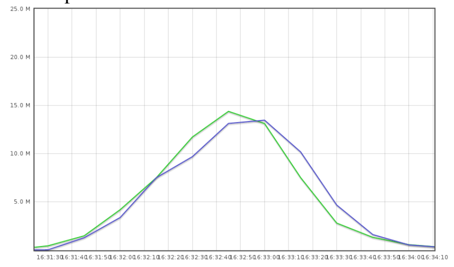
Fig. 23. Transitive Closure Network Utilization Giraph

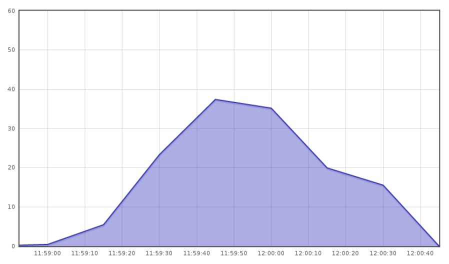
Fig. 24. HashTag CPU Utilization Hadoop

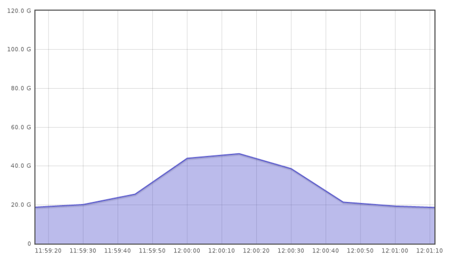
Fig. 25. HashTag Memory Utilization Hadoop

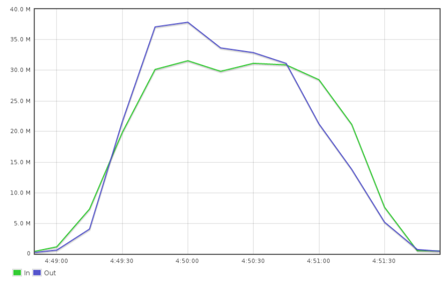

Fig. 26. HashTag Network Utilization Hadoop

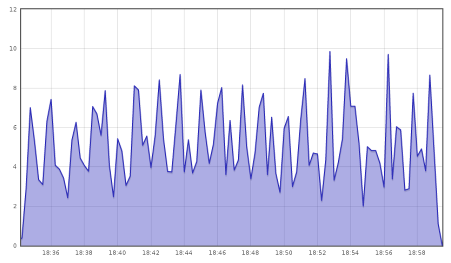
Fig. 27. HashTag CPU Utilization Giraph

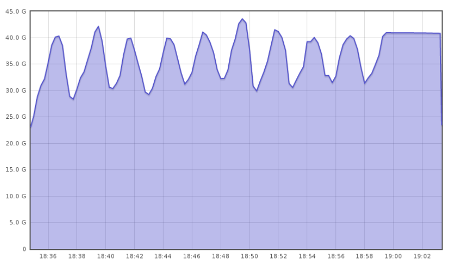
Fig. 28. HashTag Memory Utilization Giraph

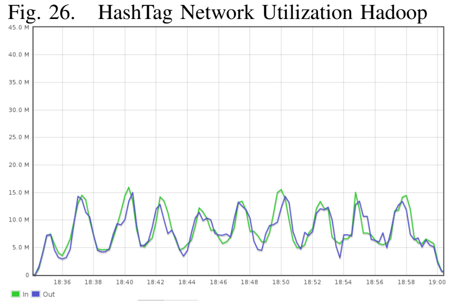

Fig. 29. HashTag Network Utilization Giraph

References

[1] Apache hadoop. http://hadoop.apache.org

[2] Apache giraph - http://giraph.apache.org

[3] MapReduce:Simplifed Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat

[4] Does Social Media Big Data Make the World Smaller? An Exploratory Analysis of Keyword-Hashtag Networks
Hamed, Ahmed Abdeen and Wu, Xindong

[5] Counting Triangles and the Curse of the Last Reducer
Siddharth Suri and Sergei Vassilvitskii

[6] Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics
Juwei Shi,Yunjie Qiu,Umar Farooq Minhas,Limei Jiao,Chen Wang,Berthold Reinwald and Fatma Ozcan

[7] Distributed Programming over Time Series Graphs
Yogesh Simmhan, Neel Choudhury

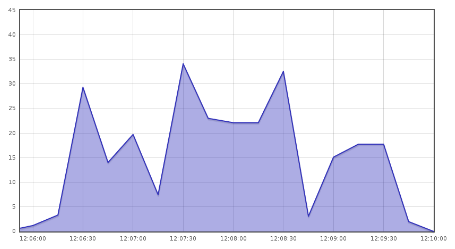
Fig. 30. Local Clustering CPU Utilization Hadoop

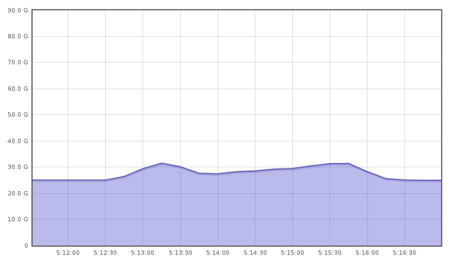
Fig. 31. Local Clustering Memory Utilization Hadoop

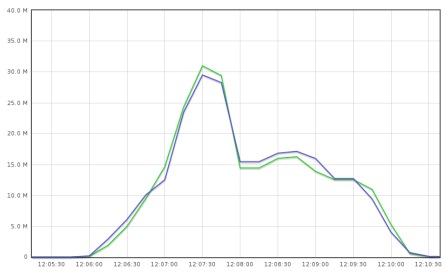
Fig. 32. Local Clustering Network Utilization Hadoop

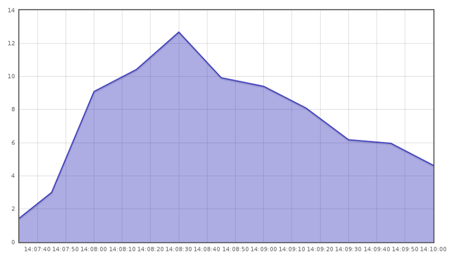
Fig. 33. Local Clustering CPU Utilization Giraph

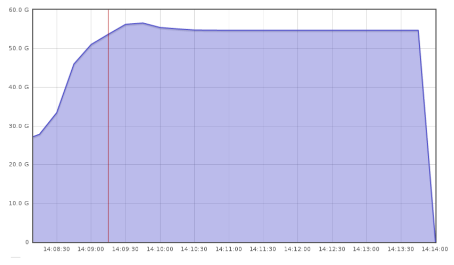
Fig. 34. Local Clustering Memory Utilization Giraph

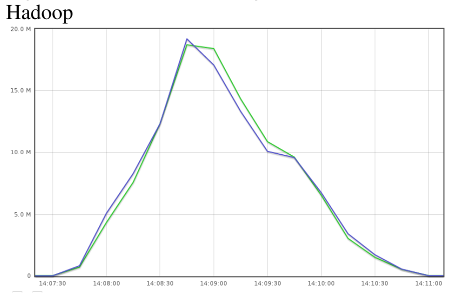
Fig. 35. Local Clustering Network Utilization Giraph