Rapport de Recherche

Handling limits of high degree vertices in graph processing using MapReduce and Pregel

M. Al Hajj Hassan
Lebanese International University, Beirut
Mostafa BAMHA
LIFO, Université d'Orléans.

Handling limits of high degree vertices in graph processing using MapReduce and Pregel

M. Al Hajj Hassan[1] and Mostafa Bamha[2]

[1] Lebanese International University, Beirut, Lebanon
mohamad.hajjhassan01@liu.edu.lb
[2] Université Orléans, INSA Centre Val de Loire, LIFO EA 4022, France
Mostafa.Bamha@univ-orleans.fr

Abstract

Even if Pregel scales better than MapReduce in graph processing by reducing iteration's disk I/O, while offering an easy programming model using "think like vertex" approach, large scale graph processing is still challenging in the presence of high degree vertices: Communication and load imbalance among processing nodes can have disastrous effects on performance. In this paper, we introduce a scalable MapReduce graph partitioning approach for high degree vertices using a master/slave partitioning allowing to balance communication and computation among processing nodes during all the stages of graph processing. Cost analysis and performance tests of this partitioning are given to show the effectiveness and the scalability of this approach in large scale systems.

Keywords: Graph processing, High degree vertices, Data skew, MapReduce programming model, Pregel, Distributed file systems.

1 Introduction

Distributed processing of algorithms on large scale graphs is increasingly used in applications such as social, biological, communication and road networks. Most of these graph analysis algorithms are iterative. For this reason, general-purpose distributed data processing frameworks such as Hadoop [8] are not efficient for processing such algorithms since data should be read from and written into disks between iterations.
Several specialised graph processing frameworks such as Google's Pregel [12], Apache Giraph [6] and GraphLab [1] are proposed to speed up the execution of iterative graph algorithms. Most of these frameworks follow *think like a vertex* vertex centric programming model. In frameworks such as Pregel and Giraph, based on Bulk Synchronous Parallel (BSP) model [15], each vertex receives messages from its incoming neighbours, updates its state then sends messages to outgoing neighbours in each iteration. On the other side, GraphLab is based on GAS (Gather, Apply, Scatter) model and shared memory abstraction. In this model, each active node can directly access and collect data from its neighbours, in the Gather phase, without the need for messages. Each active vertex then accumulates the collected values to update its state in the Apply phase then updates and activates its adjacent vertices in the scatter phase .
In these frameworks, graph vertices are partitioned into sub-graphs which are distributed over the computing nodes. In order to benefit from the processing capacity of parallel and distributed machines, the partitions should be assigned to computing nodes in a way that balances their work load and reduces the communication costs between different nodes. Due to power-law degree distribution in real networks, where few number of vertices are connected to large fraction of the graph [3, 11] such as celebrities on Facebook and Twitter, graph partitioning is considered as NP-complete problem [4]. To this end, we propose a MapReduce based graph

partitioning algorithm that allows us to evenly assign the load of all computing nodes. The algorithm proceeds in two MapReduce jobs. In the first one, the graph is read from HDFS to determine high degree vertices. In the second job, we create for each high degree vertex a master vertex in addition to incoming and outgoing slave vertices. Slave vertices are evenly assigned to workers in order to balance their loads. Graph analysis algorithms can be applied on the partitioned graph using any framework that supports reading data from HDFS. We tested the performance of our approach by executing *Single Source Shortest Paths* algorithm on partitioned and un-partitioned graphs with highly skewed nodes under Hadoop and Giraph frameworks. The test results proved the efficiency of our approach.

The remaining of the paper is organised as follows. In section 2, we review MapReduce and Pregel programming models. Our big graph partitioning and processing approach is presented in section 3 with its complexity analysis. Experiments results presented in section 4 confirm the efficiency of our approach. Related works are reviewed in section 5 and we conclude in section 6.

2 MapReduce vs Pregel programming model

Google's MapReduce programming model presented in [5] is based on two functions: *Map* and *Reduce*. Dean and Ghemawat stated that they have inspired their MapReduce model from Lisp and other functional languages [5]. The programmer is only required to implement two functions *Map* and *Reduce* having the following signatures:

map: $(k_1, v_1) \longrightarrow list(k_2, v_2)$,
reduce: $(k_2, list(v_2)) \longrightarrow list(v_3)$.

The user must write the *map* function that has two input parameters, a key k_1 and an associated value v_1. Its output is a list of intermediate key/value pairs (k_2, v_2). This list is partitioned by the MapReduce framework depending on the values of k_2, where all pairs having the same value of k_2 belong to the same group.

The *reduce* function, that must also be written by the user, has two parameters as input: an intermediate key k_2 and a list of intermediate values $list(v_2)$ associated with k_2. It applies the user defined merge logic on $list(v_2)$ and outputs a list of values $list(v_3)$.

MapReduce is a simple yet powerful framework for implementing distributed applications without having extensive prior knowledge of issues related to data redistribution, or task allocation and fault tolerance in large scale distributed systems. Most MapReduce frameworks include *Distributed File Systems* (DFS) designed to store very large files with streaming data access patterns and data replication for fault tolerance while guaranteeing high disk I/O throughput.

To cover a large of application's need in term of computation and data redistribution, in most MapReduce frameworks, the user can optionally implement two additional functions : init() and close() called before and after each map or reduce task. The user can also specify a "partition function" to send each key k_2 generated in map phase to a specific reducer destination. The reducer destination may be computed using only a part of the input key k_2 (Hadoop's default "partition function" is generally based on "hashing" the whole input key k_2) [9, 10]. The signature of the partition function is :

partition: $(Key \quad k_2) \longrightarrow Integer$. `/* Integer is between 0 and the number of reducers #numReduceTasks */`
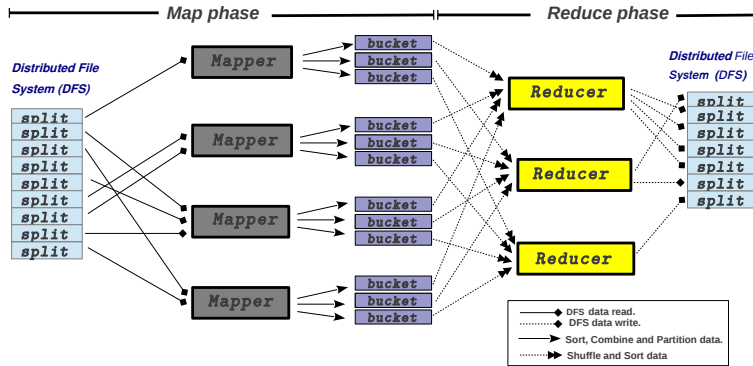
Figure 1: MapReduce framework.

For efficiency reasons, in Hadoop MapReduce framework, the user may also specify a "Combine function", to reduce the amount of data transmitted from Mappers to Reducers during *shuffle* phase (see fig 1). The "Combine function" is like a local reduce applied (at map worker) before storing or sending intermediate results to the reducers. The signature of *Combine* function is:

combine: $\quad (k_2, list(v_2)) \longrightarrow (k_2, list(v_3))$.

MapReduce excels in the treatment of data parallel applications where computation can be decomposed into many independent tasks involving large input data. However MapReduce's performance degrades in the case of dependent tasks or iterative data processing such as graph computations due to the fact that, for each computation step, input data must be read from DFS for each map phase and intermediate output data must be written back to DFS at the end of reduce phase for each iteration: This may induce high communication and disk I/O costs. To this end, *Pregel* model (a variant of MapReduce) [12] was introduced for large scale graph processing. Pregel is based on *Bulk Synchronous Parallel* (BSP) programming model [14] where each parallel program is executed as a sequence of parallel *Supersteps*, each *superstep* is divided into (at most) three successive and logically disjoint phases. In the first phase each processing node uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronisation barrier occurs, making the transferred data available for the next superstep.

To minimise execution time of a BSP program, design must jointly minimise the number of supersteps and the total volume of communication while avoiding load imbalance among processing nodes during all the stages of computation.

Similarly to BSP, Pregel programs proceed into three logical phases:

1. A Setup phase orchestred by a master node where Workers read input graph data from DFS, this data is partitioned among *Workers* using a partitioning function on vertices, this partitioning is based on hashing functions and each partition is assigned to single worker,
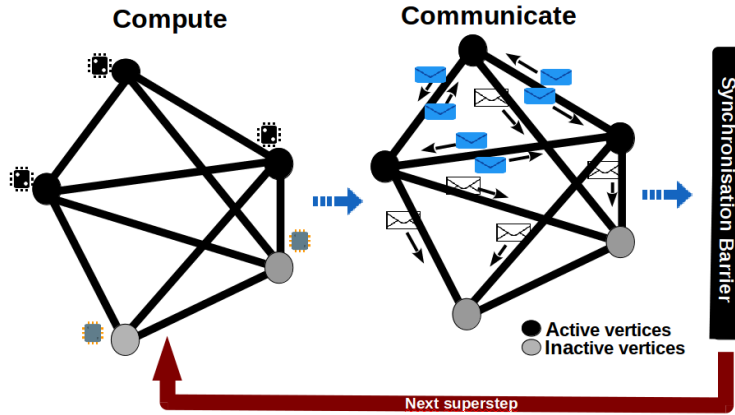
4

Figure 2: Supersteps in Pregel computation model for a graph processing

2. A compute phase where each processing node performs a sequence of Supersteps, each one consists of a local computation on worker's data : Active vertices can modify their state or that of its outgoing edges, handle received messages sent to them in the previous superstep, This step is followed by a communication step where processing nodes can send asynchronously messages or vertices/edges modification requests to other processing nodes (these messages are received in the next superstep to perform vertex data/state modification, Vertex/Edges modifications, ...), and then a global synchronisation barrier is performed to make transferred data and graph topology modifications available for the next superstep,

3. A close phase where workers store output result to DFS.

Figure 2 shows an example of a *superstep* computation in *Pregel* where only a set of "active" vertices performs a local computation. Each idle vertex is activated whenever it receives one or more messages from other vertices. The program terminates (halts) when all the vertices become inactive.

Even if, Pregel scales better than MapReduce for graph processing, it still remains inefficient in the presence of high degree vertices since for each processing iteration, high degree vertices may communicate with all their neighbours which can induce load imbalance among processing nodes. This can also lead to a memory lack whenever these messages or the list of neighbours can not fit in processing node's memory which limits the scalability of the model. To this end, we introduce, in this paper, a partitioning approach for high degree vertices based master/slave repartition allowing to avoid the load imbalance among processing nodes while guaranteeing that the amount of data communicated at each computation step never exceed a user defined value.

In this partitioning, a high degree vertex H is transformed into a master vertex called $H - 0$ connected to set of "Left" and "Right" slaves (called $H - L_i$ and $H - R_j$ respectively) depending on the number of the incoming and outgoing edges to vertex H. Figure 3 shows an example where a high degree vertex is partitioned into m "left" slaves and n "right" slaves.
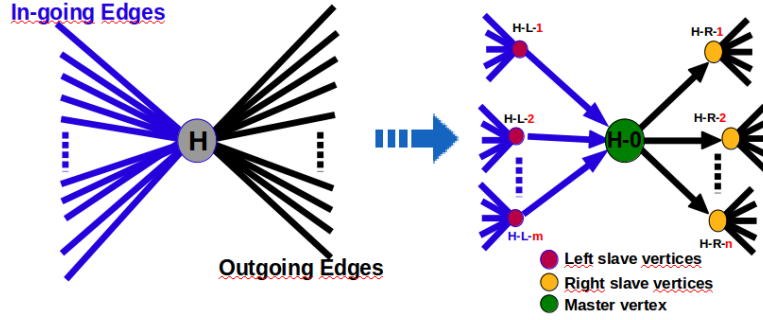
Figure 3: High degree vertex partitioning approach

In this paper, we used an open source version of MapReduce called Hadoop developed by "The Apache Software Foundation". Hadoop framework includes a distributed file system called HDFS[1] designed to store very large files with streaming data access patterns.

3 GPHD : A solution for big graph partitioning/processing in the presence of high degree vertices

To avoid the effect of high degree vertices in graph processing, we introduce, in this section, a new approach, called *GPHD* (Graph Partitioning for High Degree vertices) to evenly partition each high degree vertex into a master and a set of left and right slave vertices depending on the number of incoming and outgoing edges to these high vertices. This partitioning allows to balance communication and computation during all stages of graph processing. Vertex partitioning, in this approach, can be generalised to many graph processing problems ranging from *SSSP*, *PageRank* to *Connected Components* problems.

We will describe, in detail, computation steps in *GPHD* while giving an upper bound of execution cost for each step. GPHD proceeds in two MapReduce jobs :

- **a.** the first job is used to identify high degree vertices and to generate high degree vertices partitioning templates,
- **b.** the second job is used to partition *input graph* data using generated partitioning templates. In this partitioning, only high degree vertices are partitioned into masters and slave vertices. These slave vertices are affected to different workers in a *round-robin* manner to balance load among processing nodes.

We consider a weighted n-vertex, m-edge graph $G(V, E)$ where V is the set of vertices of G and E the set of edges in G. For scalability, we assume that graph $G(V, E)$ is stored as a set of edges E divided into blocks (splits) of data. These splits are stored in Hadoop Distributed File System (HDFS). These splits are also replicated on several nodes for reliability issues, and throughout this paper, we use the following notations:

- $|V|$: number of pages (or blocks of data) forming the set of vertices V,
- $\|V\|$: number n of vertices in V,
- $\overline{V}$: the restriction (a fragment) of set V which contains only high degree vertices. $\|\overline{V}\|$ is, in general, very small compared to the number of vertices $\|V\|$,

---

[1]HDFS: Hadoop Distributed File System.

- V_i^{map}: the split(s) of set V affected to mapper (Worker) i, V_i^{map} contains all (source and destination) vertices of E_i^{map},
- V_i^{red}: the split(s) of set V affected to reducer (Worker) i. V_i^{red} is the subset of V obtained by a simple hashing of vertices in V,
- $\overline{V}_i$: the split(s) of set $\overline{V}$ affected to mapper i holding only high degree vertices from V,
- $|E|$: number of pages (or blocks of data) forming the set of edges E,
- $\|E\|$: number m of edges in E,
- E_i^{map}: the split(s) of set E affected to mapper (Worker) i,
- E_i^{red}: the split(s) of set E affected to reducer (Worker) i,
- $c_{r/w}$: read/write cost of a page of data from/to distributed file system (DFS),
- c_{comm}: communication cost per page of data,
- t_s^i: time to perform a simple search in a Hashtable on node i,
- t_h^i: time to add an entry to a Hashtable on node i,
- NB_mappers: number of job's mapper nodes,
- NB_reducers: number of job's reducer nodes.

GPHD proceeds into 3 phases (two phases are carried out in the first MapReduce Job and a single phase in the second Job):

a.1: Map phase to generate mapper's "local" incoming and outgoing degrees for each vertex in mapper's input graph

Each mapper i reads its assigned data splits (blocks) of subset E_i^{map} from the DFS, and gets `source vertex "s"` and `destination vertex "d"` from each input edge $e(\text{source\_vertex}:s, \text{destination\_vertex}:d, \text{value}:a)$ and emits two tagged records for "s" and "d" vertices with frequency "One".

- Emit a couple (<A,s>,1) : This means that, we found "One" outgoing edge starting from `source vertex "s"`,
- Emit a couple (<B,d>,1): This means that, we found "One" incoming edge arriving to `destination vertex "d"`,

Tag 'A' is used to identify outgoing edges from a source vertex whereas tag 'B' is used to identify incoming edges to a destination vertex. The cost of this step is:

$$Time(a.1.1) = O\big(\max_{i=1}^{NB\_mappers} c_{r/w} * |E_i^{map}| + 2 * \max_{i=1}^{NB\_mappers} \|E_i^{map}\|\big).$$

The term $\max_{i=1}^{NB\_mappers} c_{r/w} * |E_i^{map}|$ is time to read input graph from HDFS by mappers, whereas the term $2*\max_{i=1}^{NB\_mappers} \|E_i^{map}\|$ is time to scan mapper's edges and to emit two tagged records for each edge.

Emitted couples (<Tag,v>,1) are then combined to generate local frequencies (incoming and outgoing degrees) for each source and destination vertex in E_i^{map}. These combined records are, then, partitioned using a user "defined partitioning function" by hashing only key part v and not the whole mapper tagged key <Tag,v>. The result of combine phase is then sent to reducers of destination in the shuffle phase of the following reduce step. This step is performed to compute the number of local incoming and outgoing edges to each vertex. The cost of this step is at most:

$$Time(a.1.2) = O\left(\max_{i=1}^{NB\_mappers} c_{comm} * |V_i^{map}| + \max_{i=1}^{NB\_mappers} \|V_i^{map}\| * \log\left(\|V_i^{map}\|\right)\right).$$

The term $c_{comm} * |V_i^{map}|$ is time to communicate data from mappers to reducers, whereas the term $\|V_i^{map}\| * \log\left(\|V_i^{map}\|\right)$ is time to sort mapper's emitted records. And the global cost of this step is therefore : $Time_{step_{a.1}} = Time(a.1.1) + Time(a.1.2)$.

Algorithm 1 GPHD algorithm's workflow

---

a.1▶ Map phase: /* To generate "Local incoming and outgoing" degrees for each vertex in input
               graph */

▷ Each mapper i reads its assigned data splits (blocks) of subset E_i^{map} from the DFS

   ▷ Get `source vertex` "s" and `destination vertex` "d" from each input edge
     e(source_vertex : s, destination_vertex : d, value : a) from E_i^{map}.
    ▷ Emit a couple (<A,s>,1) /* 'A' tag identifies one outgoing edge from `source vertex` "s" */
    ▷ Emit a couple (<B,d>,1) /* 'B' tag identifies one incoming edge to `destination vertex` "d" */

▶ Combine phase: /* To compute local frequencies (incoming and outgoing degrees) for each source
and destination vertex in set E_i^{map} */

▷ Each combiner, for each source vertex "s" (resp. destination vertex "d") computes local outgoing
(resp. local incoming) degree : the sum of generated local frequencies associated to a source
(resp. destination) vertex generated in Map phase.

▶ Partition phase:

▷ For each emitted couple (key, value)=$(< \text{Tag}, "v" >, \text{frequency})$ where Tag is "A" (resp. "B")
for source (resp. destination) vertex "v", compute reducer destination according to only vertexID "v".

a.2▶ Reduce phase: /* To combine Shuffle's records and to create Global histogram for
                         "high degree vertices partitioning templates" */

▷ Compute the global frequencies (incoming and outgoing degrees) for each vertex present in set V.

▷ Emit, for each **high degree vertex** "v", a couple ("v", Nb_leftSlaveVertices, Nb_rightSlaveVertices):
`Nb_leftSlaveVertices` and `Nb_rightSlaveVertices` are fixed depending on the values of the incoming
and outgoing degrees of vertex "v",

b.1▶ Map phase:

▷ Each mapper reads Global histogram of "high degree vertices partitioning templates" from DFS,
and creates a local Hashtable.

▷ Each mapper, i, reads its assigned edge splits of input graph from DFS and generates a set of left
and right slave vertices for each edge depending on the degrees of source and destination vertices:
only edges associated to a high degree for a source or a destination vertex are transformed whereas
those associated to low degree for both source and destination vertices are emitted as they are without
any graph transformation. Note that, left and right slave vertices associated to a high degree vertex
are created only once by a designated worker and new edges are created from each master vertex to
its left and right slave vertices.

▷ Emit all edges resulting from this graph transformation, and mark high degree vertices for a later
removal.

---

a.2: Reduce phase to combine Shuffle's records and to create Global histogram for "high degree vertices partitioning templates"

At the end of the shuffle phase, each reducer i will receive a subset, called V_i^{red}, of vertices
(and their corresponding incoming and outgoing local degrees) obtained through hashing
of distinct values of V_j^{map} held by each mapper j. In this step, received incoming and
outgoing local frequencies are then merged to compute the global frequencies (the global
incoming and outgoing degrees) for each vertex present in set V. To this end, each reducer i, emits, for each **high degree vertex** "v", a couple ("v", `Nb_leftSlaveVertices`,
`Nb_rightSlaveVertices`) where :

- `Nb_leftSlaveVertices`: is the number of "left" slaves to create in the following
phase b.1; these "left" slaves are used to partition incoming edges to a high degree
vertex "v". `Nb_leftSlaveVertices` depends only on the number of incoming edges
to vertex "v",

– `Nb_rightSlaveVertices`: is the number of "right" slaves (these slaves are created in phase b.1) used to partition outgoing edges of a high degree vertex "v". `Nb_rightSlaveVertices` depends only on the number of outgoing edges of "v".

Using this information, each reducer i, has local knowledge of how high degree vertices will be partitioned in the next map phase.

The global cost of this step is at most: $Time_{step_{a.2}} = O\left(\max_{i=1}^{NB\_reducers} \|V_i^{red}\|\right)$.

To avoid the effect of high degree vertices in graph processing, generated "left" and "right" slave vertices are affected to distinct "workers" in a round-robin manner to balance load during all the stages of graph's computation.

To guarantee a perfect balancing of the load among processing nodes, partitioning templates and graph partitioning are carried out jointly by all reducers (and not by a coordinator node). Note that, only edges associated to high degree vertices are split and sent to distinct workers whereas edges associated to low degrees for both source and destination vertices are emitted without any transformation.

b.1: Map phase to generate partitioned graph

Each mapper reads Global histogram of "high degree vertices partitioning templates" from DFS, and creates a local Hashtable.

In this step, each mapper i, reads its assigned edge splits of input graph from DFS and generates a set of left and right slave vertices for each edge depending on the degrees of source and destination vertices: only edges associated to a high degree source or destination vertex are transformed whereas those associated to low degree for both source and destination vertices are emitted as they are without any graph transformation. Note that, left and right slave vertices associated to a high degree vertex are created only once by a designated worker and new edges are created from each master vertex to its left and right slave vertices.

At the end of this step, mappers emit all edges resulting from this graph transformation, and mark high degree vertices for a later removal. The cost of this step is at most:

$$Time(b.1) = O\left(\max_{i=1}^{NB\_mappers} c_{r/w} * |E_i^{map}| + t_h^i * \|\overline{V}\| + 2 * t_s^i * \|E_i^{map}\|\right).$$

The term $c_{r/w}|E_i^{map}|$ is time to read edges of input graph from DFS on each mapper i, the term $t_h^i * \|\overline{V}\|$ is time to build the Hashtable holding high degree vertices and their corresponding values `Nb_leftSlaveVertices` and `Nb_rightSlaveVertices` whereas the term $2 * t_s^i * \|E_i^{map}\|$ is time to perform a Hashtable search for both source and destination vertices of mapper's input edges. We recall that, the size of this Hashtable, is in general, very small compared to the size of input graph.

The global cost, $Time_{GPHD}$, of `GPHD` algorithm is therefore the sum of above three phases. GPHD has asymptotic optimal complexity when:

$$\|\overline{V}\| \leq \max\left(\max_{i=1}^{NB\_mappers} \|\|E_i^{map}\| * \log(\|E_i^{map}\|), \max_{i=1}^{NB\_reducers} \|E_i^{red}\|\right), \tag{1}$$

this is due to the fact that, all other terms in $Time_{GPHD}$ are, at most, of the same order of $\|E_i^{map}\|$. Inequality 1 holds, in general, since $\overline{V}$ contains only high degree vertices and the number of these high degree vertices is very small compared to the number of input graph vertices.

4 Experiments

To evaluate the performance of our high degree vertex partitioning approach we compared the execution of the Single Source Shortest Paths (SSSP) problem on both partitioned and unpartitioned graph data using Hadoop-1.2.1 and Giraph-1.2.0 frameworks[2]. We ran a large series of experiments where 49 `Virtual Machines` (VMs) were randomly selected from our university cluster using OpenNebula software for VMs administration. Each Virtual Machine has the following characteristics : 1 Intel(R) Xeon@2.53GHz CPU, 2 Cores, 6GB of Memory and 80GB of Disk. Setting up a Hadoop cluster consisted of deploying each centralised entity (namenode and jobtracker) on a dedicated `Virtual Machine` and co-deploying datanodes and tasktrackers on the rest of VMs. The data replication parameter was fixed to three in the HDFS configuration file.

To study the effect of data skew on SSSP performance, we use a synthetic graphs following a power low distribution. To this end, generated graphs have been chosen to follow a Zipf distribution [16] as it is the case in most database tests : Zipf factor has been varied from 0 (for a uniform data distribution) to 2.8 (for a highly skewed data). Note that natural graphs follow a power low of ~ 2 which corresponds to a highly skewed data [7,13]. In our experiments, each input graph had a fixed size : 200M vertices and 1B edges (corresponding to about $\sim$25GB for each graph data).

We noticed, in all the tests and also those presented in Figure 4, that using GPHD partitioning graph algorithm, Giraph's execution for SSSP problem is insensitive to data skew whereas the same execution using non partitioned graphs fails due to lack of memory for skew factors varying from 1.2 to 2.8. Moreover, in Figure 4, we can see that, GPHD processing time (which includes both "High degree vertices processing" and "Graph partitioning") remains very small compared to Giraph SSSP execution time and the overhead related to GPHD preprocessing remains very small compared to the gain in performance related to the use of partitioned graphs due to the fact that, in partitioned graphs, computation and communication are much more balanced with respect to Giraph execution using non partitioned graphs. This shows that, GPHD preprocessing makes `Giraph` more scalable and insensitive to high degree vertices in large scale graph processing.

5 Related Work

Apache Giraph: [6] A distributed framework for processing iterative algorithms on large-scale graph. Giraph is an open implementation of Pregel [12] inspired by the Bulk Synchronous Parallel (BSP) model [15] and based on a vertex-centric programming model. In Giraph, vertices are divided into partitions assigned to a set of workers. The default partitioning method is based on a hash function (or range) applied on vertices IDs. In addition, a partitioning method implemented by the developer can be used. Vertices can communicate by sending messages and periodic checkpoints are executed for fault-tolerance. Giraph has extended Pregel API by adding master compute function, out-of-core capabilities that allows to treat main memory limitation by spilling graph partitions and messages to local disks, shared aggregators, edge-oriented input, and so on.

GraphLab: [1] An asynchronous parallel framework based on a distributed shared-memory architecture. In GraphLab, each vertex-program can directly access and accumulate data from

---

[2]Hadoop and Giraph are respectively implementations of MapReduce and Pregel by developed by "The Apache Software Foundation".
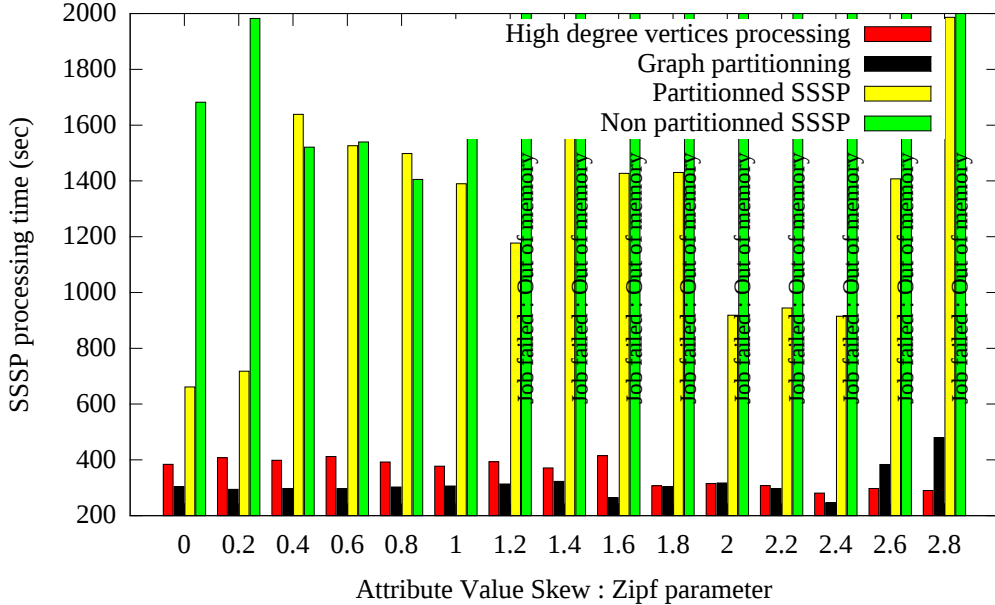
Figure 4: Graph skew effects on SSSP processing time

its adjacent edges and vertices to update its state. The updated vertex or edge data are automatically visible to the adjacent edges.

PowerGraph: [7] A distributed framework that supports both Pregel's bulk-synchronous and GraphLab's asynchronous models of computation. PowerGraph relies on Gather, Apply and Scatter (GAS) programming model to implement the vertex-program. In Giraph and GraphLab, the graph is partitioned, using edge-cut approach based on hash partitioning, in order to balance the load of different processors. However, hash partitioning is not efficient in the case of natural graph due to the skewed power-law distribution where only few vertices are connected to high number of vertices (for example celebrities on social media) while most vertices have few neighbours. To address this problem, PowerGraph follows vertex-cut approach, where high degree vertices are split over several machines thus balancing the load of machines and reducing communication costs.

GPS: [2] An open-source distributed big graph processing framework. GPS is similar with three new features. It extends Pregel's API with a new *master.compute()* function that allows to develop global computation in an easier way. In addition, GPS follows a dynamic graph repartition approach that allows to reassign vertices during job execution to other machines in order to reduce communication costs. It also has an optimization technique called large adjacency list partitioning (LALP) that allows to partition neighbours of high-degree vertices over the machines.

6 Conclusion and future work

In this paper, we have introduced an efficient and scalable MapReduce graph processing partitioning approach in the presence of high degree vertices where data associated to each high

degree vertex is partitioned among many workers in a round-robin manner. This partitioning is proved to solve, efficiently, the problem of load imbalance among "workers" where existing approaches fail to handle high degree vertices communication/processing imbalances and also the limitations of existing approaches to handle large graph datasets whenever data associated to high degree vertices cannot fit in available worker's local memory. We recall that, partitioning in `GPHD` is performed by all the mappers and not by a coordinator processor which guarantees the scalability of this algorithm. This makes also `Giraph` insensitive to the problem of high degree vertices while guaranteeing perfect balancing properties during all the stages of graph processing.

References

[1] Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud.

[2] GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM.

[3] Multilevel Algorithms for Partitioning Power-law Graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06.

[4] Some Simplified NP-complete Problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI ' 04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA*, 2004.

[6] `http://giraph.apache.org/`.

[7] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 17–30, Berkeley, CA, USA, 2012.

[8] Apache hadoop. http://hadoop.apache.org/core/.

[9] M. Al Hajj Hassan and M. Bamha. Towards scalability and data skew handling in groupby-joins using mapreduce model. In *Proceedings of the International Conference on Computational Science, ICCS 2015, Reykjavík, Iceland, 1-3 June, 2015, 2014*, pages 70–79, 2015.

[10] M. Al Hajj Hassan, M. Bamha, and Frédéric Loulergue. Handling data-skew effects in join operations using mapreduce. In *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*, pages 145–158, 2014.

[11] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters.

[12] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2010.

[13] M E J Newman. Power Laws, Pareto Distributions and Zipf's Law. 2004.

[14] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[15] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[16] G. K. Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology.* Adisson-Wesley, 1949.