Cost model for Pregel on GraphX

Reference:
Kumar Rohit, Abelló Alberto, Calders Toon.- Cost model for Pregel on GraphX
Lecture notes in computer science - ISBN 978-3-319-66917-5 - Cham, Springer, 10509(2018), p. 153-166
Full text (Publisher's DOI): https://doi.org/10.1007/978-3-319-66917-5_11

Cost Model for Pregel on GraphX

Rohit Kumar[1,2], Alberto Abelló[2], and Toon Calders[1,3]

[1]Department of Computer and Decision Engineering
Université Libre de Bruxelles, Belgium

[2]Department of Service and Information System Engineering
Universitat Politécnica de Catalunya (BarcelonaTech), Spain

[3]Department of Mathematics and Computer Science
Universiteit Antwerpen, Belgium

Abstract. The graph partitioning strategy plays a vital role in the overall execution of an algorithm in a distributed graph processing system. Choosing the best strategy is very challenging, as no one strategy is always the best fit for all kinds of graphs or algorithms. In this paper, we help users choosing a suitable partitioning strategy for algorithms based on the Pregel model by providing a cost model for the Pregel implementation in Spark-GraphX. The cost model shows the relationship between four major parameters: 1) input graph 2) cluster configuration 3) algorithm properties and 4) partitioning strategy. We validate the accuracy of the cost model on 17 different combinations of input graph, algorithm, and partition strategy. As such, the cost model can serve as a basis for yet to be developed optimizers for Pregel.

1 Introduction

Large graphs with millions of nodes and billions of edges are becoming quite common now. Social media graphs, road network graphs, and relationship graphs between buyers and products are some of the examples of large graphs generated and processed regularly [3]. With the increase in size of these graphs, the classical approach of graph processing is becoming insufficient [8, 7]. Hence, to address these shortcomings, *vertex-centric programming models* [10] have been proposed to transform the way graph problems are managed. Pregel [11] is one such programming models which supports distributed (parallel) graph computations. Many distributed graph computing (DGC) systems like PowerGraph [4] and Spark-GraphX [15] provide implementations of the Pregel model for graph computations. DGC systems distribute the graph computation by partitioning the graph over different nodes of a cluster.

There are many partitioning strategies proposed in literature [14, 12, 4] for performing efficient graph computations on DGC systems. Most of the DGC systems provide the same programing model and offer similar features and strategies to use. Depending on the internal implementation of these strategies and algorithms, the systems can give different performance. Even once a user has decided

a system to use, there are not enough guidelines on which partitioning strategy to use for which application or graph. Verma et.al. in [13] attempts to address this question with an experimental comparison of different partitioning strategies on three different DGC systems resulting in a set of rules. However, there is no clear theoretical justification of why one partitioning strategy performs better than another depending on a particular combination of graph and algorithm. Moreover, the paper does not consider the cluster properties which according to our cost model, is one of the parameters in deciding the best partitioning strategy. In this paper, we address this question by providing a cost model for the Pregel implementation in GraphX. Cost models are used in the database community for query plan evaluation. We contend that DGC systems should be able to choose the best partitioning strategy for a given graph and algorithm using our cost model in iterative graph computations.

Concretely, in this paper, we make the following contributions: *(i)* we formulate a cost model to capture the different dominating factors involved in the Pregel model (Section 3); *(ii)* we validate our cost model on GraphX by estimating the computation time and comparing it with real execution time (Section 4). To the best of our knowledge this is the first work in which a cost model based approach has been proposed for Pregel to help users to choose the best partitioning strategy. Similar cost models could be obtained for Pregel on other DGC systems.

2 Background

In this section, we present background information on (1) the Pregel model, and (2) the different partitioning strategies we used in the experiments.

2.1 Pregel Model

In order to render graph computations more efficient, new graph programming models such as Pregel have been introduced [11]. In Pregel, graph algorithms are expressed as iterative vertex-centric computations which can be easily and transparently distributed automatically. We illustrate this principle with the following graph algorithm CC for computing connected components in a graph: we start with assigning to each vertex a unique identifier. In the first step each vertex sends a message with its unique identifier to all its neighbors. Subsequently, for each vertex the minimum is computed of all incoming identifiers. If this minimum is lower than its own identifier, the vertex updates its internal state with this new minimum and sends a message to its neighbors to notify them of its new minimum. This process continues until no more messages are sent. It is easy to see that this iteration will terminate and that the result will be that each vertex holds the minimal identifier over all vertices in its connected component, which can then serve as an identifier of that connected component.

As we can see in this example, a user of Pregel only has to provide the following components:

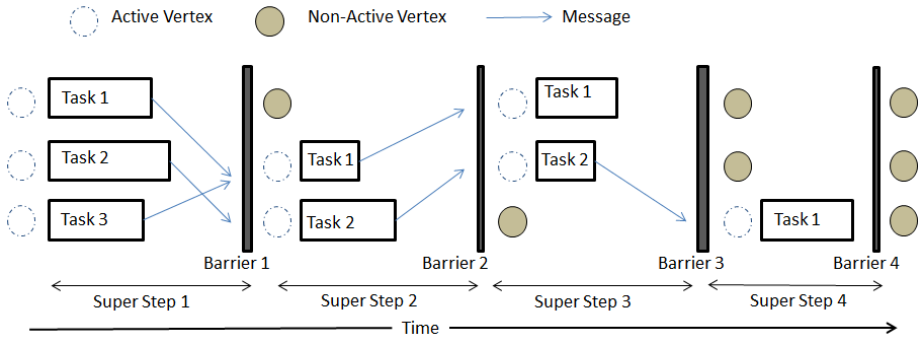


Fig. 1. An example of Pregel model consisting of three vertices.

- Initialization: one initial message per vertex. In the case of CC, this initial message contains the unique identifier of that vertex;
- Function to combine all incoming messages for a vertex. In our example, the combine function takes the minimum over all incoming identifiers.
- A function called the vertex program to update the internal state of the vertex if the minimum identifier received is less than the current identifier of the vertex.
- A function to send the vertex current identifier to its neighbors. In CC, the internal state of a vertex is updated only if the vertex receives a identifier smaller than it is already storing. Only in that case messages are sent to its neighbors with this updated minimum.

Figure 1 illustrate this programming model; every iteration of running the vertex program and combining the messages that will be input for the next iteration is called a super-step. In the first super-step every vertex is activated and executes its vertex program. In Figure 1, the vertex programs are called "tasks" and the blue lines represent messages sent between vertices. In the second super-step in this figure, vertex 1 does not receive any message and hence will not be active in super-step 2. Vertex 2 receives two messages which are combined and the vertex program is executed. Similarly, vertex 3 receives one message and executes its vertex program. The time it takes for each task could be different and hence there is a synchronization barrier after every super-step. Finally, in super-step 4 no messages are generated and the computation stops.

The main benefit of the Pregel programming model is that it provides a powerful language in which many graph algorithms can be expressed in a natural way. At the same time, however, the programs are flexible enough to allow for automatically and transparently distributing their execution as we will see in next section.

2.2 Partitioning

There are two kinds of partitioning strategies for distributed graph processing: 1) vertex-cut [4] and 2) edge-cut [6, 1]. In vertex-cut partitioning the edges are assigned to partitions and thus the vertices can span partitions i.e vertices are replicated or mirrored across partitions. In edge-cut, the vertices are partitioned and the edge can span across partitions i.e edge is replicated or mirrored across partitions. GraphX utilizes the vertex-cut partitioning strategy. In vertex-cut partitioning, the goal of a partitioning strategy is to partition the edges such that the load (number of edges) in every partition is balanced and vertex *replication* (number of mirrors of vertex) is minimum. Average *replication factor* is a common metric to measure the effectiveness of vertex-cut partitioning.

The simplest vertex-cut partitioning strategy is to partition edges using a hash function. GraphX [15] has two different variants for this: *Random Vertex Cut* (RVC) and *Canonical Random Vertex Cut* (CRVC). Given a hash function h, RVC assigns an edge (u, v) based on the hash of the source and destination vertex (i.e. $A(u, v) = h(u, v) \bmod k$). CRVC partitions the edge regardless of the direction and hence an edge (u, v) and (v, u) will be assigned to the same partition. CRVC or RVC provides a good load balance due to the randomness in assigning the edges but do not grantee any upper bound on the replication factor. There is another strategy which uses two-dimensional sparse matrix and is similar to grid partitioning [5], EdgePartition2D [2]. In EdgePartition2D partitions are arranged as a square matrix, and for an edge it picks a partition by choosing column on the basis of the hash of the source vertex and row on the basis of the hash of the destination vertex. It ensures a replication factor of $(2\sqrt{N} - 1)$ where N is the number of partitions. In practice, these approaches result in large number of vertex replications and do not perform well for a power-law graphs.

Recently, a *Degree-Based Hashing* (DBH) algorithm [14] was introduced with improved grantees on replication factor for power-law graphs. DBH partitions edges based on the hash of its lowest degree end point thus forcing replication of high degree vertices. GraphX does not provide an implementation for this strategy. Thus, we implemented DBH and used it in our experiments to compare with other partitioning strategies provided in GraphX.

3 Cost Model for Pregel GraphX

In section 3.1, we present the implementation details of the Pregel model in GraphX with the help of a Business Process Model and Notation (BPMN) diagram. Then in Section 3.2, we use the BPMN diagram to derive the cost model for the Pregel model in GraphX.

3.1 Pregel Model in GraphX

GraphX is built on top of Apache Spark which uses a distributed data structure called Resilient Distributed Datasets (RDD) [16]. A graph in GraphX is represented as a pair of vertex and edge property collections namely *VertexRDD*

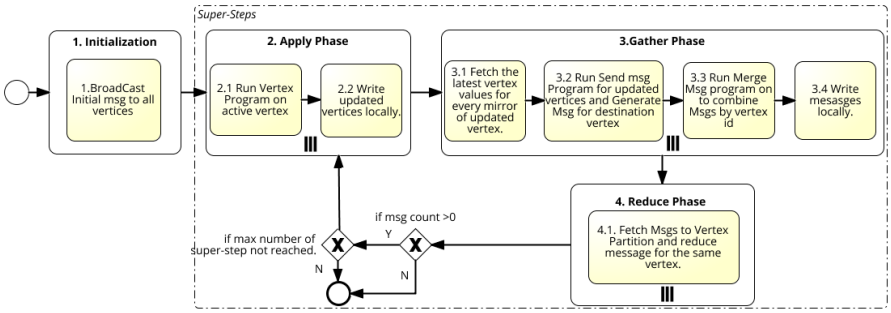


Fig. 2. BPMN diagram representing the Pregel computation model.

and *EdgeRDD*. The *VertexRDD* contains all the vertices of the graph and acts as the master copy, which runs the UPDATEVERTEX program. The *EdgeRDD* contains all the edge attributes and the vertex ids of the source and destination vertices. During Pregel execution, a materialized view (*EdgeTripletRDD*) is created by joining *VertexRDD* and *EdgeRDD* for the set of active vertices. The RDDs are partitioned across the cluster nodes and the computation happens in a shared-nothing architecture. The *VertexRDD* is partitioned randomly based on the hash of the vertex id and the *EdgeRDD* is partitioned using the graph partitioning strategy provided (vertex-cut strategies discussed in Section 2.2). *EdgeTripletRDD* is partitioned using the same partitioner used by *EdgeRDD*.

The Pregel computation in GraphX consists of four phases: Initialization, Apply, Gather and Reduce. The Initialization happens only once and the other three repeat in a loop until the program stops or a given maximal number of supersteps is exceeded. The Initialization phase, is executed by the driver/master as a single instance. The other three phases run in multiple instances. Each instance is processing of one partition of either the *VertexRDD* or *EdgeRDD*. After the Initialization phase the Apply phase runs one instance per partition of the *VertexRDD* and updates the vertices state. Then the Gather phase runs one instance per partition of the *EdgeRDD* to fetch the latest copy of the vertex state from *VertexRDD* and generate messages for next super-step. The Gather phase does a local reduce of the messages as well by combining all the messages generated for the same vertex on each instance. Finally, the reduce phase does a global reduce by combining of all the messages generated for the same vertex at vertex partitions. The reduce phase runs one instance per partition of the *VertexRDD*. Figure 2 shows all the phases and precedences. Please note, unlike the ideal Pregel model where every vertex could execute the vertex program in parallel and send and receive messages in parallel, in GraphX the parallelization is at the level of an instance or partition. For example, the vertex program of CC algorithm in GraphX will run during the Apply phase in parallel for every partition of the *VertexRDD*. Inside one partition of a *VertexRDD*, the vertex program will run in sequence for all the vertices.

3.2 The Cost model formulation

For the sake of simplicity of the cost model we make following assumptions:

1. All the nodes in the cluster have the same characteristics, i.e. they have same processing speed, IO and network bandwidth. This assumption does not reduce the applicability of the model, since extending it to heterogeneous nodes is straight forward.
2. Resource scheduling is not considered and hence, we assume all the instances run in parallel. This assumption is a natural choice to maximize performance as it offers maximum parallelization. To ensure this we just need to make sure that we keep the number of partitions to be equal to the number of available workers in the cluster.

From the BPMN diagram in Figure 2, it is clear that the cost of the Pregel job is the sum of the costs of four phases. We represent the cost of the Initialization phase as a function $cInit$ which depends on: the vertices (V), the algorithm (A) which determines the cost of creating the initial message and its size, and finally, the number of vertex partitions to which the initial message will be sent. We combine the remaining three: Apply, Gather and Reduce phases, in function $cSuperStep$, representing the cost of the subsequent super-steps. Let s be the number of super-steps. Hence, we can represent the cost of the Pregel model ($cPregel$) as shown in Equation (1). For a super-step i the cost $cSuperStep$ depends on: currently active vertices (V_i), currently active edges (E_i) and the messages (M_{i-1}) generated in previous super-step. How a vertex or an edge becomes active depends on the algorithm (A). We define A_v, A_s, and A_m as three functions for updateVertex, sendMsg, and mergeMsg programs respectively. Additionally, $cSuperStep$ also depends on how V_i and E_i is partitioned (i.e., vertex partitioning strategy (P_v) and edge partitioning strategy (P_e)).

$$cPregel(V, E, s, A, P_e, P_v) := cInit(V, A, |P_v|) + \sum_{i=1}^{s} cSuperStep(V_i, E_i, A, M_{i-1}, P_e, P_v) \quad (1)$$

The Apply, Gather and Reduce phases run in sequence and hence the cost of one super-step is the sum of the cost of each phase. But, as shown in the BPMN diagram there are multiple instances of each phase. As per our assumption, we have all the instances running in parallel in the cluster. Hence, we denote the cost of running one phase as the maximum cost among all the instances of that phase. There are tasks inside each phase which run sequentially except in the case of Reduce phase where there is only one task. Let $|P_v|$ and $|P_e|$ be number of vertex and edge partitions respectively, and q ($0 \leq q \leq |P_v|$) and k ($0 \leq k \leq |P_e|$) as corresponding index of vertex or edge partition. We define, $E_i^k \subset E_i$ as set of active edges on a partition k; V_i^k as set of vertices at super-step i in edge partition k which is either a source or destination vertex of an active edge E_i^k; $V_i^q \subset V_i$ as set of active vertices in vertex partition q; M_i^k as

set of messages generated in super-step i in edge partition k; $M_i^q \subset M_i$ as set of messages received in super-step i in vertex partition q. We represent the cost of each super-step as shown in Equation (2).

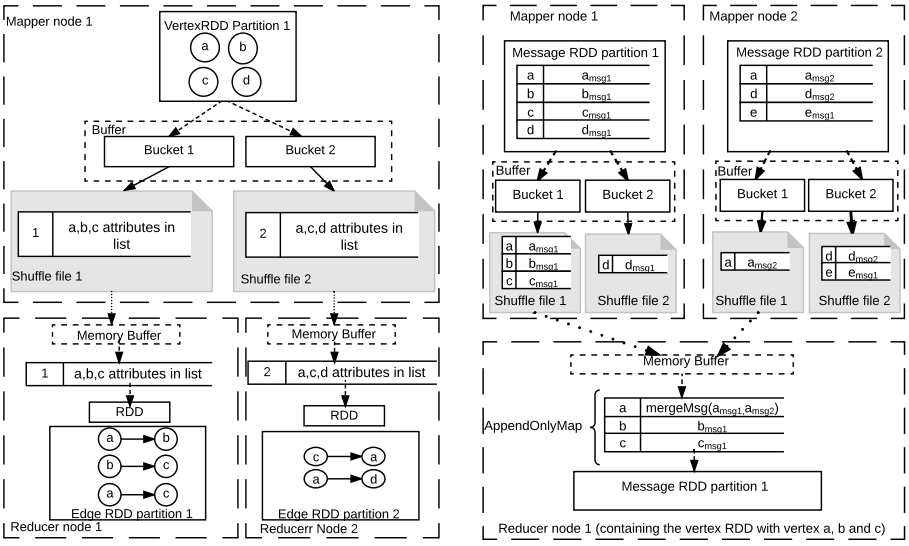
$$\begin{aligned} cSuperStep(V_i, E_i, A, M_{i-1}, P_e, P_v) := & \max_{0 \le q \le |P_v|} \{cApply(V_i^q, M_{i-1}^q, A_v, P_e, P_v)\} \\ & + \max_{0 \le k \le |P_e|} \{cGather(E_i^k, M_i^k, V_i^k, A_s, A_m, P_e)\} \\ & + \max_{0 \le q \le |P_v|} \{cReduce(M_i^q, V_i^q, A_m, P_e, P_v)\} \end{aligned} \tag{2}$$

As shown in Figure 2, the Apply phase has two tasks:

- The first task is to run the UPDATEVERTEX program on the active vertices. It runs sequentially for every vertex in the local partition. Hence, the total cost of the first task is defined as the sum of the cost of running the UPDATEVERTEX program for every active vertex in the partition, which depends on the vertex state, the input message and the algorithmic characteristics. We capture all this as a function $cVertexProg$ and assume its cost is known to the user defining the algorithm.
- The second task is to write the updated vertex attributes to file so that it can be sent to required edge partitions. It consists of creating $|P_e|$ different file segments, one for each edge partition. The writing is buffered, so each write task writes in an internal memory buffer of size B_s, and when the buffer is full, the content is flushed to the file segment. For example, in Figure 3a the mapper node having the vertex partition 1 with vertices a, b, c, d will create two files. As one vertex can have its replication in more than one edge partition, it needs to be written in more than one file segment. Let $V_i^{*q} \subseteq V_i^q$ be the set of vertices which updated their state after the first task. We define $replication(v)$ as the number of replication of vertex v in edge partitions and $sizeOf(v)$ as the size of vertex object v in bytes. Hence, the total blocks written would be equal to the size of every vertex object times its replication. Let B_w be the cost of writing one block and B_s be the size of one block, hence the total cost for this task would be $B_w \times \frac{Total\ bytes\ written}{B_s}$.

Apart from the cost of the above mentioned task we define α_1 as a constant to capture some housekeeping tasks done by Spark (like task scheduling) for this phase. We use α_2 and α_3 as separate constant costs for the other two phases. The cost of Apply phase is given as the sum of the cost of the two task and the constant α_1 in Equation (3).

$$\begin{aligned} cApply(V_i^q, M_{i-1}^q, A_v, P_e, P_v) := & \sum_{v \in V_i^q} cVertexProg(v, M_{i-1}^q(v), A_v) \\ & + \beta_w \times \left\lceil \frac{\sum_{v \in V_i^{*q}} sizeOf(v) \times replication(v)}{B_s} \right\rceil + \alpha_1 \end{aligned} \tag{3}$$



(a) Apply - Gather Phase (b) Gather - Reduce Phase

Fig. 3. Data shuffle between the phases. Dashed arrows represent in-memory data transfer, Solid arrows represent memory to local disk write and dotted arrows represent remote disk to memory read.

The Gather phase consists of four tasks :

- The first task consists of reading the file segments created in the previous phase. For simplicity, we focus only on the remote reads as local reads are quite fast and do not affect the overall cost significantly. Each file will be read and deserialized to create or update an AppendOnlyMap (an internal data structure used by Spark to create an RDD). In this case there is only one key in the map (the partition id) and the value is a list with vertex attributes. For example, as shown in Figure 3a there is only one record in the map with key "1" and value a list of vertex attributes of a, b and c. The AppendOnlyMap is then converted into an RDD and combined with *EdgeRDD* to generate *EdgeTripletRDD*. As the number of records in the map is just one, the cost of this task is due to the size of the list. Let V_i^* be the set of all vertices which got updated in previous phase, then the list of vertices read in this task is given as $V_i^k \cap V_i^*$. We represent the total cost of this task as total bytes read multiplied by the cost of reading and deserializing one byte (β_r).
- The second task consists of running the SENDMSG program on every active edge. It depends on the attributes of the source and destination vertices and the algorithm definition A_s. We capture this cost as a function *cSendProg*. Hence, the total cost for this task is given as the sum of running the *cSendProg* for every active edge.

– The third task consist of running the MERGEMSG program to combine all the messages generated for a vertex $v \in V_i^k$. We define the cost of running MERGEMSG program which combines two messages as $cMergeProg$. It depends on the algorithm definition A_m. We define $M_i^k(v)$ as the set of messages generated for a vertex v. MERGEMSG will run $|M_i^k(v)| - 1$ times.
– The final task is the shuffle write task, which consists of writing to disk the final list of reduced messages $\widehat{M_i^k}$ as shown in Figure 3b. The writing will be buffered as in the Apply phase, but the number of records written will be equal to the number of final messages ($|\widehat{M_i^k}|$). One message can belong only to one shuffle file, hence the total blocks written would be size of all messages divided by the block size.

The cost of the Gather phase is defined as the sum of the cost of the four tasks and the constant α_2 given in Equation (4).

$$\begin{aligned} cGather(E_i^k, M_i^k, V_i^k, A_s, A_m, P_e) := \beta_r \times & \sum_{v \in V_i^k \cap V_i^*} sizeOf(v) \\ & + \sum_{(u,v) \in E_i^k} cSendProg(u, v, A_s) \\ & + cProcess(M_i^k, V_i^k, A_m) \\ & + \beta_w \times \left\lceil \frac{\sum_{m \in \widehat{M_i^k}} sizeOf(m)}{B_s} \right\rceil + \alpha_2 \end{aligned} \tag{4}$$

Where,

$$cProcess(M_i^k, V_i^k, A_m) := \sum_{v \in V_i^k} \left(|M_i^k(v)| - 1 \right) \times cMergeProg(A_m) \tag{5}$$

The Reduce phase consists of only one task which is to fetch the messages generated in the previous phase and reduce the messages for the same vertex into one message. For example, as shown in Figure 3b a_{msg1} and a_{msg2} are fetched from two mappers and reduced into one message for vertex a. Unlike the read in the Gather phase, in this phase the number of records in the AppendOnlyMap will be equal to the numbers of messages. For example, as shown in Figure 3 there is one record in the shuffle file for the Gather phase where as upto 3 records in the shuffle file for the Reduce phase. The size of each message record is constant, hence the cost of the read is dominated by the number of records and not the size of the record. We define γ as the constant cost of reading and updating the AppendOnlyMap per record. Thus, we can define cost for the read task as γ times number of records fetched. The reducing of the messages can start as soon as there are two messages for the same vertex. As Spark uses parallel threads to read data and process data, there will be an overlap in the execution of these tasks. Hence, in a multi-core system, as soon as first block of messages is read, it can start processing the messages while in parallel keep fetching remaining

blocks. Let C be the number of cores in a cluster node; hence C threads can fetch data in parallel. Let b be the number of blocks of messages received in this phase and M^b represent the set of messages in the b^{th} block. Then, the overall cost of this phase is given as the sum of the cost of fetching the first block plus the cost of processing all messages (if processing is slower than fetching) or the cost of fetching remaining blocks plus processing the last block (if fetching is slower than processing) as expressed in Equation (6).

$$\begin{aligned} cReduce(M_i^q, V_i^q, A_m, P_e, P_v) := \gamma \times |M^1| \\ + max\ \{cProcess(M_i^q, V_i^q, A_m), \\ \frac{\gamma}{C} \times \sum_{2 \leq j \leq b} |M^j| + C \times cProcess(M^b, V_i^q, A_m)\} \\ + \alpha_3 \end{aligned} \tag{6}$$

For a single core node, the fetching of data and processing can not run in parallel, hence Equation (6) simplifies to the sum of the cost of fetching all messages and processing them as given in Equation (7).

$$\begin{aligned} cReduce(M_i^q, V_i^q, A_m, P_e, P_v) := \gamma \times |M_i^q| \\ + cProcess(M_i^q, V_i^q, A_m) + \alpha_3 \end{aligned} \tag{7}$$

4 Experimental Validation of the Cost Model

In this section, we describe the experimental setup to obtain the cluster specific variables (α_1, α_2, α_3, β_r, β_w and γ) in the cost model and then share the results of the validation of the cost model on different configurations.

4.1 Experiment Configuration and Setup

There are four main parameters which affect the execution of a GraphX Pregel job: 1) Cluster setup, 2) Input Graph, 3) Partitioning Strategy, and 4) Graph Algorithm to be executed. In our experiments, we always keep the cluster setup constant and vary the other three. All experiments are done on a cluster with a master node and 5 worker nodes. All nodes are Linux systems with Intel Xeon E5-2630L v2 a 2.40 GHz processor, 1 TB SATA-3 Hard disk, 128 GB RAM, and 4 GB Ethernet. We deployed Spark 2.0.2 in cluster mode with each worker node having 1 executor with 1 thread and 45 GB RAM assigned to it.

Input Graph: We used three real world datasets: the `CollegeMsg` network is a directed graph of messages sent between users on a Facebook-like platform at UC-Irvine; Higgs activity time (`Higgs`) is a dataset which provides information about activity on Twitter during the discovery of the Higgs boson particles (both datasets were taken from the SNAP repository [9]); Apart from this, we also use a re-tweet network collected from information about activity on Twitter during the Punjab Election 2017 (`twitter`) in India collected by ourselves for 3 days.

Partitioning Strategy: We use three partitioning strategies in the experiments: EdgePartition2D; Canonical Random Vertex Partitioning(CRVC) (both strategies provided by the default GraphX API) and our own implementation of Degree Based hashing (DBH). As explained earlier, these partitioning strategies only partition the *EdgeRDD*. For *VertexRDD* we used the default random Hash Based partitioner provided by Spark. The number of partitions was equal to 5 in all experiments.

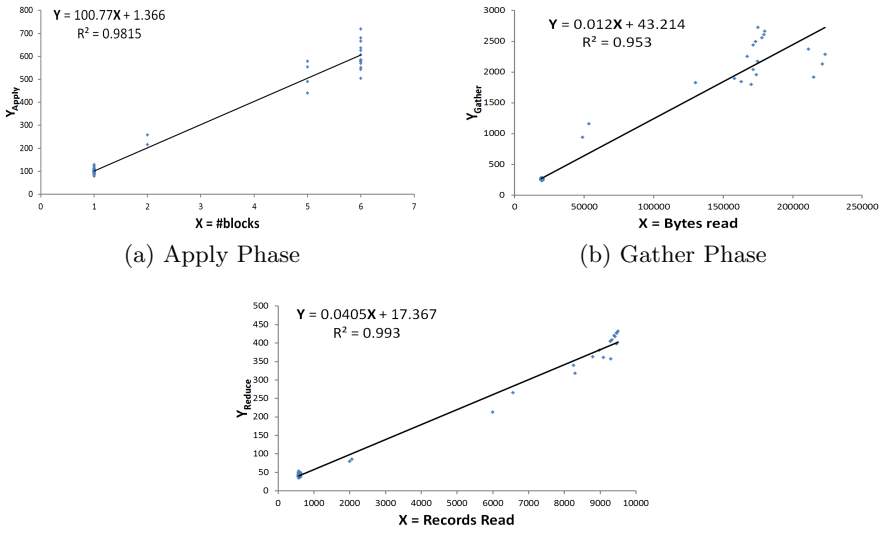
Graph Algorithm: We used the classical PageRank and Connected Component algorithms in our experiments.

4.2 Estimating α_1, α_2, α_3, β_r, β_w and γ

Monitoring the factors in the cost model is not straightforward. Hence, we applied following simplifications to approximate the value of the constant parameters:

1. We used the same code provided in GraphX for the Page Rank and Connected component algorithms but just added additional counters on each of the three GraphX functions to keep a count of how many times the UPDATEVERTEX, SENDMSG and MERGEMSG programs were executed in each task of a super-step.
2. The execution time of the three functions is very small and difficult to monitor precisely. A more accurate measurement of these functions allows for a more accurate estimation of the cluster constants in the formula, hence we introduced a constant time delay of 1 millisecond in all three functions. This constant time delay is only for accurate estimation of the cluster parameters and does not affect the cost model accuracy. Let $count(f)$ be the number of times a program f is executed in an instance. This enables us to approximate:
   - $\sum cVertexProg(v, M_{i-1}^{q}(v), A_v) = count(\text{UPDATEVERTEX}) \times 1$ msec
   - $\sum cSendProg(u, v, A_s) = count(\text{SENDMSG}) \times 1$ msec
   - $\sum \left(|M_i^k(v)| - 1\right) \times cMergeProg(A_m) = count(\text{MERGEMSG}) \times 1$ msec
3. We kept the number of edge partitions, vertex partitions and number of nodes in the cluster equal, so that every node in the cluster is processing only one partition of the *VertexRDD* and *EdgeRDD* (i.e $|P_e| = |P_v| = N$).
4. Every node has only one core assigned to it (i.e $C = 1$), hence we can use Equation 7 for the reduce phase.

We used `twitter` graph data with the CRVC partitioning strategy and the Page Rank algorithm to estimate the constants $\alpha_1, \alpha_2, \alpha_3, \beta_r, \beta_w$ and γ of the cost model. We used the SPARK UI API (a monitoring service provided by Spark) to get the run time of each phase separately and other factors of the cost model. Since we used a shared cluster while running the experiments, we repeated the experiments 10 times and took the minimum execution time of a super-step as the baseline cost of that super-step, assuming that higher time to execute the same super-step is due to the interferences with parallel executions



(a) Apply Phase (b) Gather Phase

(c) Reduce Phase

Fig. 4. Using Linear curve fitting to estimate the variables in the cost model

of other processes on the cluster. *cInit* is a constant one time cost for a graph and algorithm and do not change based on the partitioning strategy hence we do not estimate this cost for every partitioning strategy.

We estimated the value of α_1 and β_w from Equation 3 by substituting the values of all other factors. For every super-step, we replaced *cApply* by the execution time of the phase, $\sum cVertexProg(v, M_{i-1}^q(v), A_v)$ by *count(*UPDATEVERTEX*)* and the number of blocks written by total bytes written divided by 32 MB (the default value of B_s in Spark), for the task which took the maximum time for this phase. Substituting these values, results in a linear equation of the form $Y = \beta_w \times X + \alpha_1$ where $Y = cApply - count(\text{UPDATEVERTEX})$ and X is the number of blocks written. We got the value of X and Y for all the super-steps and obtained α_1 and β_w by ordinary least square (OLS) method. The result of the linear curve fitting is show in Figure 4a. We get $\alpha_1 = 1.366$ msec and $\beta_w = 100.77$ msec/block with a R-squared value of 0.9815. We believe the deviation(outliers) from the line is due to discretization of the write bytes into number of buckets as for some cases the last bucket would be almost full and for some it will be almost empty resulting in different write time. Similarly, we estimated α_2 and β_r from Equation 4 by replacing β_w with 100.77; *cGather* by the stage execution time. For the right hand side parameters of the equation we substituted values for the longest running task. Hence, we replaced $\sum_{v \in V_i^k \cap V_i^*} sizeOf(v)$ by the volume of remote bytes read by the task, $cProcess(M_i^k, V_i^k, A_m)$ by *count(*MERGEMSG*)*, $\sum_{(u,v) \in E_i^k} cSendProg(u, v, A_s)$ by *count(*SENDMSG*)* and the number of blocks written by the volume of total bytes written by the task divided by 32 MB. Sub-

Dataset	Algorithm	Partition Strategy		
		EdgePartition2D	CRVC	DBH
`CollegeMsg`	PageRank	96.4	97.9	97.7
	CC	97.6	96.1	96.7
`twitter`	PageRank	97.7	-	99.3
	CC	98.9	98.7	97.1
`Higgs`	PageRank	94.6	97.2	99.8
	CC	97.9	95.9	94.9

Table 1. Prediction accuracy(%) of the cost model for different combinations of dataset, partitioning strategy and graph algorithm.

stituting these values, results in a linear equation of the form $Y = \beta_r \times X + \alpha_2$, where $Y = cGather - count(\text{MERGEMSG}) - count(\text{SENDMSG}) - \beta_w \times \#blocks$ and X is remote bytes read. After applying OLS we get $\alpha_2 = 43.214$ msec and $\beta_r = 0.012$ msec/byte with a R-squared value of 0.953 as shown in Figure 4b. Similarly, from Equation 7 we get a linear equation of the form $Y = \gamma \times X + \alpha_3$ where, $Y = cReduce - count(\text{MERGEMSG})$ and X is the number of message records. We get $\alpha_3 = 17.367$ msec and $\gamma = 0.0405$ msec/record with R-squared value of 0.993 as shown in Figure 4c.

4.3 Cost model validation

We used 3 different graph data, 3 different edge partitioning strategy and 2 different graph algorithms in our experiments resulting in 18 different combinations of graph, partitioning strategy and algorithm. In order to validate the cost model, we estimated the cluster constants $\alpha_1, \alpha_2, \alpha_3, \beta_r, \beta_w$ and γ in the cost model for graph= `twitter`, partitioning strategy=CRVC and algorithm= Page Rank (Section 4.2), then we used other 17 combinations of graph, partitioning strategy and algorithm to estimate the execution cost. We replace the values of $\alpha_1, \alpha_2, \alpha_3, \beta_r, \beta_w$ and γ in the cost model and predict the job execution time by measuring other attributes required by the cost model. Then we estimate the accuracy of the cost model by comparing with the actual execution time of all the super-steps. We report the prediction accuracy in Table 1. We get 96.9% average accuracy in predicting the job execution time in 17 different combination with minimum accuracy of 94.6% and maximum accuracy of 99.8%.

5 Concluding remarks

We presented a cost model to estimate the execution cost of Pregel-based algorithms on Spark GraphX and evaluated on different combinations of input graph, algorithm and partitioning strategy. We see from the cost model that the overall execution time depends on different factors such as: the execution time of each function (i.e., UPDATEVERTEX, SENDMSG and MERGEMSG); the cluster

configuration (such as data transfer between different nodes). The cost model depends on many variables which are not known before hand and hence, for an optimizer, they will need to be estimated. In future work, we will experiment by varying the different dominating factors in the cost model, to see how they determine the best partitioning strategy.

Acknowledgement

This work was supported by the Fonds de la Recherche Scientifique-FNRS under Grant(s) n T.0183.14 PDR. The student is also part of IT4BI DC program.

References

1. Barnard, S.T.: Parallel multilevel recursive spectral bisection. In: Proceedings of the 1995 ACM/IEEE conference on Supercomputing. p. 27. ACM (1995)
2. Çatalyürek, Ü.i.t.V., Aykanat, C., Uçar, B.: On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. SIAM Journal on Scientific Computing (2010)
3. Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., Muthukrishnan, S.: One trillion edges: Graph processing at facebook-scale. VLDB (2015)
4. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: OSDI (2012)
5. Jain, N., Liao, G., Willke, T.L.: Graphbuilder: scalable graph etl framework. In: GRADES (2013)
6. Karypis, G., Kumar, V.: Multilevel graph partitioning schemes. In: ICPP (3) (1995)
7. Kumar, R., Calders, T.: Information propagation in interaction networks. In: Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017 (2017)
8. Kumar, R., Calders, T., Gionis, A., Tatti, N.: Maintaining sliding-window neighborhood profiles in interaction networks. In: ECML-PKDD. Springer (2015)
9. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (Jun 2014)
10. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.: Challenges in parallel graph processing. Parallel Processing Letters (2007)
11. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD (2010)
12. Petroni, F., Querzoni, L., Daudjee, K., Kamali, S., Iacoboni, G.: Hdrf: stream-based partitioning for power-law graphs. In: CIKM. ACM (2015)
13. Verma, S., Leslie, L.M., Shin, Y., Gupta, I.: An experimental comparison of partitioning strategies in distributed graph processing. Proc. VLDB Endow. (2017)
14. Xie, C., Yan, L., Li, W.J., Zhang, Z.: Distributed power-law graph computing: Theoretical and empirical analysis. In: Advances in Neural Information Processing Systems (2014)
15. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: A resilient distributed graph system on spark. In: GRADES. ACM (2013)
16. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. USENIX Association (2012)