Performance Modeling of Computation and Communication Tradeoffs in Vertex-Centric Graph Processing Clusters

Amirreza Abdolrashidi
Department of Computer Science
The University of Georgia
ara@cs.uga.edu

Lakshmish Ramaswamy
Department of Computer Science
The University of Georgia
laks@cs.uga.edu

David Seamus Narron
Department of Computer Science
The University of Georgia
narron@cs.uga.edu

Abstract—**Distributed vertex-centric graph processing systems have been recently proposed to perform different types of analytics on large graphs. These systems utilize the parallelism of shared nothing clusters. In this work we propose a novel model for the performance cost of such clusters. We also define novel metrics related to the workload balance and network communication cost of clusters processing massive real graph datasets. We empirically investigate the effects of different graph partitioning mechanisms and their tradeoff for two different categories of graph processing algorithms.**

Keywords*—*Distributed Vertex-Centric Graph Processing, Performance Modeling, Parallel Processing, Graph Partitioning.

I. Introduction

The importance of graph as fundamental structures for representing, understanding, and analyzing relationships in many diverse domains can hardly be overemphasized. This is evidenced by the fact that graph storage, querying, analytics and mining have continued to be highly active areas of research over the past several decades. However, the graphs in many emerging applications such as social networks, WWW and bioinformatics are massive. Traditional centralized graph computation algorithms cannot scale to these sizes. Recently, research has focused on harnessing the parallelism offered by shared nothing clusters for graph analytics. Frameworks such as Pregel [1], Giraph [2], GraphLab [3] and GPS [4] are notable efforts in this direction. Most of these frameworks are based upon Bulk Synchronous Parallel (BSP) [5] model for parallel computers. In the paradigm, the individual vertices of the graph form the programming units. The computation logic is expressed as a series of iterations, called supersteps. In a given superstep, each vertex performs certain computations which involves processing messages that it received from the previous superstep, updating its own state and sending messages to its neighboring vertices. The synchronization occurs at the end of each superstep.

One of the factors that is critical to the performance of vertex-centric BSP frameworks is the manner in which the graph data is partitioned and placed on various machines of the cluster. This is so because graph partitioning not only affects the amount of communication that needs to happen at the end of each superstep but also the computational loads placed on the machines during the computation. However, the interactions among various aspects of vertex-centric graph processing paradigm such as loads on individual processors, available network bandwidth and the nature of the graph processing algorithms (in terms of their computation localization properties), and their effects on the performance of various graph partitioning strategies are not well understood.

The objective of this paper is to study the behaviors of graph partitioning strategies on the performance and scalability of vertex-centric graph processing clusters. Towards this end, we first present a novel model to analyze the performance of vertex-centric graph processing clusters. This model works at the granularity of individual supersteps, and it incorporates important parameters such as computational loads on processors, messaging loads between pairs of processors and available computation and communication resources with in the cluster. This model can be used to theoritcally compare different graph partitioning strategies. Second, we present a novel categorization of graph algorithms based on their message passing and load distribution behaviors over the duration of the computation. Third, we provide a detailed experimental study involving massive real world graphs (millions of vertices and edges) on Amazon EC2 clusters with varying number of compute nodes. We also introduce novel metrics to accurately measure the load balancing and communication charachteristics of vertex-centric graph computations.

The rest of the paper is as follows: In section 2 we describe background and vertex-centric graph processing model of computation. In section 3 we describe the performance cost of this model and provide formal specifications. In the next section we discuss the metrics used for measuring the performance in our experiments. This sections also includes the experiments setups and Results. In section 5 we discuss the related works. The paper will conclude in section 6.

II. Background and Motivation

Since the introduction of MapReduce [6], many systems have used this model to process large graphs. In these systems, graph algorithms can be modeled effectively as a sequence of chained MapReduce jobs. However, this model is not appropriate for performing graph algorithms from the perspective of both data and computation model [7], [8].

COLLABORATECOM 2014, October 22-25, Miami, United States
Copyright © 2014 ICST
DOI 10.4108/icst.collaboratecom.2014.257474

Initially, the data need to be modeled as key-value pairs in order for MapReduce jobs to proceed. However, modeling graph algorithms as sequences of such functional programming constructs that operate on key-value pairs are not necessarily efficient or easy. Additionally, graph algorithms by nature are iterative where computation includes several iterations of similar operations that are performed on graph vertices. Using MapReduce to model graph algorithms will yield iterative disk intensive jobs in which the entire state of graph should be transferred from one iteration to another in order for the computation to proceed. During computation of the arbitrary graph algorithm in a large scale environment, distribution over many machines makes memory access time as well as I/O overhead worse and consequently affects the performance of the framework [1].

Another computation model that has been used recently for parallel processing of large graphs is Valiant's Bulk Synchronous Parallel (BSP) [5] model. BSP (as mentioned in [5]) is a "bridging model" for general purpose parallel computation, that is parallelism across a wide range of applications and architectures. In BSP paradigm, computation consists of a series of iterations named supersteps. Each superstep is divided into three phases as follows:

- *Computation*, where each process using local data stored in memory of its processor performs the computation.
- *Communication*, where the processes send and receive messages needed for the computation to proceed.
- *Barrier Synchronization*, where all the communications are complete and the data sent by processors are available for the destination processors in the next superstep.

Computation based on this parallel model will finally be terminated once it goes through the desired number of supersteps or a specific convergence criterion passes a certain threshold.

Pregel [1] used the above computation model to process large graphs by applying a vertex-centric approach to implementation of graph algorithms. In this approach, vertices of the graphs are considered as the work units that are partitioned among processor nodes of a cluster. At the beginning of each superstep, in the *computation* phase the processor nodes of the cluster receive messages from the previous superstep and perform the user defined logic of computation in parallel on each of the work units by running a compute method on each of the vertices. Once a processor node finishes the processing of its work units it begins the *communication* phase where it sends messages to other vertices (possibly located at other processor nodes) along graph edges. Finally, in the *synchronization* phase the processor nodes will wait for the slowest processor to finish processing and sending its messages and then they become synchronized. This marks the end of one superstep and afterwards all the processor nodes start the next superstep.

In order for the computation to terminate, graph vertices need to inform each other whether they participate in the computation. In other words, they need to be stateful. Pregel considers two states for each graph vertex along with a mechanism called *voting to halt* (that involves graph vertices changing their states between the two states) to achieve statefulness of vertices and subsequently mark the end of computation. Initially in the first superstep all the graph vertices are in the *active* state. Active graph vertices participate in the computation. After the computation on each active vertex is completed, it changes its state to *inactive* in order to inform other vertices that it has no further work to do. Inactive vertices will not participate in the computation in the next superstep unless they receive a message from other vertices at which point they will change their state back to active and will participate in the computation. The computation ends when all the graph vertices are in the inactive state and there are no messages left among vertices to process.

Vertex-centric processing of graphs with the use of BSP computational model leads to interaction of several factors that ultimately determines the performance of the underlying cluster. In the next section, we identify these factors and provide accurate and tractable model that describe the role of each of these elements.

III. Performance Model of Vertex-Centric Graph Processing

Vertex-centric processing of graphs is of nature of parallel computing. Similar to other parallel computations two major factors that determine the performance of the computation are the communication cost among the processor nodes of the cluster and the computation cost of each of the processor nodes of the cluster.

As mentioned in the previous section, in the vertex-centric processing of a graph, an active vertex sends a message to another vertex if there is an edge between them. Consequently, the manner in which graph vertices are placed among processor nodes affects the communication and computation of the cluster and hence plays a key role in determining the performance of the cluster. One approach to placement of graph vertices on processor nodes can then be applying heuristics based graph partitioning algorithms in order to compute the min-cuts of the graphs with the goal of lowering the communication cost. For instance, consider Fig.1 which illustrates the distribution of a sample graph vertices on three processor nodes based on the result of min-cut computation. As it is depicted, each partition has same number of vertices. Moreover, each highly connected partition of the graph is located on a processor node. There are four intra-clusters edges (highlighted in bold) among processor nodes. Such approach to placement of graph vertices on processor nodes results in lower communication cost among processor nodes. This is due to the fact that the majority of the edges (which vertices will send messages along) are accessible by each processor. Consequently, there will be fewer messages that need to be sent among processor nodes in order to access graph vertices. This approach, however, leads to load imbalance among processor nodes as we will demonstrate later.

The other approach for placement of graph vertices on processor nodes can be random partitioning of the graph vertices. Fig. 2 represents the random partitioning of the same exemplary graph of Fig. 1. As can be seen, in this approach the communication among processor nodes is higher as there are more intra-cluster edges. In this manner, processor nodes need to constantly send messages among each other for the computation to proceed. Despite resulting in higher communication cost, we will see that the advantage of such partitioning is in achieving a better load balance among processor nodes in case of some graph algorithms.

As it is exemplified in both figures 1 and 2, there exist a trade off among computation and communication costs of the cluster depending on how the graph vertices are assigned to processor nodes. Next, we propose a formal performance cost model for vertex-centric processing of graphs on a cluster which reflects these trade offs and enables analyzing the efficiency of vertex-centric graph processing algorithms.

A. Performance Cost Model

As described in section two, a superstep is divided into three phases. Hence, the cost of a superstep relies on the cost of each of its three phases: the (maximum) cost of the local computation on each processor, the (maximum) cost of the communication among processors and the cost of the barrier synchronization at the end of the superstep. Thus the cost of a superstep can be formulated as:

$$\begin{aligned} \textit{Cost of a Superstep} = \max_{1 \le i \le n} (\textit{Comp\_Cost}_{P_i}) + \\ \max_{1 \le i \le n} (\textit{Comm\_Cost}_{P_i}) + l \end{aligned} \tag{1}$$

where $\textit{Comp\_Cost}_{P_i}$ and $\textit{Comm\_Cost}_{P_i}$ are computation cost and communication cost of the processor node *i* respectively and n is the number of processor nodes. In the following sections we provide formal descriptions for the cost of each of the three phases and explain the factors that affect each of them. In this paper we use the terms "cost" and "time" interchangeably.

1) Computation: The cost of computation is related to the amount of the load on the processors. In order to model the cost of the computation phase of a single superstep, the first criterion might be measuring the number of vertices that each processor has to process during that superstep.

However, this approach is naive. The reason is due to the fact that during each superstep not all vertices of a processor node are active and only the vertices that are in the active state will participate in the computation and constitute the load on a processor node. For instance, all of the processor nodes in both figures 1 and 2 have the same number of vertices but it does not mean that they have the same amount of load. This is because not all of the vertices might be active during a superstep. This factor depends on the behavior of the graph algorithm in terms of message passing as we will see in the next section. Thus, in order to measure the load of a processor node, *Number of Active Vertices (NAV)* reflects the right metric.

However, the mere measurement of the number of active vertices is not sufficient to model the load of a processor node. Another factor in determining the load is the *Number of Messages* that a processor needs to process at the beginning of each superstep. There can be circumstances where two processor nodes have the same number of active vertices in a superstep but one might have to process more messages to determine the final number of active vertices that will participate in the computation. This factor depends on the

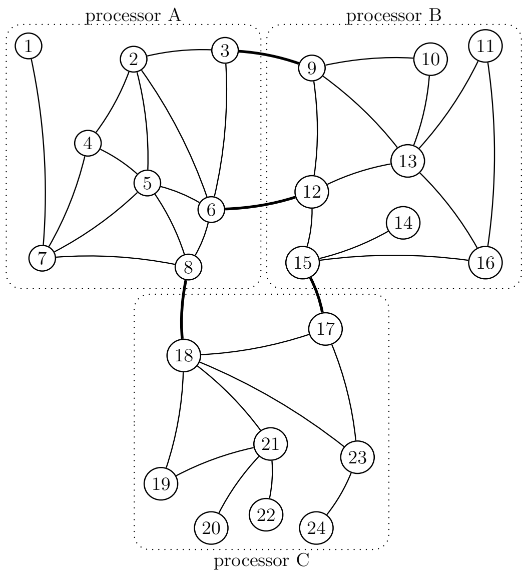

Fig. 1: Min-cut partitioning of a sample graph G(24,36) among three processors; each processor node has 8 vertices; 4 intra-cluster edges (highlighted in bold) and 32 inter-cluster edges

underlying graph structures and the distribution of degree of the vertices. For instance as can be seen in both figures 1 and 2, vertices 1, 14, 20, 22 and 24 have degree of one and hence have to process one message during a superstep whereas vertices 5, 6, 13, 18 and 21 have higher degree and have to process more messages.

Considering the above two factors, the detailed cost of the computation phase of a single superstep can be formulated as:

$$\textit{Comp\_Cost}(SS_k) = \max_{1 \le i \le n} (AV) \times \alpha + \max_{1 \le i \le n} (d_i \times \gamma) \times \beta \tag{2}$$

where the first factor is the maximum number of *Active Vertices* (*AV*) over *n* processor nodes multiplied by cost of main operation α of computation (e.g. addition of multiplication) and the second factor is the maximum number of messages that a graph vertex with input degree *d* might receive with probability γ times β, the cost of processing a message by processor. It should be noticed that in the first superstep the second factor would be zero because there would be no messages to receive and process from the previous superstep.

2) Communication: The second factor that determines the time of a single superstep is the length of the communication phase. Since the processors in the cluster communicate in parallel, this time is the maximum time it takes the communication network to deliver messages among processors. This cost is related to maximum number of bytes that has to be sent or received during a superstep and also is dependent on available network bandwidth. We model this cost formally as:

$$\textit{Comm\_Cost}(SS_k) = \max_{1 \le i \le n} (\textit{SentBytes}_{P_i}), \textit{ReceivedBytes}_{P_i})/g \tag{3}$$

where the nominator of the fraction is the maximum number of bytes to be sent or received by processor *i* among *n* processors and *g* is network bandwidth.

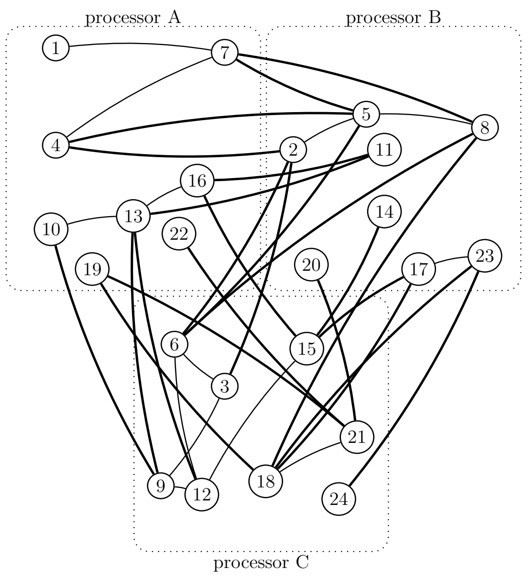

Fig. 2: Random partitioning of a sample graph G(24,36) among three processors; each processor node has 8 vertices; 24 intra-cluster edges (highlighted in bold) and 12 inter-cluster edges

3) Synchronization: The final determining factor of time of a superstep is the time it takes for all processor nodes to become synchronized and ready for the execution of the subsequent superstep. This would be a constant cost and is dependent on the cost of sending a single message across the diameter of the network among processor nodes in order to synchronize themselves. We will not measure this cost in our experiments and consider it as a constant cost *l*.

Considering the above equations we measure the time of a single superstep when processing a vertex-centric graph computation as follows:

$$\textit{Cost of a Superstep} = \max_{1\leq i\leq n}(\textit{AV}) \times \alpha + \max_{1\leq i\leq n}(d_i \times \gamma) \times \beta + \max_{1\leq i\leq n}(\textit{SentBytes}_i, \textit{ReceivedBytes}_i)/g + l \tag{4}$$

We begin our discussion by providing the pseudo codes for two graph algorithms, namely PageRank [9] and Dijkstra's algorithm to Single Source Shortest Path problem [10] (SSSP, here after) in Alg. 1 and 2, respectively. We have chosen these two graph algorithms because of their differences in their message passing behavior. In vertex-centric parallel processing of a graph, a vertex changes its state to active upon receiving a message from the previous superstep. Thus the manner in which a graph vertex sends messages to other vertices can be different for different graph algorithms. We provide the following categories for graph algorithms.

B. Categorization of Graph Algorithms

We begin our discussion by providing the pseudo codes for two graph algorithms, namely PageRank [9] and Dijkstra's algorithm to Single Source Shortest Path problem [10] (SSSP, here after) in Alg. 1 and 2, respectively. We have chosen these two graph algorithms because of their differences in their message passing behavior. In vertex-centric parallel processing of a graph, a vertex changes its state to active upon receiving a message from the previous superstep. Thus the manner in which a graph vertex sends messages to other vertices can be different for different graph algorithms. We provide the following categories for graph algorithms.

Assuming that a BSP based graph algorithm computation is defined to be executed for n supersteps it can be categorized as either:

- *Globally Active Algorithms*: if vertex i sends a message to vertex j at superstep k, it will send a message to vertex j in superstep $k+1$. The computation of these algorithms will terminate when exactly n number of supersteps has been executed and finished. PageRank algorithm is an example of this category.
- *Locally Active Algorithms*: if vertex i sends a message to vertex j at superstep k, it might not necessarily send a message to vertex j in superstep $k+1$. The computation of these algorithms might not necessarily terminate when exactly n predefined number of supersteps is reached. SSSP is representative of this category.

As seen in Alg. 1 (line 13) during computation of PageRank a vertex changes its state to inactive when the maximum number of supersteps is reached. Otherwise it will continue to update its value and participate in computation. On the other hand, as shown in Alg. 2 an SSSP vertex will vote to halt whenever its value does not update to a newer one (line 10). Otherwise it will update its value with the new shortest distance from the source vertex.

In other words, during the execution of a globally active graph algorithm all the graph vertices are active and will send and receive messages during each superstep. Thus, the range of active vertices is global with respect to graph structure. Whereas, during the execution of locally active graph algorithms, only local regions of graph vertices are in active state and participate in the computation. In these algorithms,

Algorithm 1 Vertex-Centric Implementation of PageRank

```
1:  function COMPUTEPR(msgs,superstep)
2:      if $vrtx\ has\ no\ ngbr$ then
3:          $Vote\ to\ Halt$
4:      end if
5:      if $superstep \geq 1$ then
6:          $sum \leftarrow 0$
7:          for $msg\ in\ msgs$ do
8:              $sum \leftarrow sum + msg.val$
9:          end for
10:         $numVrts = len(ngbrs)$
11:         $prVal \leftarrow PR(sum, nmVrts)$
12:         $sendMsg(ngbrs, numVrts)$
13:         if $superstep = maxSuperStep$ then
14:             $Vote\ to\ Halt$
15:         end if
16:     end if
17: end function
```

Algorithm 2 Vertex-Centric Implementation of SSSP

```
function ComputeSSSP(msgs,superstep)
    if IsSource(vID) then
        $minDis \leftarrow 0$
    else
        $minDis \leftarrow \infty$
    end if
    for $msg\ in\ msgs$ do
        $mindDis\ =\ min(mindDis\ ,\ msg.val)$
    end for
    if $minDis \leq vrtxVal$ then
        $vrtxVal \leftarrow minDis$
        $sendMsg(ngbrs, vrtxVal)$
    else
        $Vote\ to\ Halt$
    end if
end function
```

TABLE I: Categorization of Graph Algorithms

Globally Active	Locally Active
PageRank	SSSP
HITS	Minimal Spanning Tree
Bipartite Matching	Graph Isomorphism

computation progress throughout different local parts of graph structure as the computation proceeds.

Table I shows the categorization of some other graph algorithms based on the above localization properties (globally active vs locally active) of the graph algorithms. In next section, we have chosen PageRank and SSSP as representatives of each group and perform these computations in vertex-centric paradigm on different real graph datasets with two different graph partitioning strategies.

IV. Experiment and Evaluation

A. Performance Metrics

The existence of the cost model that is both tractable and accurate makes it possible to analyze efficiency of graph algorithms when the vertex-centric programming model is utilized. As shown in eq. 4, the following strategies should be considered in order to achieve high efficiency for a superstep time:

- balance the computation in each superstep among processors because of two reasons. First, the maximum number of active vertices and number of messages to process is considered among processors. Second, in the barrier synchronization phase, processors must wait for the slowest processor to complete its computation.
- balance the communication among processor nodes since the maximum of received bytes and sent bytes of data is taken among processor nodes.

In order to measure the load balance of a superstep in terms of both of the factors (number of active vertices and number of messages to receive) that affect the computation phase of a superstep, we define the following metrics. For the first factor we define the average of standard deviations of number of active vertices over all supersteps to be the first metric to measure the performance of load balance as follows:

$$\mathit{Load\_Balance_{av}(N,K,A)} = \frac{\sum_{k=1}^{K}\left(\sqrt{\frac{\sum_{i=1}^{N}(AV(P_{ik})-\mu)^2}{N}}\right)}{K} \quad (5)$$

where *Load_Balance*$_{av}$*(N,K,A)* is the load balance of *N* processor nodes during *K* supersteps when running graph algorithm A (in terms of number of *active vertices*), *AV(P*$_{ik}$*)* is the number of active vertices for processor i at superstep k and μ is the average number of active vertices on processor nodes in superstep k. Similarly for the second factor of load balance we define the following metric:

$$\mathit{Load\_Balance_{rm}(N,K,A)} = \frac{\sum_{k=1}^{K}\left(\sqrt{\frac{\sum_{i=1}^{N}(RM(P_{ik})-\mu)^2}{N}}\right)}{K} \quad (6)$$

where *Load_Balance*$_{rm}$*(N,K,A)* is the load balance of *N* processor nodes during *K* supersteps when running graph algorithm A (in terms of number of *received messages*), *rm(P*$_{ik}$*)* is the number of received messages for processor i at superstep k and μ is the average number of received messages by all processor nodes in superstep k. In order to measure the cost of the communication phase among *N* processor nodes during *K* supersteps when running graph algorithm *A*, we define the following metric which is the sum of the averages of sent and received bytes.

$$\mathit{Comm\_Cost(N,K,A)} = \sum_{k=1}^{K}\left(\frac{\sum_{i=1}^{N}(\mathit{SentBytes}(P_{ik})}{N}\right) + \sum_{k=1}^{K}\left(\frac{\sum_{i=1}^{N}(\mathit{ReceivedBytes}(P_{ik})}{N}\right) \quad (7)$$

B. Experiments Setup

In order to measure the performance of vertex-centric graph algorithms in terms of the above metrics, we have performed extensive experiments on different data sets. The graph data set information are shown at table II and are obtained by using Webgraph software [11].

We have used Amazon EC2 instances in order to set up our clusters with different sizes. We performed our experiments on clusters with 2, 4 and 8 nodes in order to investigate the effects of load balancing and communication cost on clusters with different sizes. Each node in our cluster is an Amazon's "m3.medium" instance type with single core high frequency Intel Xeon E5-2670 v2 cpu, 3.75 GB of memory running Ubuntu Server 14.04. The performance of our communication network among processor nodes is set at the moderate level. We also have used GPS to implement vertex-centric implementation of PageRank and SSSP graph algorithms as provided in Algorithms 1 and 2.

As the baseline for our experiments, first we examine the random partitioning of graph vertices on processor nodes. In

TABLE II: Graph Information

Name of Graph	Vertices	Edges	Description
in-2004	1,382,908	16,917,053	the .in domain of WWW graph
itwiki-2013	1,016,867	25,619,926	the italian part of Wikipedia as late as Feb 2013
ljournal-2008	5,363,260	79,023,142	LiveJournal virtual community social site

this scenario, the graph vertices are randomly placed on the processor nodes of clusters and no structural properties of the graph are examined for the placement of the graph vertices on cluster nodes. In our experiments we refer to this approach as *RND*. We also used Metis [12] graph partitioning software to partition the graphs and then distribute the graph vertices among processor nodes based on the results of Metis graph partitoning software. Metis uses multilevel k-way partitioning algorithms to compute the min-cut of a graph. Metis' heuristics based algorithms try to find the best partitions where the number of inter partitions (cuts) are minimum while each partition holds the same number of vertices. The idea here is that by performing min-cut the communication cost of the cluster will be lowered. We partitioned the graph so that each partition has the same number of vertices while the number of cross cluster edges are minimum. Then we assigned each graph partition to a processor node. As we will see in section 5, however, lowering communication cost using Metis will result in lower load balance for locally active graph algorithms. We referred to this graph partitioning scheme as *MTS* in our experiments.

C. Results

1) Load Balancing: In terms of the first performance metric for load balancing ($\textit{Load Balance}_{av}$) as shown in Fig. 3, when the graphs are partitioned randomly the load is evenly distributed among processor nodes. However, when the graph vertices are distributed among processors based on graph partitioning scheme (Metis) the load of processors is very unbalanced as it is shown with high values for average of standard deviation for active vertices (see Eq. 5 for definition of this metric). Fig. 3 depicts the results for this metric for both PageRank and SSSP algorithms. As shown in this figure, the load is evenly distributed if the graph vertices are randomly distributed among processor nodes compared to the case when vertices are assigned to processors based on the results of Metis. It is also noticeable that as the number of processor nodes in the cluster increases from 2 to 8 the load imbalance decreases to almost half (for instance from 7349 to 4653 in the case of in-2004 data set and from 254433 to 12834 in the case of itwiki-2013 data set) when PageRank algorithm is computed. Consequently, a possible solution for having a more balanced load among processor nodes when running a globally active graph algorithm such as PageRank is to add more nodes to the cluster.

In the case of the SSSP algorithm the load balance worsens as the number of processor nodes increase from 2 to 8 (for in-2004 and itwiki-2013 datas ets). This is because the SSSP algorithm is a locally active graph algorithm where computation starts at some region of the graph and it propagates to other regions of the graph until it terminates. Hence, the load balancing of this graph algorithm in terms of active vertices is sensitive to the starting vertex (source vertex) of the computation. For ljournal-2008, however, this is not true since the starting vertex for this graph is in a very dense part of graph where by adding more nodes to the cluster the load on processor nodes becomes more balanced and hence the average of standard deviation across supersteps gets lower. Similarly to circumstances for PageRank, random partitioning always outperforms the assignment of graph vertices to processor nodes based on Metis in terms of average standard deviation of active vertices across supersteps.

The results for the second performance metric of the load balance ($\textit{Load Balance}_{rm}$) is depicted in Fig. 4. Similar to the first metric for the load balance, randomly assigning graph vertices to the processor nodes leads to a lower average of standard deviations across supersteps for number of received messages by processors compared to when a graph partitioning scheme such as Metis is used. This is because of the correlation between the number of active vertices and received messages and the fact that the graph vertex changes its state from inactive to active upon receiving messages. This also testifies again that when graph partitioning mechanisms such as Metis are used the load among processors is distributed unevenly in terms of the active vertices that processors have to handle in order to perform the computation.

In conclusion, as it is shown both in Fig. 3 and Fig. 4 the random assignment of graph vertices to processor of clusters leads to better load distribution among processor nodes of a cluster compared to when a graph partitioning scheme such as Metis is used. However, as we will see in the next section, using Metis has advantages in terms of communication cost of the cluster when the nature of the graph algorithm requires high communication among processors. We also notice that one way to have better load balancing in a cluster is to add more nodes to the cluster.

2) Communication: For the performance metric related to the communication cost, *Comm_Cost* (see Eq. 7), utilizing graph partitioning solutions such as Metis can be beneficial as illustrated in Fig. 5. The communication cost is in MB(s) and as is shown when the number of nodes in the cluster increases, the communication cost increases too. This situation is worse for a graph algorithm such as PageRank which is communication intensive. However, when Metis is used this cost drops down significantly. This is due to the fact that by using graph partitioning the vertices that are highly connected and formed into dense clusters are grouped together and assigned to the same processor node. Hence, the processor node does not need to send a message across the cluster to another node as it holds the neighboring vertices.

In the case of the SSSP algorithm the benefits of using graph partitioning solutions such as Metis is not very significant since the graph algorithm is not very intensive in terms of communication volume. As mentioned before, the computation of SSSP algorithm starts at a source vertex and the communication among graph vertices initiates at a local region of the graph that contains the source vertex. The communication among vertices then propagates throughout

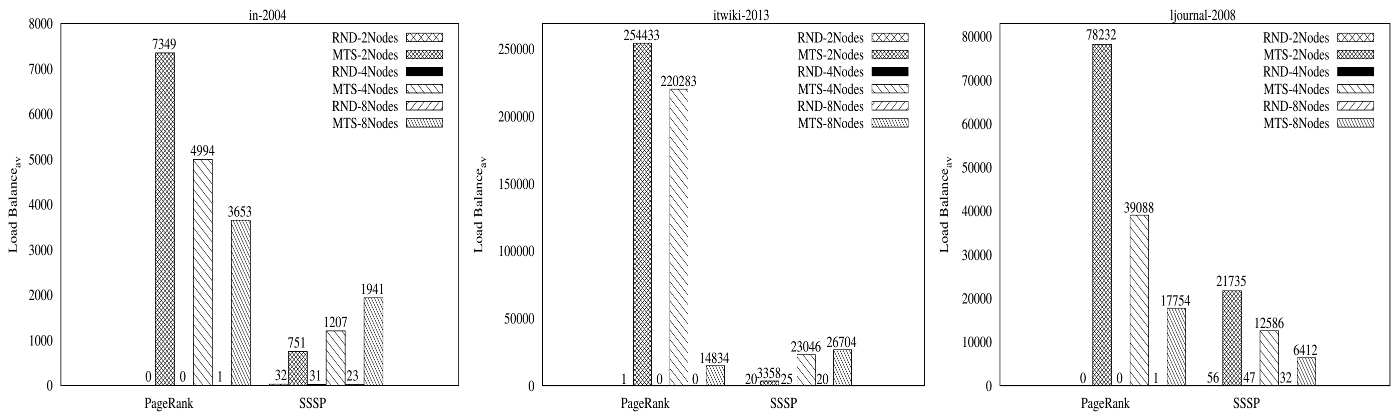

Fig. 3: *Load Balance$_{av}$* performance metric for in-2004, itwiki-2013 and ljournal-2008 data sets

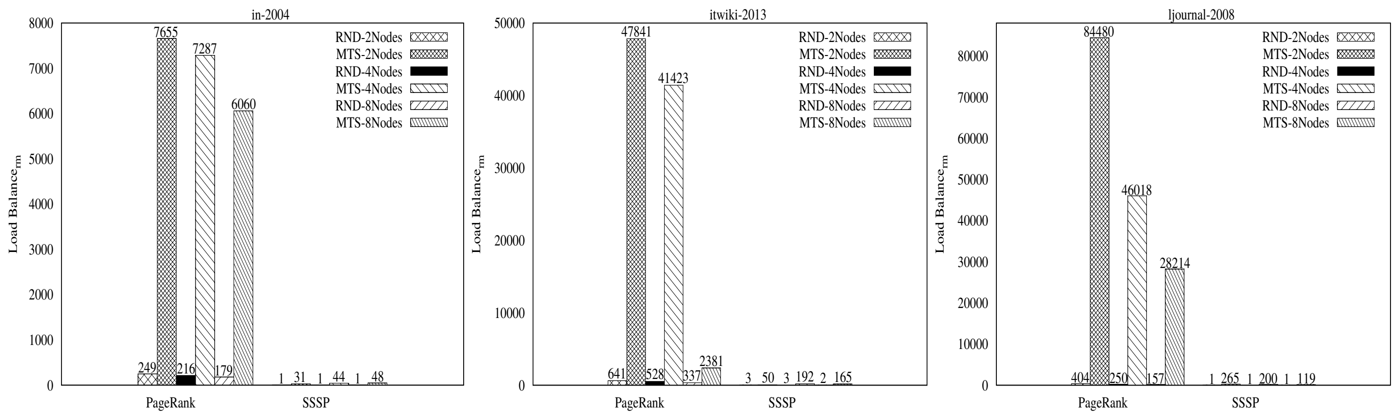

Fig. 4: *Load Balance$_{rm}$* performance metric for in-2004, itwiki-2013 and ljournal-2008 data sets

the graph structure until all the graph vertices update their value (shortest distance from source). This marks the end of SSSP computation. Compared to the PageRank algorithm (which is globally active), SSSP computation involves less communication.

In conclusion, as it is shown in Fig. 5, when the communication volume of the graph algorithm in terms of bytes the network has to deliver to processor nodes is very high, the application of the graph partitioning solution is beneficial to the communication cost of the network. However, this solution leads to load imbalance as we saw previously in Fig. 3 and Fig. 4.

3) Time: Fig. 6 shows the total time of completion for two graph algorithms for clusters with different sizes when random partitioning and Metis based partitioning is used. The charts in this figure reveal several findings as we describe below. First, for PageRank computation of all data sets, as the number of nodes in the cluster increases total time of completion decreases and execution completes faster. This is because of better load balancing both in terms of number of active vertices and received messages (as have been shown previously in Fig. 3 and Fig. 4).

Second, when the graph algorithm has a high volume of communication (e.g. PageRank) using Metis in order to decrease the communication cost leads to significant benefit and faster execution time. This reveals the fact that in the case of the PageRank computation for these three data sets communication among processor nodes is the dominant factor in determining the ultimate cost of execution.

Third, for the computation of the SSSP algorithm as shown in the right parts of the charts, it is evident that when Metis based partitioning of graph vertices to the processor nodes is applied the total time of completion is longer compared to random partitioning. In fact, utilizing graph partitioning has a negative effect on the performance of the system. The explanation for this phenomenon is depicted in the results of load balancing as well as communication cost for this graph partitioning scenario. As shown in Fig. 5, the achieved benefit in terms of lowering the communication volume is insignificant when Metis is used for the SSSP graph algorithm. On the other hand, the load imbalance in terms of both number of active vertices and received messages is high for SSSP when Metis is used (Fig. 3 and Fig. 4). These two factors (low benefit of graph partitioning and high load imbalance) leads to higher completion time for all data sets.

Finally, it is noticeable in Fig. 6 that the time of completion for the SSSP also decreases as more nodes are added to the cluster. This reemphasizes that the number of active vertices and received messages by the processors (load) is the dominant factor in determining the total run time of the SSSP.

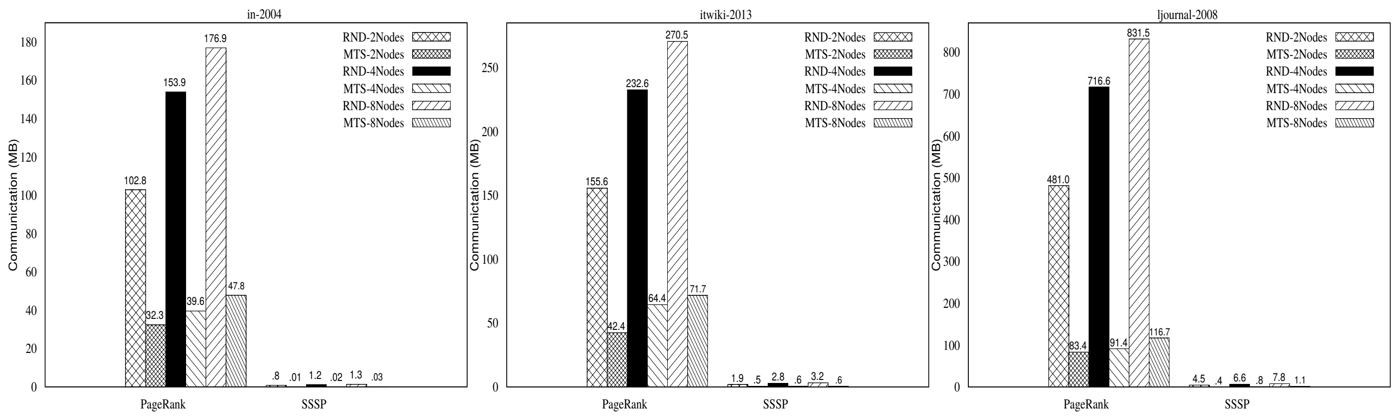

Fig. 5: *Communication Cost* performance metric for in-2004, itwiki-2013 and ljournal-2008 data sets

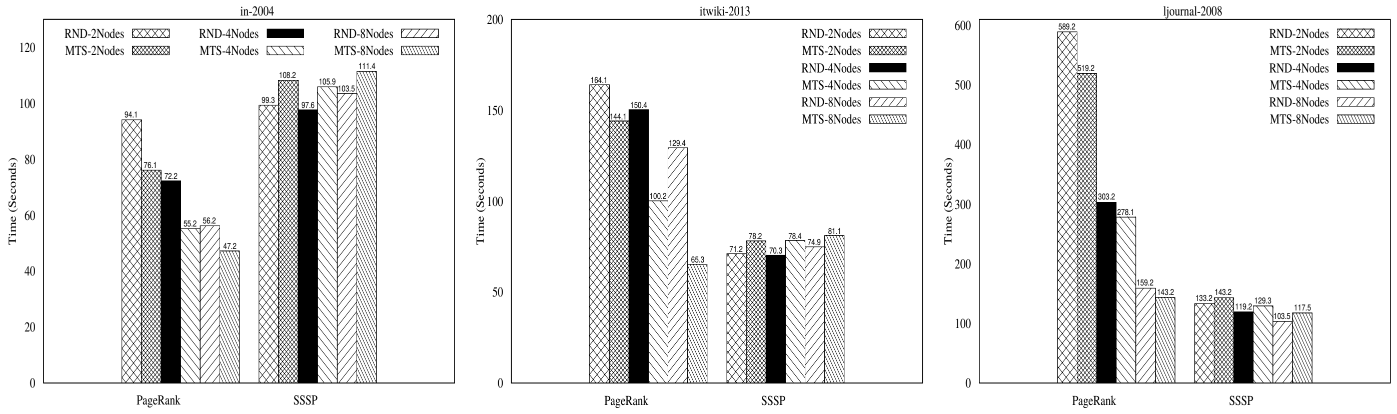

Fig. 6: *Time* performance metric for in-2004, itwiki-2013 and ljournal-2008 data sets

V. Related Work

Bulk Synchronous Parallelism (BSP) was first introduced at [5] as a computational model for general purpose parallelism. Its fundamental properties are being able to write simple parallel programs that are independent of target architecture and also having predictable performance on a given architecture. Pregel [1] was the first system to extend this computational model to graph processing and is a proprietary product of Google. In this system, efficient, scalable and fault-tolerant implementation of the BSP model are utilized on clusters of thousands of nodes. They have introduced a simple API that facilitates writing vertex-centric graph algorithms that can be used on their cluster.

Apache Giraph [2] is an open source counter part of the Pregel that is in use at Facebook to analyze their social graph data formed by its users and their interactions. Compared to basic Pregel, Giraph has several additional features such as master computation and out of core computation.

GraphLab [3] is another parallel computation abstraction that is developed by Carnegie Mellon University and is tailored for machine learning tasks. Their computational model is different than BSP as they use asynchronous message passing based graph parallel computational model in order to achieve a high degree of parallel performance in their machine learning tasks.

To the best of our knowledge, our work is the first to provide mathematical and formal specification of the cost of a superstep in terms of the basic graph structures, as well as to provide definitions of metrics for measuring different aspects of performance on several large data sets. Moreover, we have used two graph partitioning mechanisms to show the effects and interactions among the factors that affect the costs of vertex-centric implementation of graph algorithms.

VI. Conclusion

In recent years, vertex-centric parallel graph processing frameworks such as Pregel [1], Giraph [2] and GPS [4] have acquired significant popularity. However, there is a lack of analytical models to study the performance of these frameworks. In this paper, we have identified the factors that affect the performance of distributed vertex-centric graph processing clusters. We have also presented a formal model to analyze the performance of various graph partitioning strategies. We also provided a categorization of graph algorithms based on their load distribution and communication behaviors. With extensive experiments on massive real world graph data sets we have validated our performance model.

VII. Acknowledgement

This research has been partially funded by the National Science Foundation under Grant Number CNS-1338276. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the NSF.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[2] (2014) Apache giraph. [Online]. Available: http://giraph.apache.org

[3] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[4] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 22.

[5] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, *HADI: Fast diameter estimation and mining in massive graphs with Hadoop*. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2008.

[8] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 229–238.

[9] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[10] E. W. Dijkstra, "A note on two problems in connexion with graphs," *NUMERISCHE MATHEMATIK*, vol. 1, no. 1, pp. 269–271, 1959.

[11] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[12] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *Siam Review*, vol. 41, no. 2, pp. 278–300, 1999.