LogGP: A Log-based Dynamic Graph Partitioning Method

Ning Xu †, Lei Chen ‡, Bin Cui †
†Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University, China
‡Hong Kong University of Science and Technology, Hong Kong, China
†{ning.xu, bin.cui}@pku.edu.cn, ‡leichen@cse.ust.hk

ABSTRACT

With the increasing availability and scale of graph data from Web 2.0, graph partitioning becomes one of efficient preprocessing techniques to balance the computing workload. Since the cost of partitioning the entire graph is strictly prohibitive, there are some recent tentative works towards streaming graph partitioning which can run faster, be easily paralleled, and be incrementally updated. Unfortunately, the experiments show that the running time of each partitioning is still unbalanced due to the variation of workload access pattens during the supersteps. In addition, the one-pass streaming partitioning result is not always satisfactory for the algorithms' local view of the graph.

In this paper, we present LogGP, a log-based graph partitioning system that records, analyzes and reuses the historical statistical information to refine the partitioning result. LogGP can be used as a middle-ware and deployed to many state-of-the-art paralleled graph processing systems easily. LogGP utilizes the historical partitioning results to generate a hyper-graph and uses a novel hyper-graph streaming partitioning approach to generate a better initial streaming graph partitioning result. During the execution, the system uses running logs to optimize graph partitioning which prevents performance degradation. Moreover, LogGP can dynamically repartition the massive graphs in accordance with the structural changes. Extensive experiments conducted on a moderate size of computing cluster with real-world graph datasets demonstrate the superiority of our approach against the state-of-the-art solutions.

1. INTRODUCTION

Data partitioning has been studied for decades. Recently, with the scale of data from Internet becoming larger, data partitioning, especially graph data partitioning has attracted more and more attention. The unprecedented proliferation of data from web requires efficient processing methods to handle different workloads. Many parallelism frameworks have been proposed to process large-scale graphs, e.g., Pregel, GraphLab and PowerGraph [17, 15, 11]. As other distributed systems, graph data partitioning is a key technology to scale out computational capabilities.

Pregel [17], as one of the representative systems, developed by Google, is based on the BSP (bulk-synchronous parallel) model and adopts a vertex-centric concept in which each vertex executes a user-defined function (UDF) in a sequence of **supersteps**. By default, Pregel uses hash function to distribute the vertices. Although hash partitioning generates a well-balanced number of vertices across distributed computing nodes, many messages have to be sent across the nodes for updating which generates a huge communication cost.

Thus, some partitioning methods on Pregel-like systems were proposed, most of which are based on *k-balanced graph partitioning* [10]. K-balanced graph partitioning aims to minimize the total communication cost between computing nodes and balance the vertices on each partition. Andreev et al. [6] proved k-balanced graph partitioning is NP-Hard. Several approximation algorithms and multi-level heuristic algorithms have been proposed. However, as the size of graph becomes larger, they all suffer from a significant increase of the partitioning time. As shown in [25], multi-level approach requires more than 8.5 hours to partition a graph from Twitter with approximately 1.5 billion edges which is sometimes longer than the time spent on processing the workload. Therefore recently, some works focused on much simpler streaming heuristics to get comparable result performance to multi-level ones with much shorter partitioning time [23, 25]. Although the partitioning result balances the number of vertices among each node and reduces the communication cost, for many graph workloads in which not all vertices in every superstep will run the UDF, the streaming or even multi-level graph partitioning algorithms still encounter the skewed running time for some workloads.

To deal with this issue, several approaches were recently proposed: Yang et al. [27] proposed a dynamic replication based partitioning with adaption to workload change. Shang et al. [21] investigated several graph algorithms and proposed simple yet effective policies that can achieve dynamic workload balance. However, these approaches need to repartition the graph again whenever a new workload runs on it and there is no improvement on the partitioning results. In fact, the running statistics or historical partitioning logs can provide us useful information to refine the partitioning result. In this paper, we investigate how to record, analyze and reuse these statistics and logs to refine the graph partitioning result. We settle the issue of the imbalance of running time and reduce the job running time by refining the graph partitioning quality. We develop a novel graph partitioning management method - LogGP that reuses the previous and running statistical information to refine the partitioning. LogGP has two novel log-based graph partitioning techniques. LogGP first combines the graph and historical partitioning results to generate a hyper graph and uses a streaming based hyper graph approach to get a better initial partitioning result. When the workload is executing, LogGP then uses running statistics to estimate the running time of each node and reassigns the workload for the next superstep to reduce the superstep running time. To estimate the running time of superstep, an innovative technique to profile workload and graph is proposed. LogGP can be used as a middle-ware and deployed to Pregel-like systems easily.

We implement LogGP on Giraph - an open source version of Pregel. The performance of LogGP is validated with several large graph datasets on different workloads. The experimental results show that the proposed graph partitioning approaches significantly outperform existing streaming approaches and demonstrate superior scaling properties.

Our contributions in this paper can be summarized as follows:

1. We identify an important running time imbalance problem in large-scale graph processing system.

2. We design and implement LogGP to reuse the previous and running statistics information for partitioning refinement and propose Hyper Graph Repartitioning and Superstep Repartitioning techniques.

3. We conduct extensive experimental study to exhibit the advantages of our approach.

The remaining of this paper is organized as follows. In Section 2, we review the problem of graph partitioning and relevant performance issues. In Section 3 and 4, we present the novel Hyper Graph Repartitioning and Superstep Repartitioning techniques, followed by the architecture of LogGP in Section 5. Section 6 reports the findings of an extensive experimental study. Finally, we introduce the related work and conclude this paper in Section 7 and 8.

2. BACKGROUND

In this section, we first introduce the Pregel system on which our prototype system is built. We next introduce the problem of graph partitioning and streaming graph partitioning. Finally, we analyze the graph partitioning problem in Pregel.

Pregel is a distributed graph processing system proposed by Google, based on the BSP (bulk-synchronous parallel) model [26]. In the BSP model, the graph processing job is computed via a number of **supersteps** separated by synchronization barriers. In each superstep, every worker, or called node, executes a user-defined function against a subset of the vertices on it in an asynchronous computing way. These vertices are called **active vertices**. The rest of vertices, which are not used in a superstep, are called **inactive vertices**. Besides, the node sends the necessary messages to its neighbors for the following superstep. Once the communication and computation are finished, there is a global synchronization barrier to guarantee that all the nodes are ready for next superstep or the assigned job is finished. Same as the other distributed systems, to get best performance, workload should be assigned to each node equally while minimizing the communication cost. Thus graph partitioning is an essential technology to get high scalability.

Graph Partitioning: We now formally describe the general graph partitioning problem. We use $G = (V, E)$ to represent the graph to be partitioned. V is a set of vertices, and E is a set of edges in the graph. The graph may be either directed or undirected. Let $P_k = \{V_1,...,V_k\}$ be a set of k subsets of V. P_k is said to be a partition of G if: $V_i \neq \emptyset$, $V_i \cap V_j = \emptyset$, and $\cup V_i = V$, for i, j = 1, ..., k, $i \neq j$. We call the elements V_i of P_k the parts of the partition. The number k is called the cardinality of the partition. In this paper, we assume that each V_i is assigned to one computing node, and use V_i denote the set of vertices in that node.

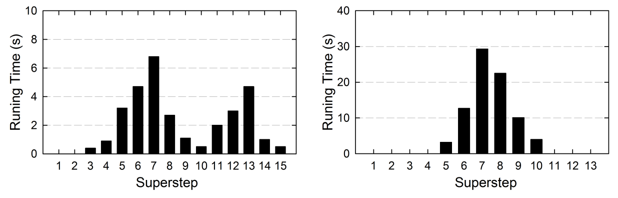
Graph partitioning problem is to find an optimal partition P_k based on an objective function. It is a combinatorial optimization problem that can be defined as follows:

Definition 1. *Graph partitioning problem can be defined from a triplet (S, p, f) such that: S is a discrete set of all the partitions of G. p is a predicate on S which creates a subset of S called admissible solution set - S_p that all the partitions in S_p is admissible for predicate p. f is the objective function. Graph partitioning problem aims to find a partition $\bar{P}$ that $\bar{P} \in S_p$ and minimizes $f(p)$:*

$$f(\bar{P}) = min_{P \in S_p} f(P) \tag{1}$$

A simple policy is to partition the data with a hash function, which is the default strategy applied in Pregel [17]. However, this approach results in high communication cost thus, degrades the performance. Some works use approximation or multi-level approaches to partition the graph. However as the graph becomes larger, the cost of partitioning is unacceptable [25]. Thus, some recent works [23, 25] use streaming partitioning heuristics to partition the graph.

In particular, if the vertices of graph arrive in some order with the set of its neighbors, and we partition the graph based on the vertex stream, it is called a **Streaming Graph Partitioning Algorithm**. Streaming graph partitioning algorithm decides which part to assign for each incoming vertex. Once the vertex is placed, it will not be removed.



a. SSSP b. BFS

Figure 1: Running Time of Supersteps on a Node

These k-balanced graph partitioning approaches balance the vertex computation job while the communication time of each part is not considered. Thus, the running time, which contains both computation and communication time, of each node may be skew for each node. A reason for the imbalance is the Traversal-Style workload [21], e.g., SSSP and BFS. These workloads explore different active vertices on each node at different supersteps which is hard to predict at the initial partitioning stage. Thus the computation time and communication time of each node on specific supersteps are imbalanced. Figure 1 shows the running time of a certain node for workloads SSSP and BFS with a multi-level graph partitioning algorithm for initial partitions. We can see that there is a significant running time imbalance for each superstep. For the Always-Active workloads [21], there is imbalance of running time as well, because the k-balanced graph partitioning only balances the vertex computation job while the communication time of each part is not considered.

The running imbalance of node for Pregel-like system affects the overall job running time for graph processing workload and limits the scale out of the whole system. Thus, using static initial partitioning result which is generated without analyzing the workload behavior cannot balance the running time of each node.

To alleviate the issues mentioned above, we propose a graph partitioning scheme LogGP which exploits historical graph information. LogGP uses two novel techniques, i.e., Hyper Graph Repartitioning and Superstep Repartitioning, which will be introduced in next two sections.

3. HYPER GRAPH REPARTITIONING

We first introduce how to generate better initial partitioning result with the help of historical log. As mentioned in Section 1, multi-level partitioning algorithms are too slow for large graphs. Thus, we use streaming approach to generate initial partitioning result.

Let $P^t = V_1^t, ..., V_k^t$ be a partitioning result at time t, where V_i^t is the set of vertices in partition i at time t. A streaming graph partitioning is sequentially presented a vertex v and its neighbors $N(v)$, and uses a streaming heuristic to assign v to a partition i only utilizing the information contained in the current partitioning P^t. Although one-pass streaming partitioning algorithm has shorter partitioning time, the partitioning result is not as good as multi-level solutions because it only has the information of the assigned vertices in the graph. In this paper, we use historical partitioning log to enlarge the vision when running streaming graph partitioning and refine the initial partitioning result.

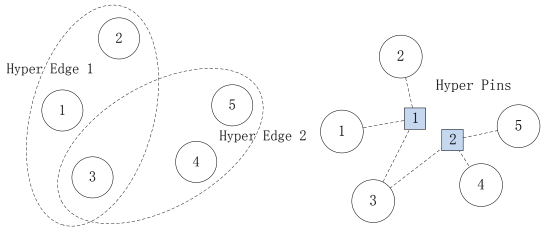
We use two kinds of useful information that are provided by the historical partitioning result.

The first one is the last partitioning result of the same graph or some parts of the graph. This result has divided the graph into several partitions generated by the streaming partitioning algorithm. Although not optimized, it provides us a better initial input than random graph. We try to appropriately use this last partitioning result to provide the streaming graph partitioning more information than that of the assigned vertices contained in the current partitioning.

The other information is the accessed active vertex set during the execution of previous workloads which is called **Log Active Vertex Set(LAVS)**. A LAVS is an active vertex set that a node accessed in continuous supersteps. We find that the active vertices in successive supersteps have some internal relationship with each other especially when the accessed vertices of the workload have natural connection, e.g., Semi-clustering [17], Maximal Independent Sets [16] and N-hop Friends List [7]. These workloads naturally gather the vertices with some kinds of internal relationships together. Take Semi-clustering as an example, a semi-cluster in a social graph is a group of people who interact frequently with each other and less frequently with the rest. When running Semi-clustering on a graph, the LAVS of this workload accesses the vertices (stand for people), who have strong connections between each other. Thus, if we use the LAVS as the additional information for graph partitioning, we will initially know this connection of the graph which can help the streaming algorithm determine which vertices should be placed together. To control the number of vertices in a LAVS for different workloads and graphs, we use a parameter k to determine how many continuous supersteps a LAVS will be concerned. The detail of how to log and compute LAVS will be discussed in Section 5.

3.1 Hyper Graph

To combine these two kinds of historical information with the original graph, we use **hyper graph** to represent the useful historical information and the original graph structure. A hypergraph $H = (V_h, E_h)$ is a generalization of a graph whose edges can connect more than two vertices called hyper edges (also named nets). A hyper graph consists of a vertex set V_h and a set of hyper edges E_h . Each hyper edge is a subset of V. For example, as shown in Figure 2(a), there are two hyper edges presented with dotted circles. The first one (**hyper edge 1**) contains vertices V_1, V_2 and V_3, and the other one (**hyper edge 2**) contains vertices V_3, V_4, and V_5. Another way to present hyper edge is converting the hyper edge to equivalent **hyper edge labels** and **hyper pins**, as shown in Figure 2(b). The hyper pin and hyper edge label can be treated as virtual vertex and virtual edge of the plain graph. The cardinality $|e_h|$ of a hyper edge e_h is the number of vertices that it contains. A hyper graph H is k-regular if every hyper edge has cardinality k. In fact, the plain graph is a 2-regular hyper graph. Hyper graph naturally represents the three layouts of our information: the original graph, the last partitioning result and the LAVS. We next proceed to introduce how to form it.



a. Hyper Edge b. Hyper Pin

Figure 2: Two Ways to Represent Hyper Graph

How to form the hyper graph: Given the original graph $G = (V_o, E_o)$, we use the same vertex set V_o to form the hyper graph vertices that $V_h = V_o$.

The edge set of the hyper graph E_h consists of 3 parts:

1. We use v_s and v_e to denote the two vertices of an original graph edge in the edge set E_o. Then we add a hyper edge $h_e = \{v_s, v_e\}$ to E_h.

2. We use $P_{last} = (P_{last}^1, ..., P_{last}^n)$ to denote the last partitioning result of the graph. P_{last}^i denotes the i-th parts of the partitioning result. We add hyper edge $h_e = P_{last}^n$ to E_h.

3. We use $LAVS_i = LAVS_i^1, ..., LAVS_m^i$ to denote the LAVS sets of node i with total number of m, and we add each hyper edge $h_e = LAVS_m^i$ to E_h.

Figure 3 shows an example of how to generate a hyper graph. We first transform the original graph (Figure 3a) to hyper graph edges (Figure 3b). These hyper edges $h_e = \{v_s, v_e\}$ are marked as hyper pin "1". Then we include the latest partitioning result of the graph (Figure 3c). These hyper edges $P_{last} = (P_{last}^1, ..., P_{last}^n)$ are marked as hyper

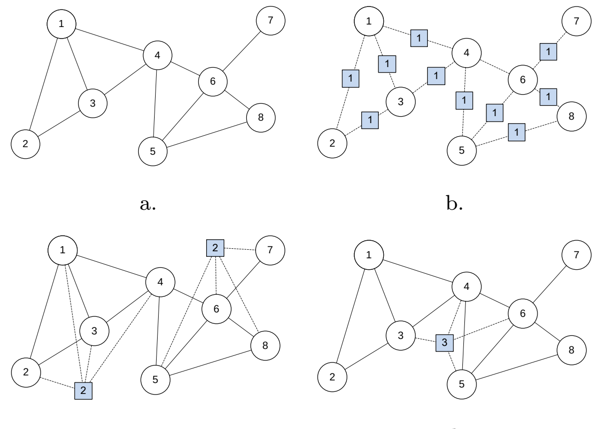


Figure 3: Example of Hyper Graph Generation

pin "2". At last, we add the LAVS to get the final hyper graph (Figure 3d) which is marked as hyper pin "3".

After generating the hyper graph, we design a novel hypergraph streaming graph partitioning approach called Hyper Streaming Partitioning(HSP). We firstly convert the hyper edge to equivalent **hyper edge labels** and **hyper pins**. A hyper graph $H = (V, E_h)$ can also be defined as a triplet $H{=}(V, \bar{H}, \bar{P})$, where V is the set of original vertices, $\bar{H}$ is the set of hyper edge labels, and $\bar{P}$ represents the set of edges connecting an element of V to an element of $\bar{H}$. As shown in Figure 2, we can use hyper edge labels and hyper pins (Figure 2(b)) to represent hyper edges (Figure 2(a)). After such conversion, we can now treat the hyper edges as vertices (hyper pins) and use them for one-pass streaming graph partitioning. Note that we only generate hyper pin for the hyper edge whose cardinality is more then 2. Thus the original edge is not represented as hyper pin. In this way, the hyper pins and hyper edge labels provide vertex connection information of both previous partitioning result and active vertex logs.

3.2 Hyper Graph Repartitioning

Our hyper graph repartitioning approach consists of two parts. We first assign the hyper pins to different partitions, and then assign the original vertices to these partitions. The hyper edge label is treated as an edge which will be used by streaming partitioning algorithms as an additional edge. When a hyper pin is assigned to a partition, the vertices in the same hyper edge would have a higher probability to be assigned to that partition. The hyper pins, in fact, provide the streaming algorithms some sort of global vision of the graph and the hyper edge can present some kinds of internal connections between the vertices.

The streaming graph partitioning algorithms make decisions based on incomplete information, and the order in which data is streamed will significantly affect the performance [23]. The order of vertices varies in several ways, i.e., random, BFS/DFS from a starting vertex or adversarial order. Isabelle [22] proved that the BFS and DFS produce nearly the same result as the random way. Of these orders, a random ordering is the simplest to guarantee in large-scale streaming data scenarios, and so we restrict our hyper graph repartitioning to only consider random node orders for simplicity. Thus we randomly assign hyper pins followed by the original vertex assignment.

We use a streaming loader to read vertices from hyper graph. The loader then sends vertices with their neighbors to the partitioning program which executes streaming graph partitioning as shown in Algorithm 1. The heuristic function determines the assignment of incoming vertices based on the current Partition state, vertex information and input graph.

Algorithm 1 Hyper Graph Repartitioning

```
 1: Input: Hyper Graph $H$
 2: Let $P_k = V_1, V_2, ..., V_k$ be the current partitioning result
    sets.
 3: $S \leftarrow$ streamingloader(H)
 4: generate random order to $S$
 5: for each hyper graph pins $\bar{p}$ in $S$ do
 6:     $index \leftarrow HeuristicFunc(P_k, \bar{p})$;
 7:     Insert hyper graph pins $\bar{p}$ into $V_{index}$;
 8: end for
 9: for each vertex $v$ in $S$ do
10:     $index \leftarrow HeuristicFunc(P_k, v)$;
11:     Insert vertex $v$ into $V_{index}$;
12: end for
```

In [23, 25], some recent works on a broad range of heuristics for performing streaming node assignment were proposed. In this paper, we do not focus on proposing new streaming heuristics. In fact, our approach can be adapted to any of these heuristics. Here we take Linear Deterministic Greedy **(LDG)** as a representative, since LDG has the best performance among these heuristics. In LDG, each vertex v is assigned to the partition based on:

$$index = argmax|V_i \cap N(v)| \left(1 - \frac{|V_i|}{C_i}\right) \qquad (2)$$

where C_i is the maximum capacity of partition i and $N(v)$ is the set of neighbors of vertex v.

As we mentioned above, the graph may be expected to change slightly each time it is used. In fact, graph datasets are typically dynamic, i.e., the graph structure changes over time. For example, the Twitter graph may have millions of vertices and edges changed per second at peak rate. Thus it is important to consider the ability of our system to accommodate incremental graphs. One advantage of hyper graph repartitioning is that it does not require any modification: our approach can repartition the new graph using the historical information, and the new vertices can be assigned by hyper graph repartitioning as the original vertices.

4. SUPERSTEP REPARTITIONING

In this section, we analyze the imbalance of running time on each node during supersteps and propose a novel repartitioning strategy based on collected statistics during the job execution.

4.1 Job Running Time

In order to improve the performance of graph processing job, we first try to investigate and model the job running time of Pregel systems. We will analyze the algorithms and model the influence of each superstep. Apart from dividing the algorithms into different categories [21], we want to find a general way to model these algorithms on graphs.

In the BSP model, a job is divided into a sequence of supersteps as shown in Figure 4. In each superstep, nodes

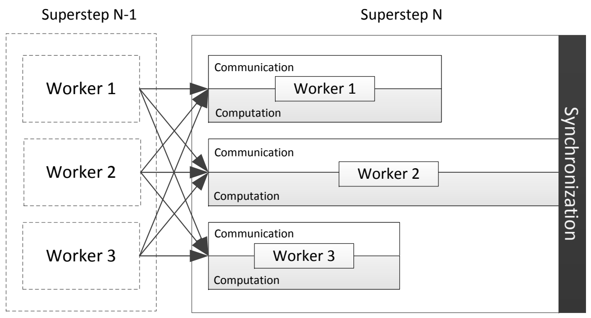


Figure 4: Abstraction of the BSP Model

generally communicate with each other, i.e., send/receive information to/from its neighbors. After all the nodes finish computing and communication jobs, there is a barrier to make sure nodes are ready for next superstep. The execution of a graph algorithm stops when all sets of the active vertices are empty or the maximal number of supersteps is reached.

The running time of each superstep is determined by the slowest node. Let ST_i^n be the time that $Node_i$ spends at the n-th superstep. The time cost of the synchronization barrier is constant for every superstep, and often negligible when compared to the costs of the computation and communication, so we ignore it in this paper. Then, the total running time of the graph algorithm, $JobTime$, can be calculated as:

$$JobTime = \sum(max(ST_i^n)) \tag{3}$$

Graph partitioning technique helps graph processing system to get minimized $JobTime$. As mentioned in Section 2, time spent in a superstep is determined by both communication time and computing time:

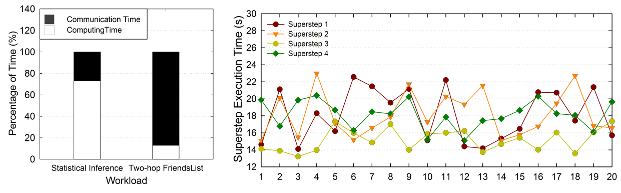
$$ST_i^n = f(Tcomp_i^n, Tcomm_i^n) \tag{4}$$

Here $Tcomp_i^n$ and $Tcomm_i^n$ denote the computing time and communication time used by $Node_i$ at the n-th superstep, respectively. The function $f(x, y)$ is determined by the system implementation. If the system adopts an I/O blocking model in which CPU and I/O are operated serially, we can add up the time which means $f(x, y) = x + y$. If the system parallelly processes I/O operations, the superstep's running time ST_i^n is determined by the slower one, thus $f(x, y) = max(x, y)$. Our system is based on the I/O blocking model, while the same result can be obtained when using a system where I/O is paralleled. In our system, the running time of n-th superstep and $JobTime$ can be presented as:

$$ST^n = Max(Tcomp_i^n + Tcomm_i^n) \tag{5}$$

$$JobTime = \sum Max(Tcomp_i^n + Tcomm_i^n) \tag{6}$$

Since $Tcomp_i^n$ and $Tcomm_i^n$ depend on graph algorithm and hardware, it is hard to get these information in the initial graph partitioning phase. Figure 5(a) shows an experiment on two workloads **Statistical Inference** and **Two-hop Friend List**. We record the average percentage of time used for computing UDF function and sending/receiving data when executing them in Giraph. As we can see, the running time of Statistical Inference is dominated by computing jobs while Two-hop Friend List is dominated by I/O data transmission. The traditional graph partitioning algorithms target at minimizing edge cuts not the running time, and thus ignore these factors. As a consequence, the running time of each node may be skew for a superstep. As shown in Figure 5(b), we record the running time of the first 4 iterations of a Pagerank workload on a 20-nodes cluster. The running time varies significantly.



a. Workloads b. Nodes

Figure 5: Running Time of Superstep on Different Nodes and Workloads

4.2 Repartitioning Heuristic

The traditional database systems use collected historical statistics to estimate running time for query optimization. For parallel graph processing systems, to solve the imbalance problem, we collect useful information to optimize the workload balance between nodes by estimating the near-future running time of each node and reassign the vertices dynamically during algorithm execution. We first discuss how to estimate the running time.

We collect statistics information to estimate the running time of each partition. First of all, we discuss how to estimate superstep running time. As mentioned in Section 4.1, running time of a superstep equals to the running time of computing part plus the running time of communication part.

$$ST_i^n = Tcomp_i^n + Tcomm_i^n \tag{7}$$

The computing time $Tcomp_i^n$ is determined by the whole active vertex set, denoted as A_i^n, and the user-defined function. This is because all the active vertices will execute the UDF. Communication time, however, is caused by parts of the active vertex set. As shown in Figure 6, we divide the vertices in a partition into three types. The dotted lines denote the connections of two vertices in different partitions and the solid lines denote vertices in the same partition.

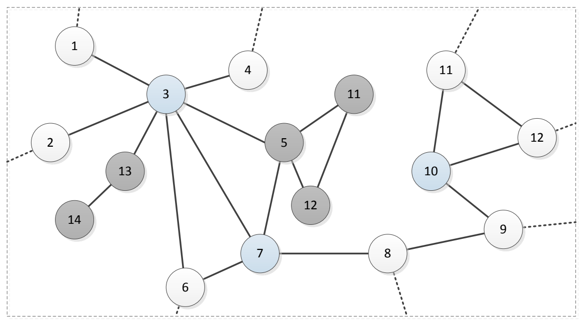


Figure 6: Three Types of Vertices in a Partition

$Type_1$: Vertices and their neighbors are adjacent to vertices in the same partition. For example, vertices 5, 11, 12, 13 and 14 are of this type. We use α_i^n to denote the vertices belonging to this type in n-th superstep on $node_i$.

$Type_2$: Vertices that are adjacent to vertices from other partition. For example, vertices 1, 2 and 4 are of this type. We use β_i^n to denote the vertices belonging to this type in n-th superstep on $node_i$.

$Type_3$: Vertices that are adjacent to only vertices in the same partition while their neighbors are adjacent to vertices from other partitions. We use γ_i^n to denote the vertices belonging to this type in n-th superstep on $node_i$. The vertices 3, 7 and 10 belong to this type.

The first type vertices generate only local messages. In parallel graph processing system, producing and processing local messages are extremely faster than non-local messages, so we ignore the time cost for dealing with local messages. Thus vertices in $Type_1$ do not generate communication time.

The second type vertices send and receive messages from their adjacent partition(s). The total communication time in this superstep is caused by this subset of active vertices. We can now denote the superstep running time as:

$$ST_i^n = tComp * |A_i^n| + tComm * |\beta_i^n| \tag{8}$$

$tComp$ and $tComm$ are the average computing time and communication time used by each vertex. LogGP uses running logs to get these two parameters during the job execution. Detailed information will be discussed in Section 5.

As same as $Type_1$, the vertices of $Type_3$ do not generate communication cost, because there is no non-local vertices adjacent to it. However, these vertices may activate the vertices of $Type_2$ to generate communication cost in the next superstep. Thus we can use these vertices of $Type_3$ to estimate the communication time in the next superstep.

Another problem is that how can we get the number of vertex set β_i^{n+1} with the information of γ_i^n. Here we introduce a new metric, active ratio, denoted as λ. Active ratio denotes the probability of an active vertex v in superstep N that causes the neighbors of that vertex to be active in superstep $N+1$. For example, the active ratio of algorithm Pagerank is always 100% because the active vertex will always activate its neighbors in next superstep. These parameters can be obtained by the historical log as well.

Then, we can calculate $|\beta_i^{n+1}|$ and $|A_i^{n+1}|$ with λ:

$$|\beta_i^{n+1}| = |\gamma_i^n| * \lambda \tag{9}$$

$$|A_i^{n+1}| = |A_i^n| * \lambda \tag{10}$$

The running time of (n+1)-th superstep can be estimated:

$$ST_i^{n+1} = (tComp * |A_i^n| + tComm * |\gamma_i^n|) * \lambda \tag{11}$$

After estimating the running time of the next superstep, we collect the estimated running time of each node, and reassign the vertices to rebalance the running time of the next superstep.

However, vertex reassignment (repartitioning) is still difficult because:

1) Finding the optimal repartitioning is an NP problem, as the size-balanced graph partition problem is NP-complete [6] which is the static setting of our problem.

2) The computational overhead from the repartitioning cost must be low. As the objective is to improve application performance, the selected technique must be lightweight and can be scalable to work on large graphs.

3) Synchronizing distributed states of different partitions dynamically is impossible. Propagating global information across the network incurs a significant overhead, which must be considered for our repartitioning technique. Thus the repartitioning approach can only use a local view of the graph.

The basic idea is to reassign the vertex from partitions with long estimated time to short ones before the next superstep. We use a threshold θ_{n+1} to determine whether the estimated time is long in the total running time.

$$\theta_{n+1} = \vartheta * \frac{\sum ST_i^{n+1}}{i} \tag{12}$$

Here ϑ is the percentage which we will discuss below. We move vertices from the partitions that ST_i^{n+1} are larger than θ_{n+1} to the other partitions and try to minimize the longest running time.

In this paper, we propose a novel heuristic that moves the vertex which will generate communication cost in the next superstep to an appropriate partition. The movement will reduce the total communication cost in the future and balance the running time of each node in next superstep. For a vertex v_i in $node_i$, LogGP logs the times of the vertex communicating to other partitions and denotes it as $C(v_i)$. If a vertex repartitioning is needed on $node_i$, we first choose the vertices that may generate remote communication in the next n+1 superstep, denoted as Γ_{n+1}^i. Let N(v) be the neighbor set of vertex v.

$$\Gamma_{n+1}^i = \bigcap N(v), v \in \gamma_i^n \tag{13}$$

We then sort the remote communication times, $C(v_i)$, for each of the vertex in Γ_{n+1}^i in a non-descending order and greedily reassign the vertex with the max $C(V_t)$ to the partition that the vertex has the most neighbors. When a vertex is removed, the estimated running time will be decreased by $tComp + tComm$. The reassignment stops when Γ_{n+1}^i is empty or the estimated running time of the next superstep is equal to the average. The whole processing is illustrated in Algorithm 2.

Algorithm 2 Superstep Repartitioning

```
1: $ST_i^{n+1} \leftarrow$ estimated running time of $N+1$ superstep
2: if $ST_i^{n+1} > \vartheta * \frac{\sum ST_i^{n+1}}{i}$ then
3:     $\Gamma_{n+1}^i \leftarrow \bigcap N(v), v \in \gamma_i^n$ //$N(v)$ is the neighbor vertex set of v
4:     $List \leftarrow C(v)$, v $\in \Gamma_{n+1}^i$ //$List$ records the $C(v)$ of vertex in $\Gamma_{n+1}^i$
5:     $List \leftarrow$ Sort $List$ in non-descending order
6:     while $\Gamma_{n+1}^i \neq \varnothing$ and $ST_i^{n+1} < \frac{\sum ST_i^{n+1}}{m}$ do
7:         // $m$ is the number of partitions
8:         $v_t \leftarrow$ pop($List$)
9:         Reassign $v_t$ to the partition that the $v_t$ has the most neighbors
10:        $\Gamma_{n+1}^i \leftarrow \Gamma_{n+1}^i - (tComp + tComm)$
11:    end while
12: end if
```

5. THE LogGP ARCHITECTURE

In this section, we present an overview of LogGP to show how the proposed techniques can be seamlessly integrated into the system to improve the performance. We first provide the system architecture, and then present the key components of the system. Finally, we discuss how we manage the vertex migration.

System Architecture: We first describe the basic architecture and operations of LogGP. Figure 7 depicts how LogGP is integrated into Giraph (open source version of Pregel) as a middle-ware. Our system can be easily migrated to any other BSP-based graph processing systems as well. In the Giraph system, there are two types of processing nodes: master node and worker node. There is only one master node which manages to assign and coordinate jobs, and there are multiple worker nodes which execute user defined function against each vertex assigned to them. The components of LogGP are running on the corresponding nodes of Giraph nodes. There is a LogGP partitioning manager (PM) running on the master node that provides partitioning meta data for the master node. A LogGP partitioning agency (PA) is running on each of the worker nodes to record, collect and report log information of that worker node to PM. The initial graph data is partitioned on PM and assigned to each worker.

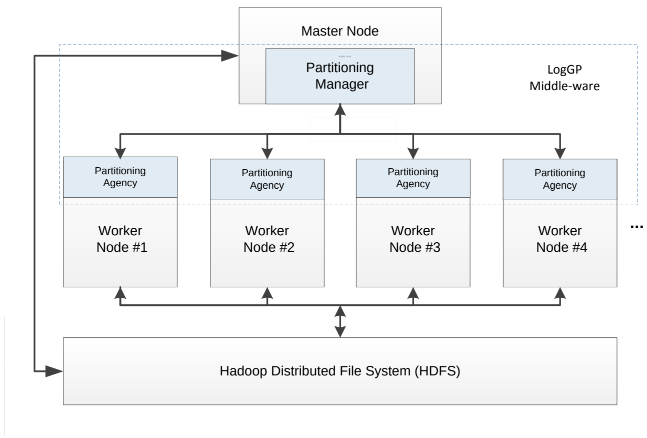


Figure 7: Architecture of LogGP Integrated on Giraph

Both PM and PA share the communication framework of Giraph to communicate with each other. The persistent data, such as historical partitioning result, is stored in the HDFS [4] shared with Giraph as well. The detailed discussion of the key components is as follows:

LogGP Partitioning Manager: Partitioning Manager (PM) is a process running on the master node. It stores the meta data of graphs and historical partitioning results. There is a unique $Graph_{ID}$ for each graph in LogGP , which represents a certain graph. The meta data stores the link of the graph's historical information, such as the latest partitioning result or the LAVS of that graph. The actual data is stored on HDFS and PM can access that file by the link of meta data. When a job is to run on a graph, the master node will ask PM to generate the Hyper Graph with the original graph, if existed, the latest partitioning result and LAVS of the that graph. After that, PM partitions the graph to generate the initial partitioning. Then, PM checks the meta data and fetches the historical information to generate the initial partitioning result for this graph. The master node then uses this partitioning result to assign vertices of the graph from HDFS to worker nodes.

During the job execution, when each worker finishes a superstep, it sends the estimated running time of that partition to PM. PM then gathers and analyzes the information from worker nodes and applies the repartitioning strategy to optimize the running time of the next superstep. If a repartitioning is needed for a partition on a node, PM will send a repartitioning trigger along with the necessary information to the PA of that node and a repartitioning procedure will be started on that node.

LogGP Partitioning Agency: Partitioning Agency (PA) is running on each worker node in the Giraph system to collect running time information and manage to repartition the graph when there will be a running time skew in the next superstep. PA uses the same communication module in Giraph to communicate with PM or other PA(s). PA collects the information when the superstep is running, such as $tComp$, $tComm$ and $C(v)$, then uses this information to compute the estimated running time for the next superstep. In fact, when a vertex is executing a user defined function, such as Pagerank, PA will check the type of vertex and record the time used for computing. When data is sent to the other partitions, PA records the vertex sets and the time spent on the communication. This running log will be analyzed after each superstep finishes and generates statistical logs for that superstep. $tComp$, $tComm$, $C(v)$ and other parameters used for superstep repartitioning are obtained from these logs. When a superstep is finished, in synchronization step, PA will send the estimated time of next superstep to PM and wait for the response from PM. If a repartition is necessary, PM will run the repartitioning heuristic to rebalance the running time. PA also writes running logs of active sets of each superstep. When the workload is finished, LAVS is generated from the running logs. When all the LAVSs of continuous k supersteps are generated, the results are uploaded to HDFS for PM to generate Hyper Graph for next workload. This process is fast and does not take much resource in that node.

Vertex Migration: For Superstep Repartitioning, when the dynamic partitioning scheme decides to reassign vertices from one partition to another, three types of data need to be sent: (1) latest value of vertices; (2) adjacency list of vertices; (3) messages for the next superstep. One solution is to add a new stage for vertex reassign between the end of superstep n and beginning of superstep n+1. LogGP uses another option that combines vertex moving within the Synchronization stage. We combine the messages sent from one partition to another to reduce the times of communication. In addition, the number of vertexes which need to be moved is small. Thus, the reassignment cost is slight compared with other operations. We will further discuss the time used for vertexes reassignment in experiment studies.

When a vertex gets reassigned to a new worker, every worker in the cluster must obtain and store this information in order to deliver future messages to the vertex. An obvious option for each worker is to store an in-memory map consisting of $< Vertex_{id}, Worker_{id} >$ pairs. However, using this solution, a vertex u must broadcast to all its neighbors when it is moved from one node to another. To reduce the cost, we implement an high-efficient lookup table service mentioned in [24] to provide a lookup table that reduces the broadcast cost when reassigning vertices.

6. EVALUATION

In this section, we evaluate the performance of our proposed partitioning refinement approaches. We implemented LogGP on Giraph [1], and the lookup table service [24] for vertex migration solution.

6.1 Experimental Settings

We first briefly introduce the experimental settings for the evaluation, including datasets, evaluation metrics and comparative approaches. All the experiments were conducted on a cluster with 28 nodes with an AMD Opteron 4180 2.6Ghz CPU, 48GB memory and a 10TB RAID disk. All the nodes were connected by 1Gbt bandwidth routers.

6.1.1 Data Sets

We used 5 real-world datasets: Live-Journal, Wiki-Pedia, Wiki-Talk, Twitter and Web-Google; and one Synthetic graph which is generated following the Erdös-Rényi random graph model. Those real-world datasets are publicly available on the Web [5], and the statistics of the datasets are shown in Table 1. We transformed them into undirected graphs, added reciprocal edges and eliminated loop circles from the original release.

Dataset	Nodes	Edges	Type	Size
Wiki-pedia	2,935,762	35,046,792	Web	401MB
Wiki-Talk	2,388,953	4,656,682	Web	64MB
Web-Google	875,713	8,644,106	Web	72MB
Live-Journal	4,843,953	42,845,684	Socia	479MB
Twitter	41,652,230	1,468,365,182	Social	20GB
Synthetic	5,000,000	100,000,000	Synthetic	930MB

Table 1: Graph Dataset Statistics

6.1.2 Evaluation Metrics

We use two metrics to systematically evaluate the result of our experiment. For both of the two metrics, a lower value represents the better performance.

Edge Cut Percentage(ECP): It indicates the percentage of cut edges between partitions in the graph, defined as $ECP = ec/|E|$, where ec denotes the number of cut edges between partitions, and $|E|$ denotes the total number of edges in the graph. This is the basic metric to evaluate the quality of partitioning result.

Execution Time: We utilize two time costs to evaluate the system performance. The first one is *Job Execution Time (JET)* which presents the elapsed time from submitting a graph workload till its completion. We use JET to evaluate the actual effectiveness of the partitioning. After we partition the graph, we run the workload on the partitioned graph and record the time. The second one is *Total Running Time* which includes the workload execution time, as well as the graph loading and partitioning time.

6.1.3 Comparative Methods

In this experimental study, we select several state-of-the-art partitioning methods for comparison to demonstrate the advantage of our proposed method.

- The proposed **LogGP** method uses two techniques, Hyper Graph Repartitioning (**HGR**) and Superstep Repartitioning (**SR**), to reduce the overall job execution time of graph system. To better examine the effectiveness of our approach, we also study the performance of **HGR** and **SR** individually.

- Linear Deterministic Greedy (**LDG**) approach [23] is considered as one of best static streaming method, and Restreaming LDG (**reLDG**) approach [19] is extended to generate initial graph partitioning using the last streaming partitioning result.

- **CatchW** [21] is a dynamic graph workload balancing approach for random initial partitioning, which is a comparative approach for **SR**, as both of them try to adjust the partitions in the supersteps.

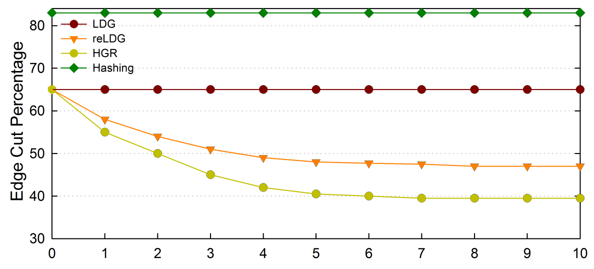
- We also use **Hashing** as one competitor because of its simplicity and popularity, e.g., Pregel uses hash function to partition the vertices by default.

6.2 Effect of Hyper Graph Repartitioning

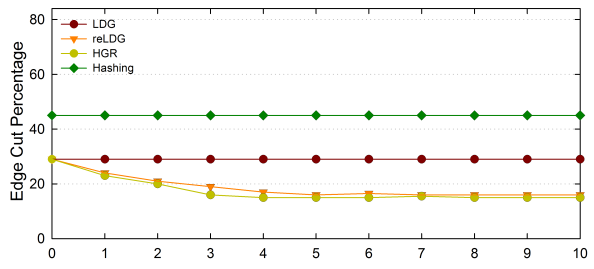
We first evaluate the performance of Hyper Graph Repartitioning (HGR). The state-of-the-art streaming algorithm LDG [23], reLDG [19] and Hashing are used as the base line. We run the experiment against all the graph datasets shown in Table 1. Here we mainly present the results on Live-Journal, Web-Google and Synthetic dataset as representative of Social Graph, Web Graph and Random Graph respectively.

Figure 8 shows the ECP of partitioning results on these three types of datasets. Each partitioning algorithm is executed 10 times, and HGR and reLDG use the last partitioning result for repartitioning. To generate the LAVS for the hyper graph, we execute a Semi-clustering workload after the partitioning. As we can see, HGR works well on all the graphs, especially on the Social Graph. This is mainly because that these social graphs representing real relationships of social network have significantly higher average local clustering coefficient. After running the Semi-clustering workload, LogGP records the LAVS of Semi-clustering that represent the internal relationships of these vertices. Then HGR generates Hyper Graph with this additional information to repartition the graph. With the help of Hyper Graph, streaming algorithm HGR can use more useful vertex relation to get better partitioning result. Thus, our approach outperforms reLDG which only uses the partitioning result while our approach uses additional LAVS information for initial partition. For Web Graph, known as seriously power-law skewed, our approach is still the best. Though in Synthetic Random Graph, our approach gets the smallest improvement, there is still about 15% reduction of ECP comparing with LDG. For Random Graph, there is no internal connection between vertices, thus, LAVS cannot fetch the relationships resulting in less improvement. However the previous partitioning result is shown to be helpful for these random graph as well.

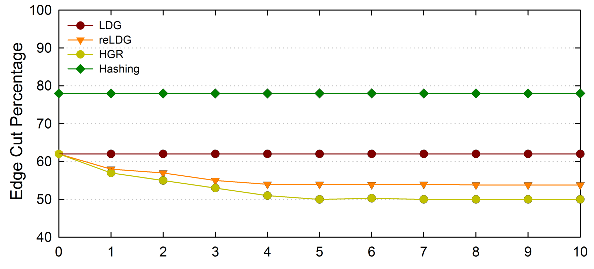
In addition, we evaluate the running time of these partitioning results after 10 times repartitioning. We run a 10-iteration of Pagerank and Semi-Clustering workloads on Live-Journal dataset. We compare the running time of HGR with Hashing, reLDG and LDG. As shown in Figure 9(a), HGR significantly reduces the running time on these two workloads, which confirms that our approach can refine the streaming partitioning result with the Hyper Graph.



a. ECP Results of 10 Iterations on Live-Journal



b. ECP Results of 10 Iterations on Web-Google

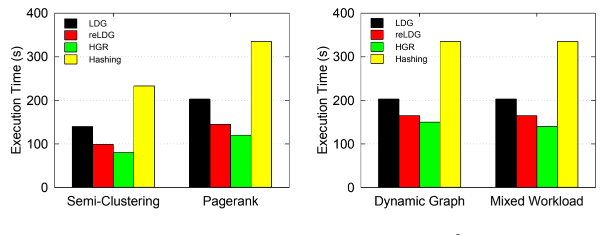


c. ECP Results of 10 Iterations on Random Graph

Figure 8: ECP of 10 Iterations on 3 Types of Datasets

In real-world, the workload and graphs are changing time by time. To simulate the graph changing, we evaluate our method using graph that changes dynamically between different workload iterations. We used 75% vertices in the graph for the first workload and then added 5% vertices each time. At last, when the workload is running for the 5 times, all the vertices is used. We evaluate the running time of HGR compared with other partitioning algorithms after 5 times repartitioning, and the results shown in Figure 9(b) demonstrate the efficiency and robustness of our approach with respect to graph updates. We also design an experiment to evaluate the situation when workload is changing during different iterations. We use 5 workloads: Semi-Clustering, Two-hop Friendship, Single Source Shortest Path, Breadth First Search and Pagerank for workload 1, 2, 3, 4 and 5 iterations separately. We execute these 5 workloads in sequence with HGR and other partitioning algorithms, and present the result of the last workload Pagerank in the paper. Figure 9(b) reports the superiority of our method.

Parameter tunning of HGR: As we mentioned in Section 3, we use a parameter k to determine the size of vertices in a LAVS. We conduct a series of experiments to investigate the selection of k. Due to the space limit, we only show the result of ECP on Live-Journal dataset using Pagerank for LAVS, in Figure 10. We find that when $k = 2$ or $k = 3$, HGR achieves the best performance for most of the workloads. This is because when k is small ($k = 1$), the size of LAVS is too small to present the connections between vertices. When the size of LAVS becomes larger ($k > 3$), the LAVS contains too much vertices which may not have strong relationships. Thus we use $k = 3$ as the number of supersteps to form LAVS in the experiment.



a. b.

Figure 9: Job Execution Time on Live-Journal Dataset

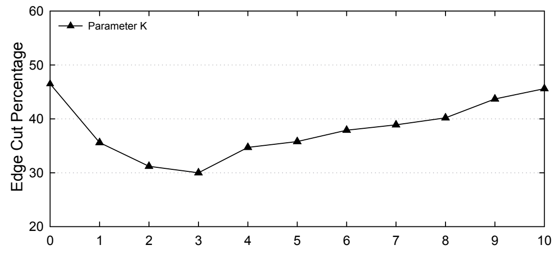


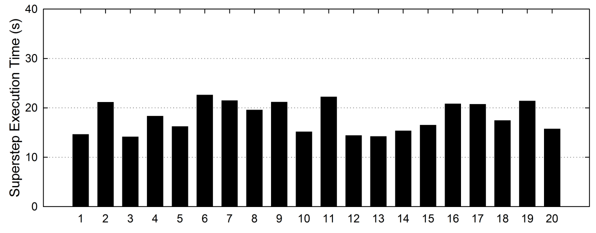
Figure 10: Parameter of LAVS Tuning

6.3 Effect of Superstep Repartitioning

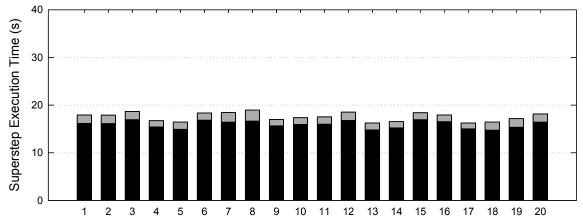
We next present the performance of Superstep Repartitioning (SR). We use a representative communication-intensive workload, Pagerank, for the evaluation. The parameter ϑ used for the percentage number that determines the threshold in Formula 12 is set as 2% which shows better result than other values. For Superstep Repartitioning which balances the running time of each partition during the execution, JET is a more important performance metric than ECP. Notice that the JET of SR contains the time of vertex reassignment.

To better understand how Superstep Repartitioning balances the running time of each workload, we show the time of a sample node on Live-Journal dataset in the first iteration of Pagerank workload in Figure 11(a) and the running time of the same node in the 4th superstep in Figure 11(b). We can observe that after three times of vertex refinement, the running time of each node is balanced with even shorter running time as well. In our experiment, the JET contains the repartitioning time which is marked as grey bars in Figure 11(b). These results also confirm the efficiency of migrating and locating the reassigned vertices in LogGP .

We next evaluate the effect of SR with different datasets. Figure 12(a) shows the result of JET of a 10-iteration Pagerank workload on different graphs using LDG streaming partitioning as the initial partitioning. Note that, the initial partitioning method is orthogonal to superstep repartitioning, and we will next show results on different initial partitioning results. We compare SR with CatchW discussed in [21] which is a dynamic graph workload balancing approach. The original method denotes running the experi-



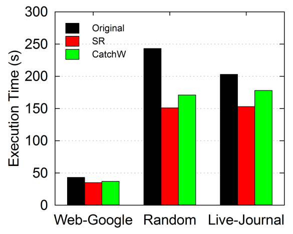
a. Execution Time of 1st Superstep



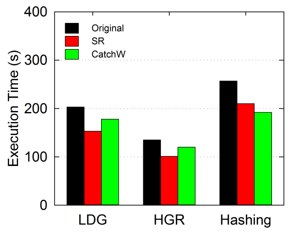
b. Execution Time of 4th Superstep

Figure 11: JET of One Sample Node in Different Supersteps

ment without repartitioning approach during supersteps. As we can see, there is a significant decrease on JET comparing with the approaches without Superstep Repartitioning, with a reduction of 11.9% to 27% JET on these datasets. SR runs faster than CatchW on all of the 3 datasets.



a. Performance of Pagerank



b. Comparing with CatchW

Figure 12: Performance of Superstep Repartitioning

We further evaluate the influence of different initial partition results to SR. We use the results of HGR, LDG and Hashing approaches as initial partition results. Figure 12(b) summarizes the JET cost of these three repartitioning approaches. We run the experiment on Live-Journal dataset for 10-iteration Pagerank workload. As shown in Figure 12(b), our repartitioning approach yields impressive performance on HGR and LDG while there is no significant improvement on Hashing initial partition. The reason is that our algorithm focuses on balancing the execution time of each node during superstep. The hashing partition is actually more balanced, although running slower, than LDG. For LDG as initial partition, our approach outperforms CatchW, while CatchW is better on LiveJournal dataset with Hashing as initial partitioning. CatchW is primarily designed for starting from an initial random graph partitioning and reassigning the vertices to reduce the communication cost, while our approach focuses on how to balance the running time of each node according to the running log from a streaming initial partitioning. Clearly, our approach is more efficient for practical situation.

This experiment indicates our Superstep Repartitioning approach can balance the running time during execution and reduce the total execution time of graph processing job.

6.4 The Superiority of LogGP

From the previous experiments, we can see that Hyper Graph Repartitioning (**HGR**) and Superstep Repartitioning (**SR**) yield promising results in initial graph partitioning and superstep graph adjustment respectively. In this experiment, we investigate the overall performance of LogGP which can take the both advantages of **HGR** and **SR**.

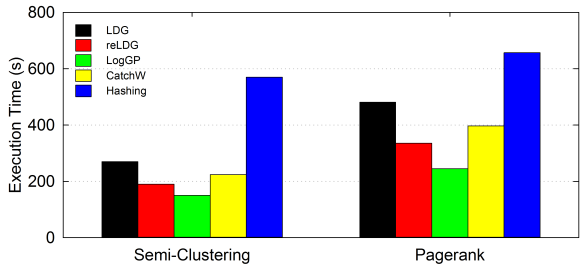
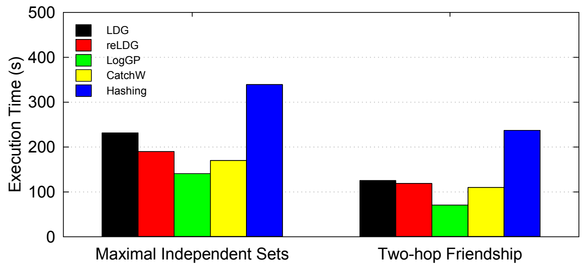
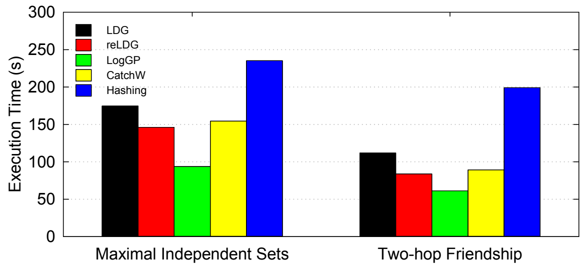


Figure 13: Overall Performance of logGP

Figure 13 illustrates the Running time of 10-iteration Pagerank and Semi-clustering workload on Twitter dataset. We compare with all the approaches mentioned above: LDG, reLDG, and CatchW. As we can observe, comparing to LDG, there are about 49.1% running time reduction on Semi-clustering and 39.2% improvement on Pagerank. The result of LogGP outperforms the CatchW and reLDG as well. These results confirm that LogGP with the help of log information can significantly reduce the job execution time when comparing with other approaches.



a. Wiki-Pedia



b. Wiki-Talk

Figure 14: Performance on Other Datasets

Besides Twitter dataset, we also evaluate the systems on Wiki-Pedia and Wiki-Talk dataset with workloads Maximal

Independent Sets and Two-hop Friendship. As shown in Figure 14, LogGP outperforms all the other methods on all these datasets and workloads. These results here also reveal the generality of the proposed approach and its wide application to various domains.

We next conduct the experiments to further study the total runing time of the graph processing job. We run the Pagerank 10-iteration workload on Live-Journal data, and Figure 15 shows the results of total running time including graph loading time (grey bar), partitioning time (white bar), and JET (black bar). The overall performance of our method is the best, though it introduces slightly more partitioning overhead.

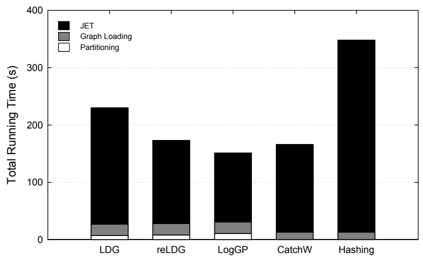
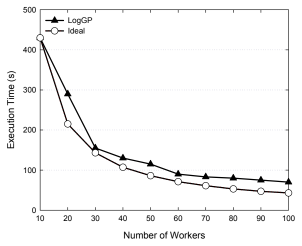
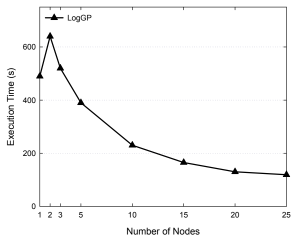


Figure 15: Study of Total Running Time



a. Different Workers



b. Different Nodes

Figure 16: Performance of Scalability

Scalability Study: Scalability is an important feature for parallel graph processing system. We further evaluated the scalability of LogGP with 1) fixed number of 28 nodes cluster and increasing number of workers; 2) fixed number of 100 workers and increasing number of nodes.

Figure 16(a) shows the performance of running 50-iteration of Pagerank on the Live-Journal with the number of workers increasing from 10 to 100. We use an ideal curve to denote the ideal execution time which assumes the performance is linear to the worker number for comparison. As expected, as the number of workers increases, the improvement decreases slightly and the performance of LogGP is close to the ideal. This result confirms that LogGP has a graceful scalability.

We also conduct an experiment by varying the number of nodes from 1 to 25 with fixed 100 workers, and the results are shown in Figure 16(b). Not surprisingly, the performance on one node outperforms that of two and three nodes, because the cluster with multiple nodes introduce extra communication cost between nodes. However, it is clear that LogGP performs better if we further increase the number of nodes, e.g., ≥ 5, as the gain from the computation capacity of cluster overcomes the communication overhead.

To summarize, based on our experimental results, we conclude that LogGP can take advantage of historical log information for partitioning refinement and provide an effective partitioning solution for large graph processing system.

7. RELATED WORK

The graph partition problem discussed in this paper is related to several fields such as graph computing systems and graph partitioning.

Large-Scale Graph Computing: To meet the current prohibitive requirements of processing large-scale graphs, many distributed methods and frameworks have been proposed and become appealing. Pregel [17] and GraphLab [15] both use a vertex-centric computing model, and run a user defined program at each worker node in parallel. Hama [2] and Giraph [1] are open source projects, which adopt Pregel programming model, and adjust for HDFS. In these parallel graph processing systems, it is important to partition large graph into several balanced sub-graphs, so that parallel workers can coordinately process them. However, most of the current systems usually choose simple hash method.

Graph partitioning: Graph partitioning is a combinatorial optimization problem which has been studied for decades. The k-balanced graph partitioning aims to minimize the number of edge cut between partitions while balance the number of vertices. Though the k-balanced graph partitioning problem is an NP-Complete problem [10], several solutions have been proposed to tackle this challenge.

Andreev et al. [6] presented an approximation algorithm which guarantees polynomial running time with an approximation ratio of $\mathcal{O}(logn)$. Another solution was proposed by Even et al. [9] who gave an LP solution based on spreading metrics which also gets an $\mathcal{O}(logn)$ approximation. Besides approximated solution, Karypis et al. [14] proposed a parallel multi-level graph partitioning algorithm to minimize bisection on each level. There are some heuristics implementations like METIS [13], parallel version of METIS [20] and Chaco [12] which are widely used in many existing systems. Although they cannot provide precise performance guarantee, these heuristics are quite effective. More heuristic approaches were summarized in [3].

The methods mentioned above are offline and require long processing time generally. Recently, Stanton and Kliot [23] proposed a series of online streaming partitioning method using heuristics. Fennel [25] extended this work by proposing a streaming partitioning framework which combines other heuristic methods. Joel Nishimura and Johan Ugander [19] futher proposed Restreaming LDG and Restreaming Fennel that generated initial graph partitioning using the last streaming partitioning result. Restreaming LDG and Restreaming Fennel exploits the similar strategy as HGR. However, HGR further uses the Log Active Vertex Set to get the internal relationship between vertices and combines the original graph, last streaming partitioning result and Log Active Vertex Set into a Hyper Graph. HGR then uses a novel Hyper Graph Streaming Repartitioning algorithm to partition the graph with global information.

Beyond these static graph partitioning technologies, [18] theoretically studies how to adapt graph structure changing without the overhead of reloading or repartitioning the graph. Some of the recent works [27, 8] can cope with the changes in graph structure. However, these approaches handle the changing in high cost. Shang et al. [21] investigated several graph algorithms and proposed simple yet effective policies that can achieve dynamic workload balance, while this approach uses hashing partitioning as the initial input. Compared to our repartitioning approach SR, CatchW tries to minimize the total communication cost. SR focuses on reducing the total JET of the systems which is more effective. In addition, SR uses the running statistical information during the execution and predicts the running time of each node in the next superstep. Base on the prediction, SR can accurately move the proper vertices.

8. CONCLUSION

In this paper, we systematically investigated the imbalance of running time on BSP model. We designed a novel log-based graph partitioning system to reuse the previous and running log information for partitioning refinement. We proposed Hyper Graph Partitioning which combines the graph and historical partitioning results to generate a hyper graph to improve the initial partitioning result. When the workload is executing, we use the running logs to estimate the running time of each node and design a novel heuristic to reassign the vertices from skew nodes to balance the running time. Prototype and experimental results confirmed the improvements of our new approaches.

There are several promising directions for our future work. First, a further theoretical analysis of streaming algorithm for Hyper Graph is preferred for better cost estimation. Second, other heuristics for vertex reassignment during the execution is an interesting topic.

9. ACKNOWLEDGEMENT

The research is supported by the National Natural Science Foundation of China under Grant No. 61272155 and 973 program under No. 2014CB340405.

10. REFERENCES

[1] *Apache Giraph.* https://github.com/apache/giraph/.
[2] *Apache Hama.* http://hama.apache.org/.
[3] *Graph Archive Dataset.* http://staffweb.cms.gre.ac.uk/~wc06/partition/.
[4] *HDFS.* http://hadoop.apache.org/common/docs/current/hdfs/design.
[5] *Snap Dataset.* http://snap.stanford.edu/data/index.html.
[6] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theor. Comp. Sys.*, 39(6).
[7] Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. Improving large graph processing on partitioned graphs in the cloud. In *Proc. of SOCC Conference*, pages 1–13, 2012.
[8] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proc. of EuroSys*, pages 85–98, 2012.
[9] Guy Even, Joseph (Seffi) Naor, Satish Rao, and Baruch Schieber. Fast approximate graph partitioning algorithms. *SIAM Journal on Computing*, 28(6):2187–2214, 1999.
[10] Michael R Garey, David S Johnson, and Larry Stockmeyer. Some simplified np-complete problems. In *Proc. of STOC Conference*, pages 47–63, 1974.
[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proc. of OSDI Conference*, pages 17–30, 2012.
[12] Bruce Hendrickson and Robert W Leland. A multi-level algorithm for partitioning graphs. *SC*, 95:28, 1995.
[13] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *Proc. of ICPP Conference*, pages 113–122, 1995.
[14] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.
[15] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. of VLDB Endow.*, 5(8):716–727, April 2012.
[16] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.
[17] 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 *Proc. of SIGMOD Conference*, pages 135–146, 2010.
[18] Vincenzo Nicosia, John Tang, Mirco Musolesi, Giovanni Russo, Cecilia Mascolo, and Vito Latora. Components in time-varying graphs. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 22(2):1–12, 2012.
[19] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proc. of SIGKDD conference*, pages 1106–1114, 2013.
[20] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.
[21] Zechao Shang and Jeffrey Xu Yu. Catch the wind: Graph workload balancing on cloud. In *Proc. of ICDE Conference*, pages 553–564, 2013.
[22] Isabelle Stanton. Streaming balanced graph partitioning for random graphs. *arXiv preprint arXiv:1212.1121*, 2012.
[23] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proc. of KDD Conference*, pages 1222–1230, 2012.
[24] Aubrey L Tatarowicz, Carlo Curino, Evan PC Jones, and Sam Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *Proc. of ICDE Conference*, pages 102–113, 2012.
[25] Charalampos E. Tsourakakis, Christos Gkantsidis, Božidar Radunović, and Milan Vojnović. Fennel: Streaming graph partitioning for massive scale graphs. Technical report, Microsoft, 2012.
[26] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
[27] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. In *Proc. of SIGMOD Conference*, pages 517–528, 2012.