Catch the Wind: Graph Workload Balancing on Cloud

Zechao Shang, Jeffrey Xu Yu

The Chinese University of Hong Kong, Hong Kong, China
{zcshang,yu}@se.cuhk.edu.hk

***Abstract*— Graph partitioning is a key issue in graph database processing systems for achieving high efficiency on Cloud. However, the balanced graph partitioning itself is difficult because it is known to be NP-complete. In addition a static graph partitioning cannot keep all graph algorithms efficient for a long time in parallel on Cloud because the workload balancing in different iterations for different graph algorithms are all possible different. In this paper, we investigate graph behaviors by exploring the working window (we call it *wind*) changes, where a working window is a set of active vertices that a graph algorithm really needs to access in parallel computing. We investigated nine classic graph algorithms using real datasets, and propose simple yet effective policies that can achieve both high graph workload balancing and efficient partition on Cloud.**

I. Introduction

Due to the large number of new applications need to deal with massive graphs, several graph database processing systems are developed [22]. As one of the representatives, Google has developed Pregel [15] as its internal graph processing platform based on bulk-synchronous parallel model (BSP) [25]. Pregel takes a vertex-centric approach and computes in a sequence of supersteps. In a superstep, Pregel applies a user-defined function (UDF) on every active vertex v in parallel, with the capability of receiving messages from other vertices to v in the previous superstep, and the capability of sending messages to other vertices (neighbors of v), which will be delivered in the next superstep. The idea behind Pregel is similar to MapReduce [7]. But, as pointed out in [15], MapReduce needs to pass the massive graph itself from one step to another step iteratively, which is time consuming. Pregel adopts BSP and implements a stateful model to support long-lived processes. Besides Google's own implementation, there are some public open source implementations which take similar approaches, such as HAMA [1] and Giraph [2]. Shao et al. in [22] survey systems and implementations for managing and mining large graphs on Cloud.

On Cloud with K computational nodes[1], the efficiency of any graph algorithm[2] relies on how to partition a large graph into K subgraphs, where all subgraphs are similar in size, the sets of the vertices of the K subgraphs are disjoint, and the number of edges across two subgraphs is minimized. Here, the similar in size is for workload balancing and the minimum number of crossing edges is for communication cost minimization, in parallel computing. However, the graph partitioning is challenging because it is known to be NP-complete [11], [4], and is challenging on Cloud as *partitioning extremely large, irregular datasets is only beginning to be addressed* [8]. Currently, the de-facto standard of graph partitioning is random partitioning for handling large graphs like social networks [20]. The two main reasons behind the random partitioning of large graphs are: (i) the extremely high cost of graph partitioning even in parallel [21] and (ii) the questionable efficiency that a single graph partitioning can achieve for any graph algorithms in general, where the efficiency is achieved by workload balancing among all computational nodes on Cloud. To deal with the graph partitioning, several replication based approaches are recently proposed [20], [16], [26], where the sets of vertices in the K subgraphs can be overlapped. Yang et al. in [26] investigate dynamic adaption of changing workload with such replication based on a reasonable good initial partitioning. Stanton et al. in [23] study several heuristic methods for graph partitioning under the stream computation model. Although they achieve considerable performance and scalability, they deal with static workloads. In this paper, as an attempt, we concentrate ourselves on dynamic workload balancing based on the vertex-centric computing that Pregel or similar systems take, without any requirements on the initial partitioning, and we do not allow vertex overlapping between computational nodes, because vertex overlapping may require an additional mechanism with possible high overhead for consistency maintenance, and costs more storage.

The main contributions of this work are summarized below. We investigate graph behaviors when executing graph algorithms in Pregel. The graph behavior is modeled as the set of the active vertices that a graph algorithm really needs to access in a superstep. The motivation behind is that we observe that the workload balancing should not be done for the entire graph once and thus used forever, because a graph algorithm does not always need to access all vertices in every superstep. This explains why a single static graph partitioning cannot achieve workload balancing because such graph partitioning is built for the entire graph. We investigate the graph behaviors by exploring the working window changes (we call it *wind* for short). We classify nine classic graph algorithms into three categories, and conduct extensive experimental studies on the working window changes for a random graph partitioning. Based on our experimental studies, we propose simple yet

[1]We use *node* to indicate a computational node, and *vertex* to indicate a vertex in the graph

[2]Here we mean general graph algorithms run on Pregel, which do not aware the underlying distributed environment

effective policies that can achieve workload balancing for the active vertices and reduce the number of edges across different computational nodes in supersteps starting from an initial random graph partitioning. We confirm the effectiveness and the efficiency of our policies on a prototyped system we have built on top of HAMA [1], an open source implementation of Pregel.

The remainder of the paper is organized as follows. We discuss Pregel in Section II on which our prototyped system is built up, and discuss nine graph algorithms in three categories to which we investigate how to support dynamic workload balancing in Section III. We show our experimental studies in Section V. We discuss the related work on graph partitioning in Section VI, and conclude our paper in Section VII.

II. The Pregel

Google has developed Pregel [15] as its internal graph processing platform based on bulk-synchronous parallel model BSP [25]. Pregel distributes data on all computational nodes, and during the entire computing all data are assumed to reside in main memory. Pregel takes a vertex-centric approach and computes in a sequence of supersteps. In each superstep, every node in Pregel computes a user-defined function (UDF) against each vertex in a graph in parallel. The UDF computes u's value based on the receiving messages to u, which are sent from other vertices in the previous superstep, and the UDF may also send messages from u to other vertices (neighbors of u) which will receive the messages in the next superstep. In general, a vertex u can receive/send messages from/to its neighbors which reside in the same or different nodes. The synchronization is done at the end of every superstep to ensure that all nodes complete their tasks (including sending/receiving messages) before entering the next superstep.

Pregel hides the underlying communication mechanism completely. The application to be built on top of Pregel cannot control any low level operations such as mode of sending/receiving messages, workload balancing, data partition and replication.

Some functions provided in the vertex-centric API in Pregel are shown below for accessing vertices.

```
class Vertex {
    VertexValue getVertexValue();
    void setVertexValue(Value);
    int getSuperStep();
    void SendMessageTo(Vertex, MessageValue);
    void VoteToHalt();
    void abstract UDF (MessageIterator);
}
```

Here, *getVertexValue()* and *setVertexValue()* get/set the vertex value, respectively. *SendMessageTo()* takes two inputs: the vertex to be sent and the message to be delivered. *getSuperStep()* gets the current superstep number. UDF is the function to be implemented by an application. The input to UDF is a *MessageIterator*, which is used to access every receiving messages to the corresponding vertex. In the following, for simplicity, for a vertex u, we take the *MessageIterator* as a set of messages to u denoted as M_u. Any vertex is initially set as **active**, and becomes **inactive** by calling *VoteToHalt()* by the vertex itself. An inactive vertex will become active again when it receives message(s). The Pregel repeats the supersteps until all vertices become inactive.

Algorithm 1: A Computational Node in Pregel

```
1 foreach active vertex u do
2     u.UDF (received messages to u from the previous
      superstep);
3 send out all outgoing messages;
4 enter barrier (all nodes wait here for sync);
```

TABLE I
SUMMARY OF THE NINE GRAPH ALGORITHMS

No.	Algorithm	Category
A1	PageRank [15], [17]	I
A2	Semi-clustering [15]	
A3	Graph Coloring [19], [13]	
A4	Single Source Shortest Path (SSSP) [15], [5]	II
A5	Breadth First Search (BFS) [6]	
A6	Random-Walk [18]	
A7	Maximal Matching (MM) [15], [3]	III
A8	Minimum Spanning Tree (MST) [19], [10]	
A9	Maximal Independent Sets (MIS) [19], [14]	

Algorithm 1 depicts a node in Pregel. In the *for* loop, it computes the UDF for each active vertex u in a superstep, and then sends out all messages from a vertex that requests. The sending (line 3) in Algorithm 1 is completely different from the member function of *SendMessageTo()* in the Vertex class. The latter is to highlight that it needs to send messages and the former is the one that really sends messages. Besides Google's own implementation, there are some public open source implementations which take the similar approaches, such as HAMA [1] and Giraph [2].

III. Nine Graph Algorithms

We discuss nine classic graph algorithms including PageRank [15], [17], Semi-clustering [15], Graph Coloring [19], [13], Single Source Shortest Path (SSSP) [15], [5], Breadth First Search (BFS) [6], Random-Walk [18], Maximal Matching (MM) [15], [3], Minimum Spanning Tree (MST) [19], [10], and Maximal Independent Sets (MIS) [19], [14]. In the Pregel framework, all graph algorithms need to implement a vertex-centric UDF function as indicated in line 2 in Algorithm 1. We divide all the 9 graph algorithms into 3 categories by the ways of a UDF function being designed, namely, *Always-Active-Style*, *Traversal-Style*, and *Multi-Phase-Style*. We discuss them below.

Always-Active-Style (Category I): By *Always-Active-Style*, every vertex in every superstep sends messages to all its neighbors. The main steps are illustrated in Algorithm 2. Here, the UDF function defined on a vertex u receives a set of messages M_u, which were sent to u in the immediate previous superstep by the neighbors of u. The UDF function first computes the value of u based on both the old value that u holds and the messages received (line 1). Then, it will inform all u's neighbors of the current value of u (line 2-3), and also

Algorithm 2: *Always-Active-Style* (M_u)

Input: M_u, all incoming messages to vertex u

```
1 Call an aggregate function on u based on M_u;
2 foreach outgoing edge e = (u, v) do
3     Send a message to v;
4 Change the local vertex value on u if necessary;
```

Algorithm 3: PageRank-UDF

Input: M_u, all incoming messages to vertex u

```
1 sum ← 0;
2 foreach message m ∈ M_u do
3     sum ← sum + m;
4 p ← u's value;
5 p ← α · p + (1 − α) · sum;
6 foreach outgoing edge e = (u, v) from u do
7     Send a message with value p/degree(u) to vertex v;
8 Set u's value to be p;
```

update the current value of u to be held for the next superstep (line 4). PageRank, Semi-clustering, and Graph Coloring are all in this category.

As an example, the vertex-centric PageRank UDF is shown in Algorithm 3 based on Pregel implementation [15]. Every vertex is assigned to an initial value, which is its initial rank. The UDF function computes a new rank at the i-th superstep for vertex u by combining its previous rank and the aggregation of the ranks of its neighbors (line 1-5). It then sends the new rank of u to its neighbors (line 6-7), and sets the new rank of u to be held for the next iteration. The termination condition is based on the pre-determined number of supersteps, and all nodes will terminate at the same time. It is important to notice that all vertices are active sending their updated ranks to their neighbors, because such rank values have impacts on the ranks of their neighbors, and their neighbors' neighbors, etc.

Traversal-Style (Category II): By *Traversal-Style*, a specific vertex is treated as a starting point, and the other vertices are involved in computing based on how it propagates on conditions. The main steps are illustrated in Algorithm 4. Initially, only one vertex will be set as active and all others are in the inactive status. In the computing, a vertex u becomes active when it receives some messages (M_u). As shown in Algorithm 4, the UDF function updates the value of u based on the messages received (line 1-2). If the value of u is updated, it implies that it is now involved in the computing of the given graph algorithm, and it will propagate to some of its selected neighbors on conditions (line 4-5). Because u does not necessarily need to be involved in the computing all the time, u will vote to halt by calling the member function *VoteToHalt()* to voluntarily be inactive (line 7). Note that a vertex will be waked up by the messages received. It is interesting to note that the category I algorithms, for example PageRank, cannot effectively use *VoteToHalt()*, because the value of a vertex will

Algorithm 4: *Traversal-Style* (M_u)

Input: M_u, all incoming messages to vertex u

```
1 Call an aggregate function on u based on M_u;
2 Update the vertex value for u if needed;
3 if u's vertex value is updated then
4     foreach outgoing edge e = (u, v) do
5         Send a message to v on condition;
6     Set the local vertex value on u if needed;
7 VoteToHalt();
```

Algorithm 5: SSSP-UDF

Input: M_u, all incoming messages to vertex u

```
1 d_min ← ∞;
2 foreach message m ∈ M_u do
3     d_min ← min(m, d_min);
4 p ← u's value;
5 if d_min < p then
6     foreach outgoing edge e = (u, v) from u do
7         Send a message to v, with a value of d_min + ω(u, v);
8     Set u's local value as d_min;
9 VoteToHalt();
```

always affect others, even if the value of the vertex satisfies the converge condition, for example, the error bound between the current value and its previous value is less than or equal to a threshold. Typical algorithms in the category II are Breadth First Search [6], Single Source Shortest Path [6], and Random-Walk [18].

The implementation of SSSP is illustrated in Algorithm 5. Initially every vertex u, except the start vertex, is assigned to an initial value ∞ for the shortest distance from the starting vertex to u. In SSSP, a vertex u receives messages from some of its neighbors if some of its neighbors have updated their shortest distance from the starting vertex. The UDF function will determine the new shortest distance from the starting vertex to u itself via some of u's neighbors (line 1-3). If the new shortest distance becomes smaller, u will send messages to its neighbors to tell them the new updates (line 6-7), where $\omega(u, v)$ is the edge weight on the edge from u to its neighbor v. Because u may not necessarily need to be involved in the shortest distance computing, it votes to halt (line 9).

Multi-Phase-Style (Category III): As illustrated in Algorithm 6, a single phase, for example, to identify one matching edge among many, cannot be done in a single superstep, because a matching edge is an edge (u, v) that neither u nor v is involved in another matching edge. It needs to be done in several supersteps in Pregel. The entire computation is divided into a number of phases, and each phase, P_j is done in k supersteps: $P_{j_0}, P_{j_1}, \cdots, P_{j_i}, \cdots, P_{i_{k-1}}$. Typical algorithms in the category III are MM [15], [3], MST [19], [10], and MIS [19], [14].

We discuss the maximal matching problem (MM), as an example, which is to find a maximal subset of edges, M_E,

Algorithm 6: *Multi-Phase-Style* (M_u)

Input: M_u, all incoming messages to vertex u

```
if u has completed its computing then
    VoteToHalt(); return;
switch getSuperStep() % k do
    ...;
    case i                                   /* 0 ≤ i < k */
        call either Always-Active-Style or Traversal-Style style
        UDF; break;
    ...;
```

Algorithm 7: MM-UDF

Input: M_u, all incoming messages to vertex u

```
if u itself is a matched point then
    VoteToHalt(); return;
switch getSuperStep() % 4 do
    case 0                                   /* invitation */
        foreach outgoing edge e = (u, v) from u do
            Send a message to v to invite ;
        accept_u ← false;
        break;
    case 1                                   /* acceptance */
        Randomly pick an edge e = (u, v) up from M_u;
        Send a message to v to accept; accept_u ← true; break;
    case 2                                   /* confirmation */
        if accept_u = false then
            Randomly pick an edge e = (u, v) up from M_u;
            Send a message to v to confirm;
            Set u as a matched point; VoteToHalt(); break;
    case 3                                   /* marking */
        if M_u ≠ ∅ then
            Set u as a matched point; VoteToHalt(); break;
```

in a given graph G where no two edges in M_E have a common vertex. In Pregel, a randomized algorithm [3] is used to support a bipartite maximal matching, which can be extended to handle maximal matching in general graph. The computation is divided into a number of phases, and each phase, P_i, is done in four supersteps, namely, P_{i_0}, P_{i_1}, P_{i_2}, and P_{i_3}. In P_{i_0}, all vertices that have not been involved in any matching send a message to their neighbors to invite them to join a match. In P_{i_1}, a vertex randomly accepts one of the invitations to join a matching, and replies the corresponding sender a message, because a vertex may possibly receive many matching invitations from its neighbors. In P_{i_2}, in a similar fashion, a vertex that has sent invitations may receive several acceptances, and will confirm one acceptance by replying a message. In P_{i_3}, the vertex that receives a confirmation will mark itself as marked. The computation repeats until no more matches can be identified. As illustrated in Algorithm 7, one important thing is that a vertex may not respond to any when it has already been matched.

IV. Super-Dynamic Partitioning

In this section, we discuss two important issues which are also conflict with each other in Pregel-like systems for large graph processing. The two issues are workload balancing and communication cost minimization. In Pregel, the number of vertices is the dominant factor for workload in each computational node, because of its vertex-centric computing. The workload balancing suggests that all nodes are best to have similar amount of workload in every superstep. The intuition is that the computing cost of a superstep is the max computing cost of the node that has the largest workload. Workload balancing will lead to minimization of computational cost of supersteps. On the other hand, the communication cost minimization suggests that the communication cost among supersteps shall be minimized. It is worth noting that, in graph processing in Pregel, such communication cost depends on the number of messages across computational nodes, and the number of messages is heavily dependent on the number of edges that cross different computational nodes. Minimizing the communication cost will significantly reduce the time for synchronization. The two issues are important, because the system excution time depends on both of them. The two issues are conflict. First, balancing workload among computational nodes may increase the number of edges that are across different computational nodes which leads to possibly high communication cost. Second, minimizing the communication cost may make all workload goes to a single computational node.

Graph Notations: Consider a graph $G = (V, E)$, where V is the set of vertices, and $E \subseteq V \times V$ is the set of directed edges. We use $n = |V|, m = |E|$ to denote the numbers of vertices and edges, respectively.

A graph G will be distributed to the K computational nodes, $\{N_1, N_2, \cdots, N_K\}$, in Pregel for parallel processing, where all N_i have the same memory capacity C.

At the s-th superstep, each node N_i holds a subset of vertices, $V_i^s \subseteq V$ under the condition that $V_i^s \cap V_j^s = \emptyset$ for $(i \neq j)$ and $V = \bigcup V_i^s$, and holds a subset of internal edges, $IE_i^s = \{(u,v)|(u,v) \in E, u \in V_i^s, v \in V_i^s\}$. The edges that cross nodes are called external edges. We use $EE_{i,j}^s$ to denote the set of external edges which cross two different computational nodes, N_i and N_j, $i \neq j$, such as $EE_{i,j}^s = \{(u,v)|(u,v) \in E, u \in V_i^s, v \in V_j^s\} \cup \{(u,v)|(u,v) \in E, u \in V_j^s, v \in V_i^s\}$. The edges associated with partition N_i is $E_i^s = \cup EE_{i,j}^s \cup IE_i^s$.

Working Window, Workload, and Workload Balancing: The computational cost on a computational node mainly depends on the number of *active* vertices that receive messages in the vertex-centric computing framework like Pregel. Here, the active vertices are a subset of the entire vertices in a superstep, because not all vertices will be invoked by messages in a superstep. We call the set of active vertices in the s-th superstep on a computational node N_i a working window, and denote it as W_i^s, and the entire working window at the s-th superstep is $W^s = \cup_{i=1}^K W_i^s$. The average size of working

window per node at the s-th superstep is denoted as $\overline{W}^s$, which is computed as $\overline{W}^s = |W^s|/K$.

We consider the **workload** in the s-th superstep on a computational node N_i denoted as $\mathcal{W}_i^s$.

$$\mathcal{W}_i^s = |W_i^s| + |AE_i^s| \quad (1)$$

Here, $|W_i^s|$ is the number of active vertices on N_i, and AE_i^s is the *active* subset of the edges. The average workload in the s-th superstep is $\overline{\mathcal{W}}^s = \sum_{i=1}^{K} \mathcal{W}_i^s/K$.

Eq. (1) indicates the workload depends on two factors. $|W_i^s|$ is the number of UDF functions must be invoked to compute, and $|AE_i^s|$ is the number of messages that UDF functions must process $|W_i^s|$. Eq. (1) takes the simple sum of $|W_i^s|$ and $|AE_i^s|$ for two reasons. First, it is based on our observations. For most UDF functions, the number of computing operations is linear w.r.t. the incoming messages. Second, the message serialization/deserialization time is linear to the size of messages, which usually overtakes other operations and dominates the UDF execution time.

In addition, we use $D_{i,j}^s$ to denote the number of messages sent from N_i to N_j at the s-th superstep, which is based on $EE_{i,j}^s$. $D_{i,j}^s$ implies the communication cost from N_i to N_j at the s-th superstep. The total communication cost at the s-th superstep is denoted as D^s.

Because it is very difficult to minimize the two objectives, namely, workload balancing and communication cost minimization, at the same time, in this paper, we study to minimize communication cost under the condition that workloads are balanced. The problem statement is given below.

Problem Statement: For a given graph algorithm expressible in Pregel, the problem we study is to minimize $\sum_s D^s$, subject to two conditions in every superstep: (1) $|V_i^s| + |E_i^s| \leq C$ and (2) at each superstep s, all computational nodes are **balanced**, namely $\mathcal{W}_i^s \leq (1+\epsilon) \times \overline{\mathcal{W}}^s, \forall i \in \{1..K\}$ where $0 \leq \epsilon \ll 1$. The first condition suggests that a node N_i must be able to hold the workload assigned, and the second condition suggests that the workload must be balanced.

The problem is challenging. First, the problem itself is an NP problem even in a static setting, as the size-balanced graph partition problem is shown to be NP-complete [4]. In addition, the active vertices, we are most interested in, are those vertices that will be active in the **near future** rather than those vertices that are active in the current superstep. This is because in the current superstep the active vertices have already been determined based on the messages already received from the immediate previous superstep, and cannot be rebalanced. Second, it is known to be very difficult to predict the workload in parallel graph processing on computational nodes. Third, there does not exist the optimal graph partitioning under one set of conditions that can be efficiently used for any graph problems in general, for the reason that the workloads vary greatly, as discussed for the 3 categories of graph algorithms. Fourth, this requests a super-dynamic solution during the computation of a graph algorithm, that must be effective and simple.

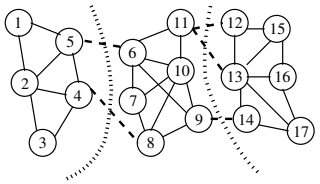
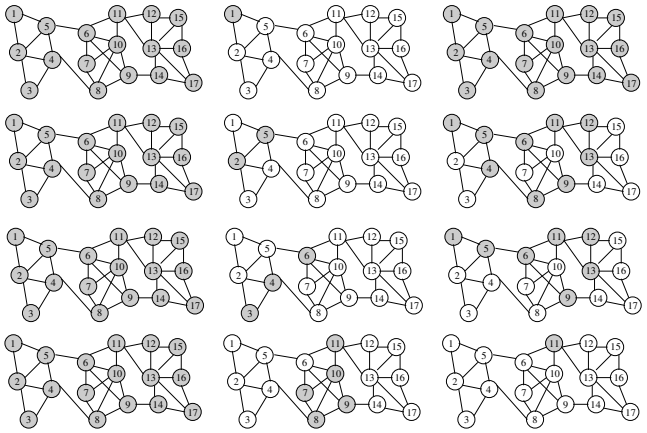


Fig. 1. A Graph Example



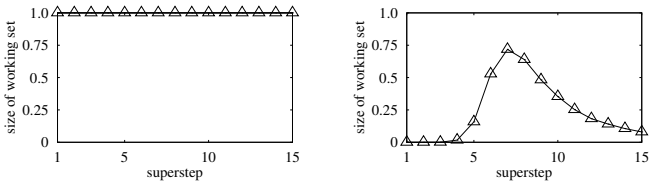
(a) PageRank (b) BFS (Start from 1) (c) MM

Fig. 2. Graph Examples

A. Various Working Window Behaviors

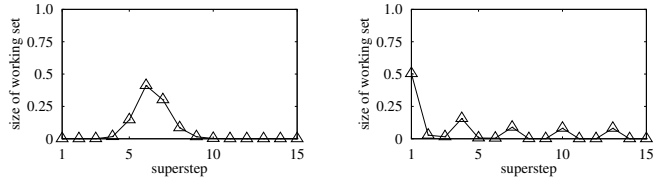
We show various working window behaviors using a graph example. Fig. 1 shows a graph with 17 vertices and 31 edges. Assume that there are 3 nodes N_1, N_2, and N_3. As divided by the dotted line in Fig. 1, N_1 holds the vertices $V_1 = \{v_1, v_2, v_3, v_4, v_5\}$ and the internal edges among V_1, N_2 holds the vertices $V_2 = \{v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$ and the internal edges among V_2, and N_3 holds the vertices $V_3 = \{v_{12}, v_{13}, v_{14}, v_{15}, v_{16}, v_{17}\}$ and the internal edges among V_3, respectively. The numbers of vertices in the 3 nodes are balanced with 5, 6, and 6 vertices. It implies that it is workload balanced if we simply let the working window $W_i^s = V_i$, for $1 \leq i \leq 3$, regardless the graph algorithms. The numbers of external edges are the minimum: 2 edges between N_1 and N_2 and 3 edges between N_2 and N_3. It implies that the communication cost is minimized. This graph partition can be done by most state-of-the-art graph partitioner like METIS [12].

Fig. 2 shows the the working windows in several supersteps using 3 graph algorithms, PageRank, BFS, and MM. In every superstep, a shaded vertex indicates an active vertex, and a white vertex indicates an inactive vertex. Fig. 2(a) shows the working windows for PageRank in the first 4 supersteps from top to bottom. All vertices in every superstep are active. Fig. 2(b) shows the working windows for BFS in the first 4 supersteps from top to bottom, starting from the starting vertex v_1. As can be seen from the supersteps, the 4 working windows are, $W^1 = \{v_1\}$, $W^2 = \{v_2, v_5\}$, $W^3 = \{v_3, v_4, v_6\}$, and $W^4 = \{v_7, v_8, v_9, v_{10}, v_{11}\}$, in the 1st, 2nd, 3rd, and 4th superstep, respectively. In the 1st and 2nd supersteps, the working windows are in N_1. In the 3rd superstep, N_1 and N_2 have active vertices to work on, but the number of active vertices is small. In the 4th superstep, all active vertices



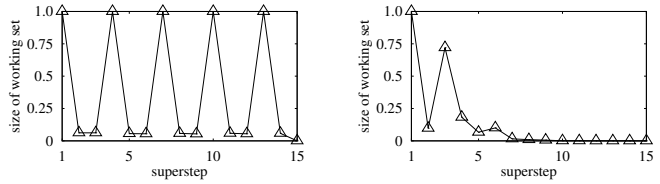
(a) PageRank (A1)

(b) SSSP (A4)



(c) BFS (A5)

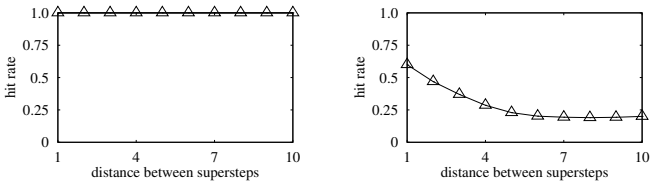
(d) MM (A7)



(e) MST (A8)

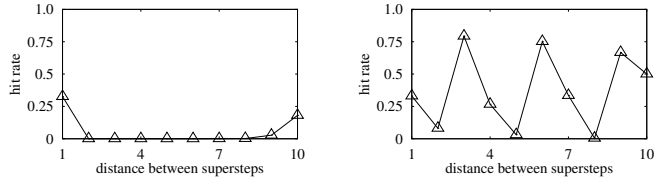
(f) MIS (A9)

Fig. 3. Working Window Sizes



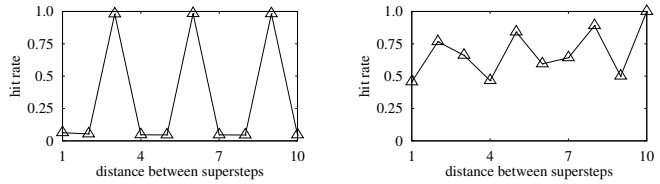
(a) PageRank (A1)

(b) SSSP (A4)



(c) BFS (A5)

(d) MM (A7)



(e) MST (A8)

(f) MIS (A9)

Fig. 4. Average correlation between the working windows

are on N_2. Fig. 2(c) shows the working windows for MM. Unlike the previous figures, we show the 1st, 5th, 9th, and 13th supersteps, because a single phase consists of $k = 4$ supersteps. Initially, in the 1st superstep, all vertices are active. In the later supersteps, unlike others, the node N_3 does not have much active vertices to work on.

As a remark, it is difficult to predict how the working windows change from time to time in general as the algorithm varies. Such changes possibly depend on the graph algorithm to be used and the large graph to be processed.

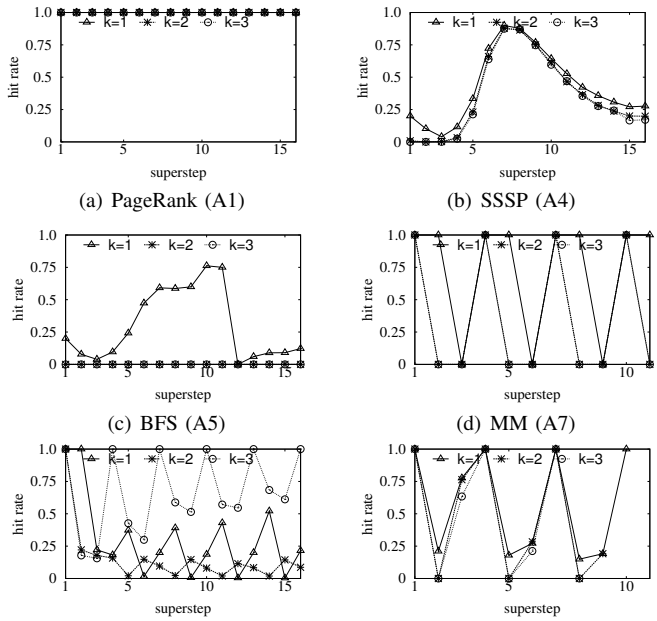
B. Catch the Working Window Patterns

We investigate the behaviors of the nine graph algorithms in Pregel in Table III, by analyzing the properties of working windows. The 9 graph algorithms are numbered by A1, A2, $\cdots$, A9. We have conducted the testing using all the datasets listed in our experimental studies. They all have the similar behaviors. Below, we show the results using the CA-Condmat data (`http://snap.stanford.edu/data/`), which is a collaboration dataset with 23K vertices and 186K edges. Auxiliary supersteps which do not have any communication are omitted in following charts for clarity.

Dynamic Working Window Changes: First, we show that working windows change dynamically. Fig. 3 shows the ratio of the working window size over the entire vertex set in all supersteps. Fig. 3(a) shows that all vertices are active for PageRank (Category I). For SSSP and BFS of *Traversal-Style* in Category II, the working windows increase and then decrease, as shown in Fig. 3(b) and Fig. 3(c). For MM, MST, and MIS of *Multi-Phase-Style* in Category III, the working windows show some periodical patterns in different ways, as shown in Fig. 3(d), Fig. 3(e), and Fig. 3(f). It confirms that it is difficult to use one graph partition to support any graph algorithms in Pregel. We also verify it by partitioning the CA-Condmat graph data into 16 computational nodes using METIS (the static graph partitioning approach), and run the nine graph algorithms. To show the workload balancing, we define an imbalance factor, which is the sum of the max workload for every superstep divided by the sum of average workload for every superstep. If the workload is well balanced, the imbalance factor is near 1. A larger imbalance factor implies that workloads are not well balanced. Our experiment shows that during processing graph algorithms, the imbalance factor can be up to 1.6. This leads to that the costly static graph partitioning can not serve all the purposes for workload balancing, because the workload changes dynamically.

Next, we discuss whether it is possible to catch the wind (working *wind*ow) in some way, even though it is known to be very difficult. We give some notations below first. Consider two working windows, W^p and W^q, at the p-th and q-th supersteps, respectively. The distance between the two working windows is denoted as $dis(W^p, W^q) = |p - q|$. We say a working window W_p is a k-distance previous working window of W_q if $p < q$ and $dis(W^p, W^q) = k$. The hit-rate of a working window W^q regarding another k-distance previous working window W^p is denoted as $hit_k(W^p, W^q)$ and is defined as $hit_k(W^p, W^q) = \frac{|W^p \cap W^q|}{|W^q|}$.

Long-Term Working Window Behaviors: Fig. 4 shows the relationship between k-distance and the hit-rate for every two working windows using the graph algorithms. Here, the x-axis in Fig. 4 is the k-distance, and the y-axis shows the average hit-rate for every two working windows that are k-distance. For example, when $k = 1$, the average hit-rate is the average of all $hit_1(W^{q-1}, W^q)$. In Fig. 4, we observe the behaviors of graph algorithms in the three categories. Fig. 4(a) shows hit-rate maintains 100% for any two working windows in k-distance where $1 \leq k \leq 10$, when PageRank (A1) is



(a) PageRank (A1) (b) SSSP (A4)

(c) BFS (A5) (d) MM (A7)

(e) MST (A8) (f) MIS (A9)

Fig. 5. The hit-rates between the working windows

executed. This is the typical behavior of graph algorithms in Category I (Table III), because all vertices are always active (Algorithm 2). Fig. 4(b) and Fig. 4(c) show the hit-rate when SSSP (A4) and BFS (A5) are executed, in Category II, respectively. The working windows show some locality property. The hit-rate decreases naturally when the distance become larger. Fig. 4(d), Fig. 4(e), and Fig. 4(f) show the periodical behaviors regarding the hit-rates, for MM (A7), MST (A8), and MIS (A9), in Category III, respectively. It is worth mentioning the periodical behaviors of the hit-rate for Category III algorithms. Every three supersteps, or in other words, a working window at a superstep, q, accesses the similar working window as the 3-distance previous working window, $q-3$. This has a strong relationship with the design of the vertex-centric algorithms, as can be seen in MM-UDF (Algorithm 7). For all the three categories, it suggests that moving working window to the right computational nodes at a superstep can possibly reduce the communication cost in the following supersteps.

Short-Term Working Window Behaviors: Fig. 5 shows the $hit_k(W^{q-1}, W^q)$, for $k = 1, 2, 3$, for each superstep. Unlike Fig. 4 which shows the average, Fig. 5 shows $hit_k(W^{q-1}, W^q)$ for each superstep from 1 to 15. For PageRank, since it is *Always-Active-Style*, the hit-rate is always 100% (Fig. 5(a)) for $k = 1, 2, 3$. For SSSP and BFS, during the computing, the hit-rate increases up to a heap and then decreases (Fig. 5(b) and Fig. 5(c)). It exhibits high hit-rate between two consecutive supersteps ($k = 1$). For MM, MST, and MIS, Fig. 5(d), Fig. 5(e), and Fig. 5(f) show the periodical patterns. It exhibits high hit-rate between two consecutive supersteps when $k = 3$ instead of $k = 1$, due to the graph algorithm designs in Category III.

Discussions: The traditional parallel system load balancing approaches collect statistics for a moderate-length history to perform analyze and strategies. It becomes infeasible in our problem, because we have to consider load balancing in every superstep in processing a graph algorithm. In other words, we have to deal with load balancing using a very short history which dynamically changes. A question that arises is whether there exist some patterns for us to predict working window movements. Such an answer depends on both the underneath graph and the graph algorithm. It is very difficult to have an analytical result at this stage. In this work, based on the extensive experimental studies, we observe the three categories of graph algorithms have potential predictability. The *Always-Active-Style* algorithms have stable working windows, as we can predict working windows. For the *Traversal-Style* algorithms, even though the predictability cannot be ensured in general, however, in real applications, most of large scale graphs have the low-diameter property. This means that the average distance between two vertices are short, hypothesized to be in the same scale of log of the size of graph. Also the size of k-neighbors, which means all vertices are reachable in no more than k hops, is exponential to k. It implies that some *Traversal-Style* algorithms, for example SSSP, have a reasonable hit-rate between supersteps. However, on the other hand, BFS in Category II has a very low hit-rate between supersteps, because it literally traverses a graph. For Category III algorithms, we need to consider a phase as a whole. The hit-rate between two successive phases is very high, due to the nature of the algorithm design that accesses a certain set of vertices, determines some partial results, and moves from the set of such vertices to another nearby set of vertices. Our observation suggests that in many cases the working windows can be possibly predicted using a very short history.

C. A Distributed Vertex-Centric Scheduler

In this section, we discuss our distributed vertex-centric scheduler. The main idea is to allow each node N_i to decide by itself how much workload should be moved out to another N_j, in order to minimize the total communication costs. There are two issues. The first issue is how much workload we should move from N_i to N_j. And the second issue is what we should move from N_i to N_j.

How much to move: For the first issue, we keep the hard workload limit as much as possible, in order to ensure the workload balancing. Assume the current superstep is s. If N_i is overloaded in the current s-th superstep, all other nodes N_j, for $i \neq j$, should not move any vertices towards N_i, and the overloaded node N_i should move its vertices out to other nodes, to keep all nodes balanced in the (s+1)-th superstep. This approach can prevent nodes from being severely overloaded. We introduce a quota $Q^s_{i,j}$ to determine the workload to be moved from N_i to N_j on the current s-th superstep for the next (s+1)-th superstep, and call it a quota function.

$$Q^s_{i,j} = \frac{\mathcal{W}^{s-1}_i - \mathcal{W}^{s-1}_j}{K} \quad (2)$$

Algorithm 8: Move(i,j)

```
if Q^s_{i,j} = 0 then
    Q^s_{i,j} ← Q_threshold;
Δ ← Q^s_{i,j};
Let L_{i,j} be a ranking list of vertices in N_i;
while Δ > 0 do
    remove u from the top of L_{i,j};
    if W^s_{i,j}(u) < Δ then
        move u from N_i to N_j;
        Δ ← Δ − W^s_{i,j}(u);
```

Here, $Q^s_{i,j}$ is determined based on the workload in the previous (s-1)-th superstep. A question that arises is why we must use the previous workloads, $\mathcal{W}_i^{s-1} - \mathcal{W}_j^{s-1}$, and cannot use the current workload, $\mathcal{W}_i^s - \mathcal{W}_j^s$. Because the workload in the current superstep has already been determined and cannot be changed without introducing an extra synchronization barrier which is costly in BSP computing. Based on the discussions in Section IV-B, the working window has high hit-rate based on the previous working window, we use $\mathcal{W}_i^{s-1} - \mathcal{W}_j^{s-1}$ to estimate $\mathcal{W}_i^s - \mathcal{W}_j^s$. This is confirmed to be effective in our extensive testing.

Based on $Q^s_{i,j}$, a node N_i can move the workload from N_i to N_j, if $Q^s_{i,j} > 0$. No workload can be transferred from N_i to N_j, if $Q^s_{i,j} < 0$. By a non-zero $Q^s_{i,j}$, it becomes unlikely that other nodes, N_j, will move its workload to N_i if N_i is overloaded. When $Q^s_{i,j} = 0$ in a superstep or in the initial superstep where we do not know the workload yet, N_i and N_j may move some workload to each other. We control the amount of workload to be moved under a small threshold, for example, $\alpha = 1\% \times \frac{|V|+|E|}{K^2}$. The numerator, $|V|+|E|$, is the total number of vertices and edges, $\frac{|V|+|E|}{K^2}$ is the amount that can be moved from N_i to N_j on average. We only move out up to its 1%, which is a very small amount. It is better to allow some vertices to be moved from N_i to N_j even if $Q^s_{i,j} = 0$. Otherwise if the workload is perfectly balanced, which could be achieved by hash based distribution policy, no move will be allowed at all. And in *Always-Active-Style* type algorithms, there will be no move forever.

Suppose $Q^s_{i,j} > 0$. We move some workload from N_i to N_j as bounded by $Q^s_{i,j}$. Suppose that we move a vertex u from N_i to N_j, then the workload associated with this move on the vertex u is denoted as $W^s_i(u)$. Here, $W^s_i(u)$ is equal to $1 + |AE^s_{i,j}(u)|$, where the value of 1 is the number of vertices to be moved, $|AE^s_{i,j}(u)|$ is the number of active edges. A vertex u can be moved from N_i to N_j if $W^s_i(u)$ is less than $Q^s_{i,j}$. Algorithm 8 illustrates the idea. It is worth noting that a ranking list, $L_{i,j}$ is used for $Q^s_{i,j}$ in Algorithm 8. A vertex can only appear in a ranking list at most once, and the vertices in the list is ranked based on the degree of benefit of moving it from N_i to N_j . Below, we discuss the ranking which is based on a function called *willingness score*.

What to move: Assume a vertex u is in a computational node N_i. Let $\mathcal{I}^s_{i,j}(u)$ and $\mathcal{O}^s_{i,j}(u)$ be the number of in-messages received and the number of out-messages sent to/from the vertex u to a node N_j in the s-th superstep, respectively. We denote the total communication cost for the vertex u by $IO_{i,j}(u)$, which is defined in Eq. (3).

$$IO^s_{i,j}(u) = \mathcal{I}^s_{i,j}(u) + \mathcal{O}^s_{i,j}(u) \tag{3}$$

It is important to note that a vertex u is an active vertex if it receives/sends messages. Based on $IO^s_{i,j}(u)$, the first vertex-centric willingness function for workload balancing is to select the computational node N_j for a vertex u to move to if N_j has the max $IO^s_{i,j}(u)$ value among all nodes (Eq. (4)).

$$\varpi^s_0(u) = argmax_j\ IO^s_{i,j}(u) \tag{4}$$

Suppose u is in node N_i. Eq. (4) takes the simple majority, and decides the node N_j for u to move. If u has more internal edges than the external edges, $\varpi^s_0(u)$ will return N_i, which suggests that u is not supposed to be moved, and will stay in the same node N_i.

Suppose u is about to be moved to N_j from N_i. As can be seen from Eq. (4), it does not consider whether the vertices in the neighbor of u in N_j will move to a different node N_k for $k \neq j$ at the same time when u is moved to N_j. However, it is difficult to predict whether the vertices in the neighbor of u in N_j will move or not, and it is costly to do so, even it can be done. Alternatively, from a different angle, the strategy we take is to keep a vertex u in N_i for certain interval, in order for other vertices to be moved to N_i. In other words, we do not move u from one to another new computational node frequently. Based on this observation, we define a new willingness function, $\varpi^s_1(u)$.

$$\varpi^s_1(u) = \begin{cases} \varpi^s_0(u) & \text{not moved in the past } \beta \text{ supersteps} \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

Here, β is a parameter to control at least how long a vertex u is forced to stay in a node.

Next, we consider the potential problem with the simple majority used in Eq. (4). Consider the following example. With Eq. (4), suppose that if u remains in the same node N_i, $IO^s_{i,i}(u) = 100$, and further suppose if u moves to a different node N_j, $IO^s_{i,i}(u) = 101$. The gain of moving u to a different node is little and is with cost. In order to handle this problem, we consider to take a ratio into account when moving vertices, instead of simple majority. In other words, we only move u to N_j if the gain is higher than a ratio λ, $\frac{IO^s_{i,j}(u)}{IO^s_{i,i}(u)} \geq \lambda$. We modify Eq. (4) as follows.

$$\widetilde{IO}^s_{i,j}(u) = \begin{cases} \lambda(\mathcal{I}^s_{i,j}(u) + \mathcal{O}^s_{i,j}(u)), & i = j \\ \mathcal{I}^s_{i,j}(u) + \mathcal{O}^s_{i,j}(u), & \text{otherwise} \end{cases} \tag{6}$$

where $\lambda > 1$. Accordingly, we define a new willingness function, $\varpi^s_2(u)$ below.

$$\varpi^s_2(u) = argmax_j\ \widetilde{IO}^s_{i,j}(u) \tag{7}$$

All the willingness functions are vertex-centric and do not consider the workload in the computational nodes. Since the

workload is a very important factor to be balanced, we define a communication cost formula

$$\widehat{IO}^s_{i,j}(u) = (\mathcal{I}^s_{i,j}(u) + \mathcal{O}^s_{i,j}(u)) \times (1 - \frac{\mathcal{W}^{s-1}_j}{\overline{\mathcal{W}}^{s-1}}) \quad (8)$$

that takes both vertex willingness to move and the immediate previous workload into consideration. Consider a computational node N_j for a vertex to be moved to. If its workload in the previous (s-1)-th superstep, $\mathcal{W}^{s-1}_j$, is higher than the average workload $\overline{\mathcal{W}}^{s-1}$, then we do not attempt to move a vertex to N_j. There are many possible ways to define such a function. For simplicity, we show the simplest one here. Accordingly, we give a new willingness function, $\varpi^s_3(u)$.

$$\varpi^s_3(u) = argmax_j \ \widehat{IO}^s_{i,j}(u) \quad (9)$$

D. Implementation

We have implemented our prototyped system on HAMA [1] (Version 0.4) which is a BSP computing framework on top of HDFS (Hadoop Distributed File System). Our proposed methods can be easily migrated to any BSP-based graph processing systems. Assume that there are K computational nodes, $N_1, N_2, \cdots, N_K$. We discuss the two main components in our prototyped system on top of HAMA, which are working workload monitoring and workload moving.

Workload Monitoring: On HAMA, in every superstep, a vertex is active if it is active already or it is inactive in the previous superstep but receives messages in the current superstep. The number of active vertices as well as the number of messages received and the number of messages sent can be collected. The workload $\mathcal{W}^s_i$ in the s-th superstep on N_i, for $1 \leq i \leq K$, will be sent to all the other computational nodes, and in the (s+1)-th superstep such collected workload assists the determination of the workload to be moved among computational nodes, $Q^{s+1}_{i,j}$, for $1 \leq i \leq K$ and $1 \leq j \leq K$. The information exchange can be done using the built-in functions, for either synchronization or broadcasting, provided by the BSP-based graph processing systems. In our implementation on HAMA, we define several subclasses under HAMA *BSPMessage* class, and exchange messages using HAMA *MessageManager*. Based on the workload, each computational node will select its active vertices to move.

Workload Moving: The quota $Q^s_{i,j}$ on N_i is known at the beginning of the s-th superstep. On a node N_i, we maintain K priority queues, where each priority queue, $P_{i,j}$, maintains the top active vertices which are willing to move to N_j from N_i up to $Q^s_{i,j}$, using *willingness score* functions defined in previous subsection. All the active vertices will appear in one queue at most. The inactive vertices do not appear in any queues. At the end of the superstep before entering the synchronization barrier, we remove the top active vertices from a priority queue $P_{i,j}$ on N_i and move them to N_j.

It is important to note that we need to trace where the vertices are when vertices are moved from one computational node to another. This is because, initially, if a computational node, N_i, maintains a vertex u in N_i which links to another vertex v not in N_i, N_i knows which computational node N_j keeps v. When a vertex v can be moved around, it needs to identify the computational node N_k that has v. If the BSP-based graph processing system has built-in functions to support transactional vertex movement to ensure the ACID property, we will use that function to move vertices. Otherwise, we can support the vertices moving by using a state-of-the-art graph partition management module such as Zephyr [9] and Lookup Table [24]. In our implementation on HAMA version 0.4, we implement the Lookup Table as used in [24] with additional catching function to reduce the overhead.

TABLE II
DATASETS

Graph	Type	$\|V\|$	$\|E\|$
CA-CondMat[3]	Collaboration	23K	186K
citeseerx[4]	Academic	6.5M	15M
cit-Patents[3]	Citation	3.8M	16.5M
dblp[5]	Academic	1M	8M
Flickr[6]	Social Networks	80K	11.8M
soc-LiveJournal1[3]	Social Networks	4.8M	70M
web-Google[3]	Web graphs	0.8M	5.9M
uk-2005[7]	Web graphs	39.5M	936M

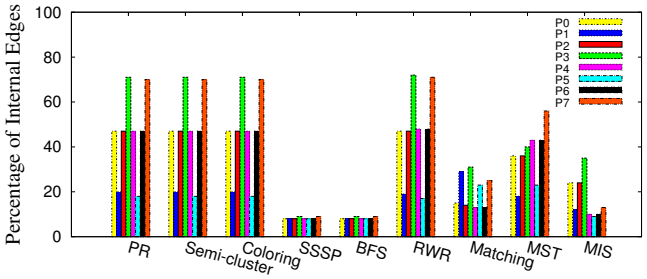
V. EXPERIMENTS

We have implemented our prototyped system on HAMA [1] (Version 0.4) on top of HDFS (Hadoop Distributed File System). We conducted extensive testing in a cloud environment with 24 PCs connected by a 100Mbps network, where each PC is equipped with a Intel i3-2100 3.1GHz CPU, 4GB memory, and a 320GB disk, running Scientific Linux release 6.1. Table II shows basic information about the graph datasets used in our experiments.

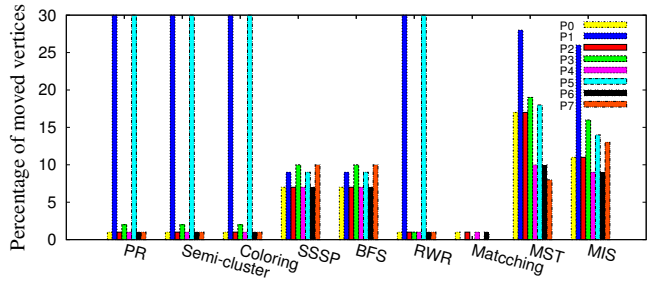
Nine Policies: In the following, we show the performance of our proposed policies based on ϖ^s_0 (Eq. (4)), ϖ^s_1 (Eq. (5)), ϖ^s_2 (Eq. (7)), and ϖ^s_3 (Eq. (9)). Here, the default β used in ϖ^s_1 is $\beta = 1$, and the default λ used for ϖ^s_2 is $\lambda = 1.05$. The 8 policies are formed as follows: P0 (ϖ^s_0), P1 (ϖ^s_3), P2 (ϖ^s_2), P3 ($\varpi^s_3 + \varpi^s_2$), P4 (ϖ^s_1), P5 ($\varpi^s_3 + \varpi^s_1$), P6 ($\varpi^s_2 + \varpi^s_1$), and P7 ($\varpi^s_3 + \varpi^s_2 + \varpi^s_1$). We tested all the policies against all the graph datasets (Table II). Initially, a graph is **randomly** partitioned into 16 partitions. Because all graphs have similar behaviors, we show the results measured in the CA-Condmat dataset.

We consider the edges that connect to the active vertices. We call an edge an active edge if it connects to at least one active vertex. Fig. 6(a) shows the percentages of the total number of active internal edges over the total number of active edges for the CA-Condmat dataset, and Fig. 6(b) shows the percentages of the total number of the moved active

[3]http://snap.stanford.edu/data/
[4]http://csxstatic.ist.psu.edu/about/data
[5]http://socialcomputing.asu.edu/datasets/Flickr
[6]http://dblp.uni-trier.de/xml/
[7]http://law.dsi.unimi.it/webdata/uk-2005/

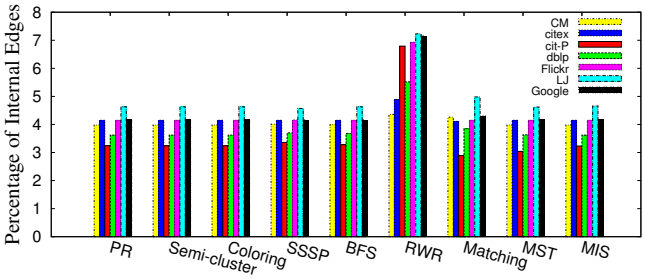


(a) The Percentage of Active Internal Edges

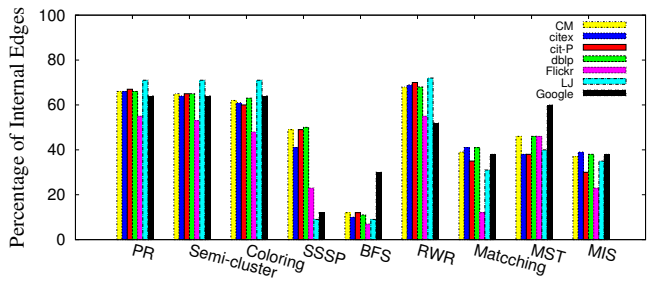


(b) The Percentage of Moved Active Vertices

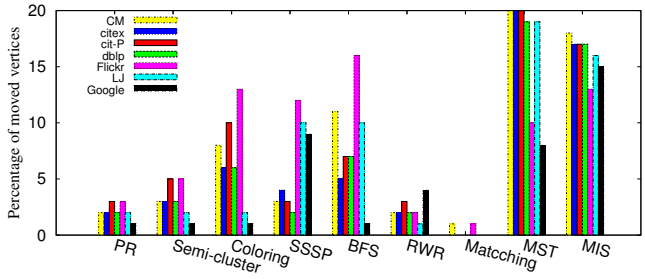
Fig. 6. Nine Policies



(a) The Percentage of Active Internal Edges (Random)



(b) The Percentage of Active Internal Edges (CatchW)



(c) The Percentage of Moved Active Vertices (CatchW)

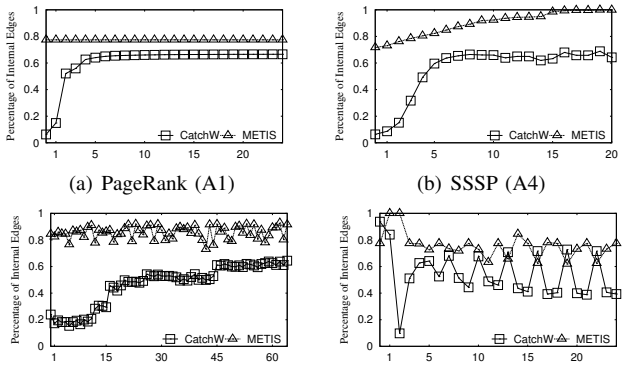
Fig. 7. Various of Graph Datasets

vertices over the total number of active vertices in the CA-Condmat dataset, for all 8 policies and all 9 graph algorithms. A higher percentage implies less cost for communication between computational nodes. As shown in Fig. 6(a), the policies, P3 and P7, outperform the other policies in most cases. In Fig. 6(b), the policy P7 moves a smaller number of vertices compared to P3. It suggests that the combination of the three $\varpi_3^s + \varpi_2^s + \varpi_1^s$ reduces the max number of external edges effectively with a small overhead. In the following, we use P7 as the default policy, and call it CatchW.

With CatchW, we further test the graph datasets. As a basis for comparison, Fig. 7(a) shows the percentages of the total number of active internal edges over the total number of active edges using the initial random partitioning without CatchW. Such percentage is very low 4-6%, which implies a high communication cost due to the large percentage of active external edges across different computational nodes. Fig. 7(b) and Fig. 7(c) show that, with CatchW, we can increase the number of active internal edges up to 70% with a small overhead (number of moved active vertices).

CatchW vs METIS: Given a set of active vertices in a superstep, it is infeasible to run METIS to partition using the active vertices due to its high overhead. In this testing, as an indicator, we compare CatchW with METIS. For the METIS, we initially partition the CA-Condmat dataset into 16 partitions using METIS, and we run METIS every superstep using the information of the active vertices. For CatchW, we randomly partition the CA-Condmat dataset into 16 partitions, and we use the default policy P7 to balance workload at run time. As expected, METIS outperforms CatchW in terms of the percentage of the total number of active internal edges over the total number of active edges. However, Fig. 8 shows that our CatchW can approach the METIS which dynamically repartition the graph with a high overhead for four representative graph algorithms, PageRank, SSSP, Random-Walk, and MST.

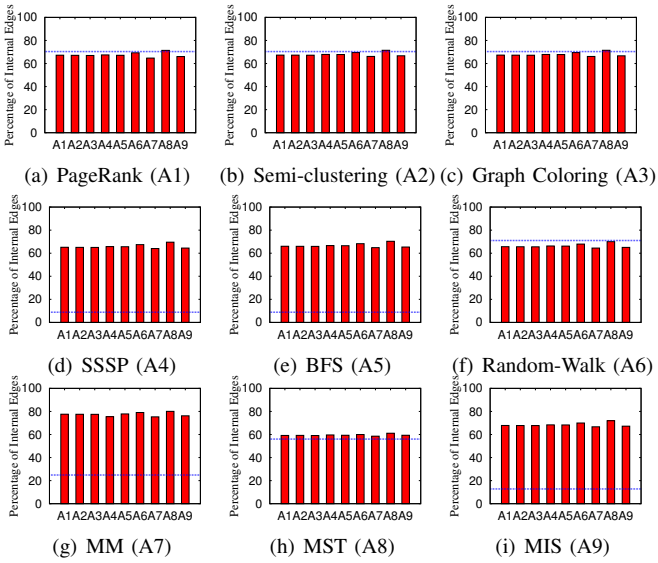
Cold-Start and Hot-Start: We test the cold-start and hot-start of different graph algorithms when we use CatchW to dynamically balance workload. All the 9 graph algorithms are named A1, A2, ... A9 (refer to Table III). Fig. 9 shows the percentages of the total number of active internal edges over the total number of active edges for the CA-Condmat dataset which is initially randomly partitioned into 16 partitions. Fig. 10 shows the percentage of the total number of the moved active vertices over the total number of active vertices. We explain our testing results. Take PageRank (A1) as an example. In Fig. 9(a), the horizontal blue line shows the percentage of the number of active internal edges when we run PageRank under the cold-start, using the initial randomly partitioned graph. Each of the 9 bars, representing the 9 graph algorithms, shows the same percentage for PageRank using the graph partition resulting from one of the 9 graph algorithms under the hot-start. In a similar fashion, in Fig. 10(a), the horizontal blue line shows the percentage of the number of active vertices moved when we run PageRank under the cold-start, using the initial randomly partitioned graph. Each of the 9 bars, representing the 9 graph algorithms, shows the same percentage for PageRank using the



(a) PageRank (A1) (b) SSSP (A4)

(c) Random-Walk (A6) (d) MST (A8)

Fig. 8. The Percentage of Internal Edges



(a) PageRank (A1) (b) Semi-clustering (A2) (c) Graph Coloring (A3)

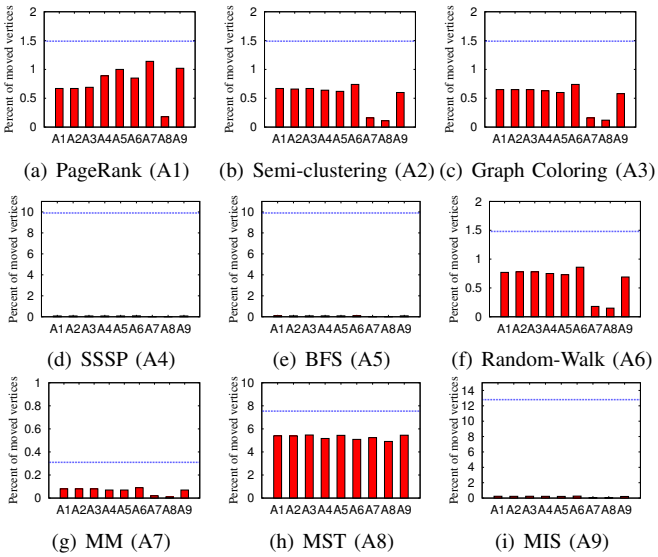
(d) SSSP (A4) (e) BFS (A5) (f) Random-Walk (A6)

(g) MM (A7) (h) MST (A8) (i) MIS (A9)

Fig. 9. Hot Starts: The Percentage of Internal Edges

graph partition resulting from one of the 9 graph algorithms under the hot-start. Fig. 9(a) and Fig. 10(a) show that the active vertices moved can help moving the workload in a right direction. In other words, with a small overhead (Fig. 10(a)), similar results can be obtained (Fig. 9(a)). The similar patterns can be seen for other algorithms as well.

Large Dataset: We show our testing results using the uk-2005 dataset, which has roughly 39M vertices and 936M edges. We randomly partition it into 24 partitions. We compare CatchW with HAMA. Fig. 11 shows the results for PageRank for every superstep. Fig. 11(a) shows the percentages of the number of active internal edges over the total number of active edges **per superstep**. CatchW can maintain over 3.4 times of active edges as internal edges over HAMA. Fig. 11(b) shows that the total execution time can reduce 31.5%, and Fig. 11(c) shows that the total communication time can reduce 42.6%. Note that communication time is included in the execution time. Fig. 12(a) and Fig. 12(b) also show the percentages of the number of active internal edges over the total number of active edges in every superstep when computing SSSP and MM, respectively. For SSSP and MM, the execution times reduce 2% and 9%.

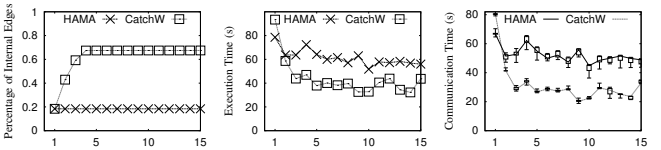


(a) PageRank (A1) (b) Semi-clustering (A2) (c) Graph Coloring (A3)

(d) SSSP (A4) (e) BFS (A5) (f) Random-Walk (A6)

(g) MM (A7) (h) MST (A8) (i) MIS (A9)

Fig. 10. Hot Starts: The Percentage of Moved Active Vertices



(a) Percentage of internal edges (b) Execution time (c) Communication time

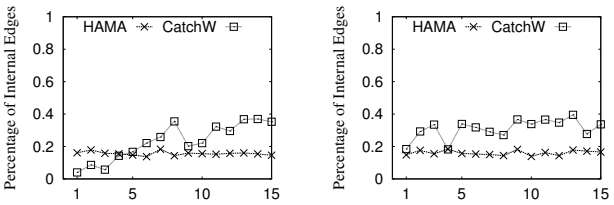
Fig. 11. PageRank (per superstep)

Scaleup: We test the scaleup using a part of uk-2005 that can be held in 8 computational nodes, and increase the number of computational nodes to 24. Our CatchW outperforms HAMA.

VI. Related Works

Graph partitioning is to partition a large graph into a number of partitions, and is widely used to support scientific simulations in a parallel environment [21]. Because graph partitioning is known to be an NP-complete problem [11], [4], many heuristics and approximate approaches are proposed. Schloegel, et al. in [21] survey static and dynamic graph partitioning methods, where the former implies that all information is given for graph partitioning, and the latter does so with only partial information. We discuss the approaches based on [21].

For the static graph partitioning techniques, there are combinatorial techniques, spectral methods, and multilevel schemes. The combinatorial techniques partition a graph based on its highly connected components, and the representative approaches are the KL (Kernighan-Lin) algorithm and the FM (Fiduccia-Mattheyses) algorithm. Spectral methods are to partition a graph based on the Laplacian for the matrix representation of a graph and computing of the second eigenvalue of the Laplacian. Multilevel schemes consist of three phases. In the first graph coarsening phase, it recursively collapses vertices/edges into a smaller coarser graph. In the second initial partitioning phase, it partitions the sufficient the coarsest graph into partitions. And in the multilevel refinement phase, it partitions a graph based on the partitioning results from the coarsest graph to the original graph in order. The representative



(a) SSSP (b) MM

Fig. 12. Percentage of internal edges per superstep

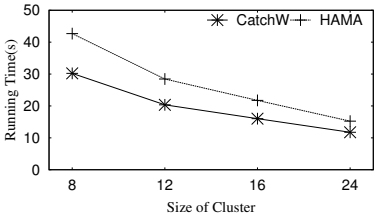


Fig. 13. Scaleup

method is METIS [12]. The qualitative comparison is given in [21]. These approaches cannot be used to address our problem for two reasons. First, it is difficult to know all the information needed for graph algorithms. Second, it is infeasible to partition a graph before processing a graph algorithm due to its high overhead.

For dynamic graph partitioning (or adaptive graph partitioning), it is to minimize the communication cost for rebalancing workloads among computational nodes. The objective function takes both the balancing and the cost to balance (the number of vertices to be moved) into consideration. There are repartitioning methods, scratch-remap methods, and diffusion-based methods. The repartitioning methods partition the graph either from the scratch or by some cut-end-phase strategy. The scratch-remap methods compute the new partitioning, and minimize the moving cost from the existing old partitions to the new partitions. The diffusion-based methods take an incremental approach and migrate workload from the overloaded computational nodes to their neighbor nodes that are underloaded repeatedly. The diffusion-based methods mainly address two questions: how much to move and what to be moved. It is worth noting that the diffusion-based methods need several iterations to achieve global balancing. In our problem setting, we have to rebalance a small amount of data simultaneously only in a single iteration.

Devine et al. in [8] survey the partitioning and load balancing for emerging parallel applications and architectures.

VII. Conclusion

In this paper, we investigate the graph behaviors by exploring the working window changes in parallel computing on a public open source implementation HAMA of Pregel for nine graph algorithms using real datasets. We show that it is possible to use immediate previous working window as a basis to catch the working window in the next superstep in Pregel like system. We propose simple yet effective policies to move small amount of active vertices in order to achieve high graph workload balancing. We confirm the effectiveness and the efficiency of our policies on a prototyped system built on top of HAMA [1].

Acknowledgment

The work was supported by grant of the Research Grants Council of the Hong Kong SAR, China No. 418512.

References

[1] http://hama.apache.org/.
[2] http://incubator.apache.org/giraph/.
[3] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High speed switch scheduling for local area networks. In *ASPLOS '92*, 1992.
[4] K. Andreev and H. Räcke. Balanced graph partitioning. In *SPAA '04*, 2004.
[5] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Math. Program.*, 73(2), 1996.
[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04*, 2004.
[8] K. D. Devine, E. G. Boman, and G. Karypis. Partitioning and load balancing for emerging parallel applications and architectures. In *Frontiers of Scientific Computing*. SIAM, 2006.
[9] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD '11*, 2011.
[10] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1), 1983.
[11] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some simplified np-complete problems. In *STOC '74*, 1974.
[12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20, 1998.
[13] N. Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1), 1992.
[14] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4), 1986.
[15] 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 *SIGMOD '10*, 2010.
[16] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD '12*, 2012.
[17] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
[18] K. Pearson. The Problem of the Random Walk. *Nature*, 72, 1905.
[19] D. Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.
[20] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: scaling online social networks. In *SIGCOMM '10*, 2010.
[21] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. Tech. Report TR 00-018, Computer Science and Engineering, University of Minnesota, 2000.
[22] B. Shao, H. Wang, and Y. Xiao. Managing and mining large graphs: systems and implementations (tutorial). In *SIGMOD '12*, 2012.
[23] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD '12*, 2012.
[24] A. L. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *ICDE '12*, 2012.
[25] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.
[26] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD '12*, 2012.