Accelerating Parallel Graph Computing with Speculation

Shuo Ji, Yinliang Zhao
Xi'an Jiaotong University
Xi'an, China
jishuo@stu.xjtu.edu.cn,zhaoy@xjtu.edu.cn

Qing Yi
University of Colorado at Colorado Springs
Colorado Springs, USA
qyi@uccs.edu

ABSTRACT

Nowadays distributed graph computing is widely used to process large amount of data on the internet. Communication overhead is a critical factor in determining the overall efficiency of graph algorithms. Through speculative prediction of the content of communications, we develop an optimization technique to significantly reduce the amount of communications needed for a class of graph algorithms. We have evaluated our optimization technique using five graph algorithms, Single-source shortest path, Connected Components, PageRank, Diameter, and Random Walk, on the Amazon EC2 clusters using different graph datasets. Our optimized implementations have reduced communication overhead by 21-93% for these algorithms, while keeping the error rates under 5%.

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**.

KEYWORDS

distributed computing, graph computing, speculative computing

ACM Reference Format:
Shuo Ji, Yinliang Zhao and Qing Yi. 2019. Accelerating Parallel Graph Computing with Speculation. In *CF '19: ACM International Conference on Computing Frontiers 2019, April 30 – May 2, 2019, CF, Sardinia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Graphs are widely used in practice to model the relationships among entities in various networks. With the rapidly increasing scale of data on the internet, distributed graph computing has become a necessity to adequately process the large amount of data [17]. A number of the parallel graph computing frameworks, e.g., Pregel[16], Giraph[1], PowerGraph[7], among others [8, 9, 29], exploit a vertex-centric programming model, which repetitively invokes a user-defined function over all vertices of a graph. Each function typically includes a computation phase, where each participating thread performs local computation for vertices it owns, and a communication phase, where data are exchanged among the neighboring vertices.

CF '19, April 30 – May 2, CF, Sardinia

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

When an algorithm is evaluated on such graph computing frameworks, all the vertices of its input graph are first partitioned, typically based on a random hash function [1, 2, 8, 16, 27], e.g., a modulus operation between a global unique identifier for each vertex and the number of partitions. Then, each partition is packaged as a task and mapped to a compute node. Due to the lack of consideration for locality when partitioning the vertices, heavy network traffic is often resulted as the tasks intensively communicate with one another[13, 19, 26]. For example, in our study of the Giraph[1] on Amazon EC2, each graph algorithm has spent 50-80% of its time in the communication phase.

Existing graph computing platforms have mostly explored two types of communication optimization: aggregation of messages to reduce communication overhead [1, 16], where messages to the same destinations are combined, and locality-aware vertex partitioning where densely connected vertices are placed on the same nodes [23, 24]. This paper presents a new optimization technique, speculative message prediction, which reduces network communications by revising the graph algorithms to predict the content of the communications, so that a network communication is triggered only when the prediction is inaccurate, where the differences between the original messages and the incorrectly-predicted ones are communicated to correct the mis-predictions.

Our optimization targets a special class of graph algorithms, where it is acceptable to sacrifice a small degree of precision for a faster turnaround time, e.g. in the case of computing shortest paths for GPS users. Therefore it targets a similar set of applications as the field of approximate computing [3, 20, 21, 28]. The key insight is that by allowing a small degree of errors in each communication, the overall runtime can be significantly reduced, not only through the reduction of the amount of network communications, but also by allowing an iterative algorithm to converge faster and terminate early. Therefore, a small degree of communication errors can be tolerated without significantly affecting the final outcome of the overall algorithm.

To illustrate our optimization, Figure 1(a) shows the original implementation of the PageRank algorithm by using Giraph[1], an iterative BSP (Bulk Synchronous Parallel)[25] graph computing system. The PageRank algorithm aims to rank the importance of a web page based on the number of links to it from other pages. In particular, the *compute* function in Figure 1(a) updates the rank of each page v by iteratively having v receiving messages about the ranks of neighboring nodes that link to v (lines 4-6), computing a new rank for v (lines 7-8), and then propagating the rank of v to all the other pages linked by v (lines 9-11). The *getSuperstep* function invoked at line 2 returns the current superstep (iteration) of the computation, which is then compared with a pre-configured maximum to determine when to terminate the algorithm, and if yes,

```
1.  void compute (Vertex vertex,  Iterable<T> messages){
2.     if  (getSuperstep() <= MAX_Superstep ) {
3.          double sum=0;
4.          for (T msg : messages) {
5.               sum += msg;
6.          }
7.          double PR = 0.15 / NumVertices() + 0.85 * sum;
8.          vertex.setValue(PR);
9.          long edges = vertex.getNumEdges();
10.         double msg = PR /edges;
11.         sendMessageToAllEdges(vertex, msg);
12.    } else {
13.         vertex.voteToHalt();
14.    }
15. }
```

(a)

```
//add class members preSum and preMsg for the class Vertex
//the definitions are private double preSum = 0; private double preMsg = 0;
1.  void specCompute (Vertex vertex,  Iterable<T> messages){
2.     if  (getSuperstep() <= MAX_Superstep ) {
3.          double sum=predictMsg(vertex.getPreSum(), getSuperstep());
4.          for (T msg : messages) {
5.               sum += msg;
6.          }
7.          vertex.setPreSum(sum);
8.          long edges = vertex.getNumEdges();
9.          double PR = 0.15 / NumVertices() + 0.85 * sum;
10.         vertex.setValue(PR);
11.         double msg = PR/edges;
12.         double pmsg = predictMsg(vertex.getPreMsg(), getSuperstep()+1);
13.         vertex.setPreMsg(msg);
14.         boolean succeed = verify(msg, pmsg);
15.         if (!succeed) {
16.           double delta = msg - pmsg;
17.           sendMessageToAllEdges(vertex, delta);
18.         }
19.    } else {
20.         vertex.voteToHalt();
21.    }
22. }
```

(b)

Figure 1: Optimizing the PageRank algorithm with speculative message prediction. (a) The original algorithm. (b) The optimized algorithm.

the algorithm is terminated by invoking the $VoteToHalt$ function at line 13 so that the vertex no longer needs to be updated.

Figure 1(b) shows our optimized algorithm, which makes two key modifications to line 11 and line 3 of the original algorithm in Figure 1(a). Line 11 of the original algorithm is modified so that instead of having each vertex sending the actual page rank, $PR/edges$, to its neighbors, the modified algorithm in Figure 1(b) sends the differences between the original message and a value speculated by invoking the $predictMsg$ function at line 12 of Figure 1(b). This difference is sent at line 17 of the modified algorithm only if the speculated message ($pmsg$) (calculated at line 12) is different from the original one (msg) at line 11 by a significant margin (determined by invoking the $verify$ function at line 14 of the algorithm in Figure 1(b)). Note while lines 4-6 in both algorithms look identical, the messages they receive are different — in the original algorithm, each vertex receives the new page ranks of its neighbors; in the modified algorithm, each vertex receives differences between the actual and predicted page ranks of its neighbors, sent when predictions are inaccurate in the remote neighbors. The remote messages are then used to correct the local prediction of a new page rank for each vertex v, made at line 3 of the modified algorithm, by adding (on top of the local predicted page rank) the differences received from all the neighboring vertices of v.

The optimized algorithm in Figure 1(b) is correct only if the prediction function $predictMsg$, invoked at line 3 and line 12 respectively, are consistent with each other, in that the local prediction for the new page rank of each vertex v at line 3 equals to the the sum of predicted page ranks of all neighbors of v (predicted at line 12), so that any local mis-prediction can be corrected by simply adding on top of it the messages sent from all the neighbors. This constraint is readily satisfied by making the prediction function distributive to the addition operator, the operator used to relate the page rank of each vertex v with those of its incoming neighbors in the original algorithm. The same optimization technique can be readily applied to a number of other popular graph algorithms, e.g., Connected Components and Single-source shortest path. The optimization is profitable only if the content of the communications is predictable, i.e., the prediction function is correct for a non-trivial number of times. Since no communication is needed when the predictions are sufficiently accurate, the overall amount of communications can be significantly reduced.

We have expanded three computing frameworks, Giraph[1], PowerGraph[7], and PowerLyra[4] to support the $predictMsg$ and $verify$ functions illustrated in Figure 1(b) and have validated our optimization by using the extended model to implement five algorithms, Single-source shortest path (SSSP), Connected Components (CC), PageRank, Diameter, and Random Walk (RW). When evaluated on Amazon EC2 Cloud, our optimized implementations have consistently out-performed their original implementations over a variety of data sets. Our main contributions include:

- We introduce a new speculation-based optimization approach to reduce the communication overhead and reason about the correctness and error bounds of the optimization when applied to commonly used graph algorithms. We experimentally study the communication overhead of these graph algorithms implemented on top of three graph computing platforms: Giraph[1], PowerGraph[7], and PowerLyra[4].
- We develop two approaches to predict the content of messages based on their histories and apply the techniques to optimize the graph algorithms. We were able to significantly speed up these algorithms compared to using their original implementations.

The rest of the paper is organized as follows. Section 2 reasons about the correctness and generality of our optimization in more detail. Section 3 presents the prediction methods. Section 4 presents experimental results. Section 5 presents the related work. Finally, conclusions are shown in Section 6.

2 OVERVIEW OF OPTIMIZATION

Figure 2(a) shows the typical workflow of an algorithm expressed using the vertex-centric computation model. In particular, an algorithm is typically expressed by defining a function, say $compute(v, msg)$, where v is an arbitrary vertex of the graph, and msg is the set of messages sent from other vertices to v in the previous iteration.

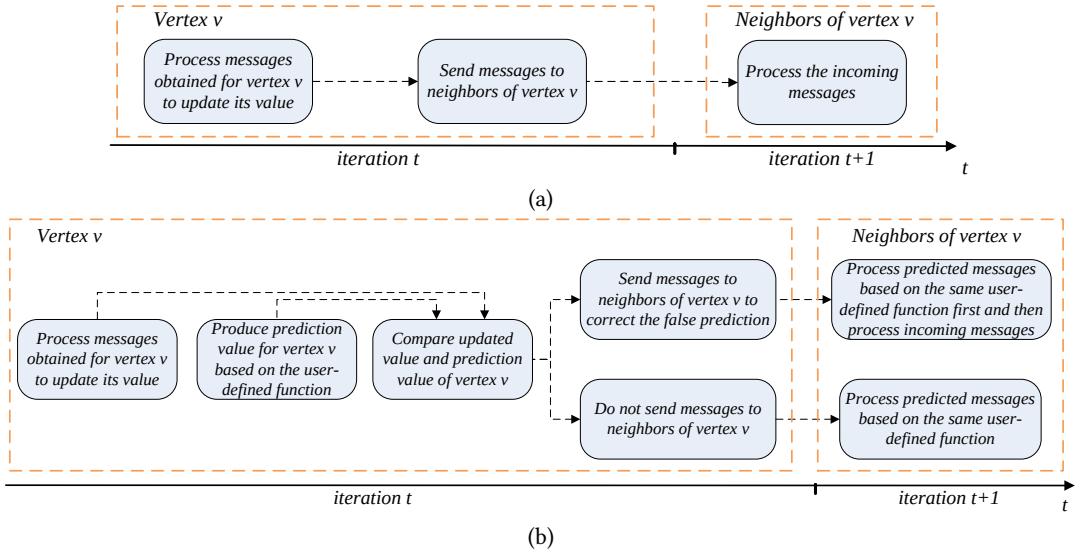

Figure 2: (a)Workflow of the original implementation. (b)Workflow of the optimized implementation.

Figure 1(a) illustrates such a function for the PageRank algorithm. Each time this function is invoked, it essentially computes a new value for a given vertex v, based on the old value of v and a set of updates sent from the other vertices. Our revised workflow is shown in Figure 2(b), where before sending each message from a vertex v, a *predictMsg* function is invoked to predict the content of the message. The predicted content is then compared with the message to be sent. If their differences are within a pre-specified allowable error rate, the communication is omitted. All the graph computing framework we use automatically support the needed communications among operations on the vertices, where the user only has to specify the content and destinations of each message to be sent, and all messages are automatically delivered without requiring anything from the receiving end. The following first uses the PageRank algorithm as an example to reason about correctness constraints of our algorithmic optimization. Then, additional properties of the optimization are studied after classifying a variety of other graph algorithms.

2.1 Example: The PageRank Algorithm

Given an arbitrary graph $G = (V, E)$, where V and E are the vertices and edges of the graph respectively, PageRank in Figure 1(a) repetitively computes the following two functions for each vertex v until they converge.

$$value_i(v) = 0.85 * msg_{i-1}(v) + 0.15/|V| \tag{1}$$

$$msg_i(v) = \Sigma_{\forall u \in Vs.t.(u,v)\in E}(value_i(u)/|Out_u|) \tag{2}$$

Here $value_i(v)$ is the new value for the vertex v at the ith iteration, $|V|$ is the overall number of vertices in the graph, $msg_i(v)$ and $msg_{i-1}(v)$ are the sums of all messages received by v at the ith and $i-1$th iteration respectively (computed at line 4-6 of Figure 1(a)), and for each vertex u that is connected to v in the graph, $value_i(u)$ and $|Out_u|$ are the new value computed for u at the ith iteration and the number of edges going out from u respectively.

Our optimized code in Figure 1(b) modifies the PageRank algorithm in Figure 1(a) by introducing a new function, *predictMsg*, and by invoking the $verify(x, y)$ at line 14 to conditionally skip communications where the prediction is successful. Suppose $verify(x, y)$ returns true only if $x = y$. The computation becomes:

$$\begin{aligned} value_i(v) =& 0.85 * (predictMsg(sum_{i-1}(v), i)+ \\ & msg_{i-1}(v)) + 0.15/|V| \end{aligned} \tag{3}$$

$$sum_i(v) = predictMsg(sum_{i-1}(v), i) + msg_{i-1}(v) \tag{4}$$

$$\begin{aligned} msg_i(v) =& \Sigma_{\forall u \in Vs.t.(u,v)\in E}(value_i(u)/|Out_u|- \\ & predictMsg(value_{i-1}(u)/|Out_u|, i+1)) \end{aligned} \tag{5}$$

Here both $value_i(v)$ and $msg_i(v)$ have been modified to incorporate the results of a series of *predictMsg* invocations as new factors in the calculations. The prediction baseline used in calculating $value_i(v)$ is a recursively defined function $sum_i(v)$, which represents the accumulated page ranks of all incoming neighbors of v. On the other hand, the prediction baseline for calculating $msg_i(v)$ is simply the scaled-down value of most recent page rank of each individual incoming neighbor of v.

In order for the modified algorithm to compute the same $value_i(v)$ as the original algorithm for each vertex v, the following equation needs to hold.

$$\begin{aligned} \Sigma_{\forall u \in Vs.t.(u,v)\in E} & value_{i-1}(u)/|Out_u| = \\ & predictMsg(sum_{i-1}(v), i)+ \\ & \Sigma_{\forall u \in Vs.t.(u,v)\in E}(value_{i-1}(u)/|Out_u|- \\ & predictMsg(value_{i-2}(u)/|Out_u|, i)) \end{aligned} \tag{6}$$

The above equation can be simplified to

$$\begin{aligned} & predictMsg(sum_{i-1}(v), i) = \\ & \Sigma_{\forall u \in Vs.t.(u,v)\in E}(predictMsg(value_{i-2}(u)/|Out_u|, i)) \end{aligned} \tag{7}$$

The above equation requires that the *predictMsg* function can be distributed across the addition (Σ) operator. In this case, the equation becomes

$$sum_{i-1}(v) = \Sigma_{\forall u \in V s.t.(u,v) \in E}(value_{i-2}(u)/|Out_u|) \quad (8)$$

The above equation can be proved via induction on the variable i, which represents the iteration index where the *compute* function is invoked. Therefore, as long as the *predictMsg* function can be distributed across the addition (Σ) operator, and the *veriy* function returns true only if its given parameters are identical, the optimized code is guaranteed to compute the same page ranks for all vertices as the original algorithm. In our optimized algorithm, we have implemented the function $predictMsg(x, i)$ so that given an arbitrary floating point number x and a designated iteration i, the function returns $r(i) * x$, where $r(i)$ is a function that maps each iteration i to a floating point number that is ≥ 1. So each prediction essentially scales up the value of the previous iteration by some factor, since the page rank of each vertex is expected to continuously increase as more iterations are completed.

Suppose the $verify(x, y)$ function allows approximation, in that it returns true as long as $|x - y| \leq \alpha * x$, where α is an allowed error ratio. Then a messages is sent only if the result of the prediction differs from the correct value x by at least $\alpha * x$. Suppose for the first time at some iteration i, $verify(msg, pmsg)$ returns true when $msg \neq pmsg$ for some neighbors of a vertex v. Then the value of $sum_{i+1}(v)$ computed via Equation (4) at iteration $i + 1$ can be off by at most $\alpha * sum_i(v)$. So $value_{i+1}(v)$ can in the worst case differ from the correct value by $0.85 * \alpha * sum_i(v)| \leq \alpha * value_i(v)$. Therefore the error ratio accumulates (compounds) by at most α each time the computation proceeds to the next iteration. Suppose the computation iterates N times, the compounding of the error ratios per vertex can be calculated in the worst case as $N * \alpha + (N - 1) * \alpha^2 + ...2 * \alpha^{N-1} + \alpha^N$. The whole application error rate therefore can be bounded in the worst cases. Note that the actual error rates at runtime will vary when different input graphs are processed. It is beneficial to be able to dynamically predict the expected error rate as each input graph is processed. However, this is currently outside the scope of this paper and belongs to our future work.

2.2 Extending To Other Graph Algorithms

In the context of vertex-centric graph computing, our optimization can be applied to all graph algorithms that require communications among vertices, where the patterns and content of the communications are predictable, and mis-predictions can be corrected by simply re-integrating their mis-predicted portions. In particular, the predictability of communication, both in terms of the participating vertices and in terms of the content exchanged among the vertices, is key to both the applicability and profitability of the optimization. The requirement on the ease of re-integrating mis-predictions has more implications in terms of the profitability of the optimization than applicability, because an operation can always be defined, so that the original content of communications is simply recovered, and the progress of the algorithm rolls back if needed. Such recovery mechanisms, however, are likely too expensive and should be used only if a very high prediction accuracy can be assured.

To study how frequently graph algorithms can and are likely to benefit from our optimization, Table 1 classifies 11 widely used graph algorithms that we have collected from various sources[7, 21], based on their runtime behavior relative to five properties related to whether they can potentially benefit from our optimization. Currently we were able to optimize only five out of these 11 algorithms, and our current selection criteria requires that the algorithm must take a non-trivial number of iterations (A=1), each vertex of the graph must always communicates with the same partners (B=1), the content of communication must be predictable (D=1), and the differences of communication content must be differentiable and re-integratable (E=1). Repeating Communications between each pair of vertices (C=1) turns out to be not required as long as the messages are predictable (D=1). The most stringent requirement is the predictability of the communications (column D), as the other requirements can be potentially met by most graph algorithms, especially after some manual modifications to their original implementations. The following briefly describes each of the other four algorithms that we optimized besides PageRank.

Table 1: Graph algorithms

Algorithm	Abbr.	A	B	C	D	E
Bipartite Matching	BM	1	0	1	0	0
Connected Components*	CC	1	1	0	1	1
Diameter*	D	1	1	1	1	1
Graph coloring	Gc	1	1	1	0	1
K-core	Kc	1	1	0	0	0
KMeans	KM	1	1	1	0	1
Label Propagation	LP	1	1	1	0	1
PageRank*	PR	1	1	0	1	1
Random Walk*	RW	1	1	1	1	1
Single-source Shortest Path*	SSSP	1	1	0	1	1
Triangle Counting	TC	0	1	0	0	0

[1] A: whether the algorithm iterates a non-trivial number of times; B: whether each vertex always communicates with the same partners; C: whether repeated communications are needed among vertices; D: whether the content of communication has regularity (can be predicted). E: whether the differences of communication content can be extracted and re-integrated.

[2] 1 vs 0 indicate yes vs no.

The Single-source shortest path (SSSP) algorithm aims to compute the shortest distance from a pre-designated vertex s to v. The original algorithm starts by first initializing the shortest distance for each vertex v to be a maximum value and then iteratively updates the value for each vertex by computing the minimum of all the paths to v from incoming neighboring nodes of v. Since the shortest distance recorded for each vertex can only become smaller at each iteration, and each vertex v always communicates with the same partners (its neighbors), the communication pattern of the algorithm is relatively regular and predictable. The Connected Components (CC) algorithm aims to identify vertices that are connected with one another in a graph. The original algorithm starts by initially assigning to each vertex a unique integer identifier and then repetitively compute the minimum integer identifier among its neighbors. The logic of CC algorithm is quite similar to that of the SSSP algorithm. We can optimize both SSSP and CC by having each vertex first predict the minimum among its incoming messages and then use the minimum operator to revise the predicted value based on incoming messages from its neighbors, which send the differences between their respective updated and predicted values, omitting communications when the differences are minor.

The Diameter algorithm aims to compute the effective diameter of a graph G in which 90% of all connected pairs of nodes can reach each other) [12], by iteratively computing a function $N(v, i)$, which counts the number of neighbors of vertex v in $\leq i$ hops. In particular, k bitstrings [18] $b(i, v)$ are encoded to remember vertices reachable to v within distance i (i.e., the neighborhood for each vertex v), and are iteratively updated by computing the union(OR) between its previously updated neighborhood and each received neighborhood from its neighboring vertices until it converges or reach the predefined number of iterations. Then $N(i, v)$ is estimated by using the Flajolet-Martin [18] equation with the maintained bitstrings of each vertex. As the original computation proceeds, the bit-masks for some vertices will stay the same across the iterations. Therefore by predicting the bit-mask of each vertex does not change, a subset of the communications can be omitted. Since the bit-masks do not admit a reasonable definition of approximation, a communication is omitted only when the prediction is 100% accurate.

The Random Walk (RW) algorithm [11] tries to simulate a stochastic or random process, by constructing a path that consists of a succession of random steps on an given input graph. Since each vertex v randomly selects its neighboring nodes to send messages, the communication pattern of RW is relatively obscure. However, the algorithm demonstrates a varying level of repeatability among successive communications when having a large number of random walks being computed in parallel. It is also relatively easy to extract and re-integrate the differences of the communication content. The Random Walk algorithm is quite similar to PageRank (both use the addition operator to combine incoming messages for each vertex), except that the communication content is randomly decided in RW. It can be optimized in the same fashion as PageRank, except that we disabled approximation in the optimized algorithm to avoid chain reactions in the randomly selected values.

Among the other graph algorithms that we were not able to optimize in Table 1, Bipartite Matching have each vertex communicate with a randomly selected neighbor in each iteration. In this case, communications are hard to predict because the successive messages are sent to randomly selected neighbors. Triangle Counting has only two iterations in its entire computation, and only in one of the iterations vertices communicate with each other. The rest of the algorithms, Graph coloring, K-core, KMeans and Label Propagation, assign special-purpose labels as values to their vertices, in fashions that are hard to predict due to the lack of regularities.

2.3 Correctness And Error Compounding

Among the five algorithms we optimized, PageRank and Random Walk both use addition to combine incoming communication messages, while SSSP and CC both use the minimum operator, and the OR operator is used in the Diameter algorithm. The correctness of these algorithms can be similarly proved as PageRank under the constraint that the *predictMsg* function can be distributed across the respective operators.

Since each algorithm may demand its own acceptable level of accuracy, our optimization allows the user to control what levels of approximation are allowed to trace off the accuracy of a distributed graph computing algorithm with its runtime communication cost and efficiency, by defining the $verify$ function invoked at line 14 of Figure 1 differently. In particular, the user can allow no approximation by setting the $verify$ function to returns true only when its to input parameters are identical. The user can also allow a per-message error rate by allowing the two input messages to differ by a pre-set percentage, which will be compounded as multiple iterations of the algorithm are evaluated. The error rates of SSSP and CC are compounded similarly to PageRank, except that the eventual error rate for each vertex equals to the minimal of the error rates across all iterations, so their overall error rates are much smaller than that of PageRank. For Diameter and Random Walk, the compounding of the errors is rather unpredictable. So we conservatively disabled approximation in these algorithms are not acceptable, by setting the $verify$ function for them to return true only when the prediction is 100% accurate.

For the purpose of this paper, we define the overall error rate of the whole algorithm to be the average or the maximum of the percentages of differences between the expected and final computed values for each vertex. Since approximations are allowed in three of the five algorithms we studied, Figure 3 shows the average and maximum error rate per vertex for the algorithms (SSSP, CC, and PageRank) when the max allowed error ratio per communication is 0.5%. As the computation proceeds through each iteration, the average error rate at each iteration i is computed as:

$$R_i(u, v) = \frac{\sum_{k=1}^{n} |u_i(k) - v_i(k)|}{\sum_{k=1}^{n} |v_i(k)|} \tag{9}$$

where u and v are vectors saving the values of all the vertices (k=1,...,n) by using and without using message prediction respectively. The maximal error rate among all vertices at each iteration i is computed as:

$$R_{max}(u, v) = max\{\frac{|u_i(k) - v_i(k)|}{|v_i(k)|}, u_i(k) \in u, v_i(k) \in v\} \tag{10}$$

These results confirm our worst-case error compounding calculations and show that in average, the error rates are much smaller than the worst case. Since the number of expected iterations can be anticipated, the allowed per-message error rate can be configured to be a sufficiently small percentage to bound the allowed whole application error ratio.

3 PREDICTION METHODS

For our optimization to be beneficial, the time spent in predicting the content of a message must take significantly less time than the time to actually transmit the message among different tasks, either across the network or within a single node. Since it is more difficult to enhance the accuracy of predictions than to simply reduce the runtime overhead of the prediction method, a prediction method with a smaller runtime overhead can improve the overall performance with a similarly lower prediction accuracy. We have developed two message prediction strategies, a memory-based method and a curve-fitting method, both are simplistic (fast) and therefore require a low prediction accuracy to outweigh the runtime overhead. For the purpose of this paper, we define prediction accuracy as the percentage of the successfully predicted messages among all the messages transferred among vertices.

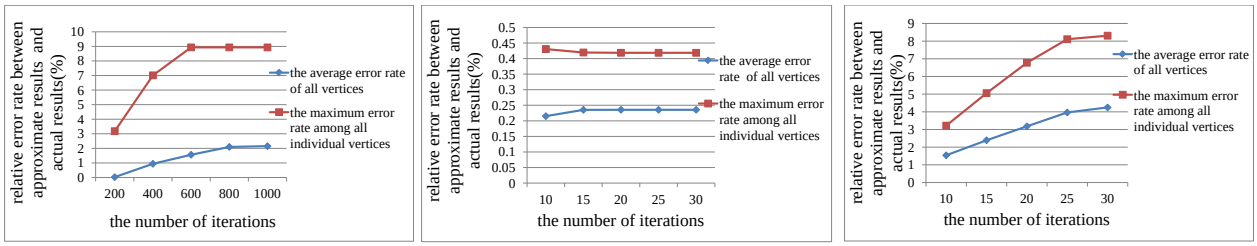

Figure 3: Compounding of error rates (the max allowed per message error ratio is 0.5%). (a) SSSP on USA-road[5]. (b) CC on web-google[22]. (c) PageRank on web-google[22].

3.1 Memory-based method

The memory-based approach is designed for graph algorithms that are similar to SSSP, CC, and Diameter, which successively modify the value of each vertex as a monotonic function of the values of the incoming messages, e.g., by taking the minimum of the incoming messages in SSSP and CC, or taking OR operator in Diameter. Given two input parameters, x, the value in the previous iteration, and i, the new iteration to predict, this method simply returns x as the new value for i, essentially predicting value will stay the same across iterations.

Table 2 shows the resulting runtime (normalized to the execution time of the respective original implementations) and prediction accuracy (calculated as the percentage of the successfully predicted messages among all the messages transferred among vertices) of this prediction method, when using it for SSSP, CC and Diameter on different input datasets. The prediction method has the lowest prediction accuracy, 5.56%, when used to predict the content of messages for the Connected Component algorithm using the live-journal[14] input dataset, where the revised algorithm shown slight superiority in the runtime compared to the original algorithm without prediction. A higher prediction accuracy in the other use cases had uniformly resulted in better performances.

Table 2: Prediction accuracy of memory-based method

Algorithm	Dataset	Runtime	Accuracy
SSSP	USA-road-W	37.85%	86.05%
SSSP	USA-road	8.89%	94.59%
CC	live-journal	95.35%	5.56%
CC	indochina	84.0%	38.66%
Diameter	web-google	86.05%	25.4%
Diameter	wiki-topcats	86.89%	21.21%

3.2 Curve fitting method

The curve-fitting method is used for the PageRank algorithm, where the value of each vertex is a non-obvious function of its previous values. Given two input parameters, x, the value in the previous iteration, and i, the new iteration to predict, this curve-fitting method predicts that x will change in the similar ways as in the past. It uses curve fitting, where a curve is constructed as a mathematical function that has the best fit to a series of data points, to model the most likely content of the next value for x.

In PageRank, the algorithm iteratively updates the value for each vertex by increasing its rank as needed at each iteration. The new value of each vertex, after being divided by the out-degree of the vertex, is then communicated to the neighboring vertices for additional updates. As more iterations are completed, the results are expected to eventually converge, so that the ranks of the vertices would slowly increase, and the content of the messages approximate to those of the previous iteration. Based on understanding of the algorithm, we used the following exponential function to predict the values at different iterations for PageRank.

$$PR_{sp}\left(v^t\right) = \left(1 + \alpha * \beta^t\right) * PR\left(v^{t-1}\right) \quad \beta < 1 \tag{11}$$

Here, α and β are parameters to be optimized. Based on our experimental results of using different values for these variables to predict values at different iterations for PageRank on a small graph dataset, we have selected α to be 0.1 and β to be 0.5. Table 3 shows the resulting runtime (normalized to the execution time of the respective original implementations) and prediction accuracy of the implementation. Here because the runtime overhead of the curve-fitting method is higher than the memory-based method, a higher accuracy is required for the prediction to improve application performances.

Table 3: Prediction accuracy of curve-fitting method

Algorithm	Dataset	Runtime	Accuracy
PageRank	web-google	80.56%	57.89%
PageRank	uk-2002	39.13%	63.16%

3.3 Combination of memory-based and curve-fitting

This strategy is used for the Random Walk algorithm [11], which implements a stochastic process to traverse a path via a succession of random steps on an given graph, as a very special case of a Markov chain. It is studied and validated by having a large number of random walks being computed in parallel. When using web-google as the input graph, the algorithm demonstrated varying levels of repeatability among the successive communications. To approximate such patterns, we used a combination of the memory-based and the curve-fitting method. In particular, the memory-based method is used repetitively to predict the content of messages for a few successive iterations, and a curve-fitting method is used intermittently

to change the saved content of messages when the memory-based prediction leads to a wrong prediction. After using the curve-fitting method to model the next most likely content, the memory-based method is again used. The curve-fitting method is adaptively adjusted by gradually increasing or decreasing the previously saved message, based on how the previous messages change across iterations. Table 4 shows the resulting execution time (normalized to the execution time of the respective original implementations) and prediction accuracy of the implementation. Here because the runtime overhead of the combination method is also a little higher than the memory-based method, a higher accuracy is required for the prediction to improve application performances.

Table 4: Prediction accuracy of combination method

Algorithm	Dataset	Runtime	Accuracy
RandomWalk	web-google	75.3%	26.94%
RandomWalk	live-journal	76.78%	28.9%

4 EXPERIMENTAL EVALUATION

We have implemented five graph algorithms, SSSP, CC, PageRank, Diameter and Random Walk, on three open-source computing frameworks, Giraph[1], PowerGraph[7], and PowerLyra[4], both with and without our optimization. While Giraph supports bulk synchronous communication, both PowerGraph and PowerLyra are more recent frameworks that support asynchronous communications [4, 7] with more advanced optimizations. The input data sets shown in Table 5, which represent a variety of different scales of graphs, are used to study study the effectiveness of our optimization, particularly its implications to the overall performance, communications behavior, and the accuracy of the graph algorithms, Additional graphs from [14], including uk-2014, in-2014, frwiki, dewiki and uk-2002, are used to validate the effectiveness of the curve-fitting prediction designed for PageRank.

Table 5: Graph data in our experiments

Dataset	$\|V\|$	$\|E\|$
USA-road-W[5]	6,262,104	15,248,146
USA-road[5]	23,947,347	58,333,344
frwiki[14]	1,352,053	34,378,431
indochina[14]	7,414,866	194,109,311
web-google[22]	916,428	5,105,039
uk-2002[14]	18,520,486	298,113,762
wiki-topcats[22]	1,791,489	28,511,807
live-journal[14]	5,363,260	7,902,314

We conducted all studies on the Amazon EC2 Cloud with 8 instances, each instance using two dual-core CPUs and 8G memory. Each algorithm is evaluated with the number of computing threads equal to the number of available CPU cores on each node of the cluster. Each node is configured with Ubuntu Server 14.04(64bit) and Apache Hadoop 0.20.203 and Java 1.7. Each measurement is conducted five times, and the averages across different runs are reported. what is the variation across runs of the same measurement?

4.1 Communication overhead and runtime performance

Figure 4 compares the amount of inter-vertex communications (the cumulative number of bytes of all messages communicated) for the algorithms implemented with and without our optimization on the three platforms when using graphs from Table 5 as their inputs and allowing a maximum 0.5% per message error rate is this correct?. Figure 5 compares the respective runtime of these algorithms on the same inputs. Less communications are observed for Diameter and Random Work in Figure 4(d) and (e) because they used smaller input graphs and have converged faster. Why do they use smaller graphs and converge faster? On all platforms, the optimization has significantly reduced the amount of communication for SSSP, which had accordingly resulted in the most significant performance speedups in Figure 5(a). The communication and execution time reductions for CC, PageRank, Diameter and Random Walk are not as significant, due to lower prediction accuracies. In particular, the amount of communication for SSSP is reduced by 91-94% on the three platforms, because the *predictMsg* function was able to correctly predict that the shortest distance does not change for many vertices at each iteration. On the other hand, the reductions for the other algorithms are 21-40%, as their communication patterns are less regular (harder to predict) Since Giraph produced the most communication for all algorithms, greater execution time reductions are observed on the Giraph platform. Allowing approximation in SSSP, CC, and PageRank has also allowed the respective algorithms to converge faster and therefore requiring fewer iterations to converge, resulting in more significant performance speedups.

To further break down the impact of communication reduction in terms of network latencies vs system overheads, Figure 6 compares the runtime of executing the graph algorithms with and without our optimization on PowerGraph and PowerLyra, by using only a single compute node, where the communication overhead includes only system overheads and no network latency. Here the runtime reduction is still significant, even when no network latency is involved in the communications.

4.2 Error Analysis

Since three of our optimized algorithms (SSSP, CC, and PageRank) allow the eventual results of computation to be approximated, for each of these algorithms, we computed an average per-vertex error rate on the Giraph platform, as the differences in the final results of all vertices computed by the optimized algorithm, normalized against their final results in the original algorithm. This is a common formulation for error analysis [21]. For SSSP and PageRank, the error rate reflects in average the expected degree of approximation for the final computed results. For CC, it reflects how likely a connected component is mistakenly divided into smaller ones, Note that our optimization can be used even when approximation is not allowed, as in many cases of realistic applications, by having the $verify$ function to return true only when the prediction is 100% accurate.

Table 6 shows the resulting relative error rates, which range between 0.02%-5%. As a case study, we looked deeper into the results of the PageRank algorithm using web-google as input. We found

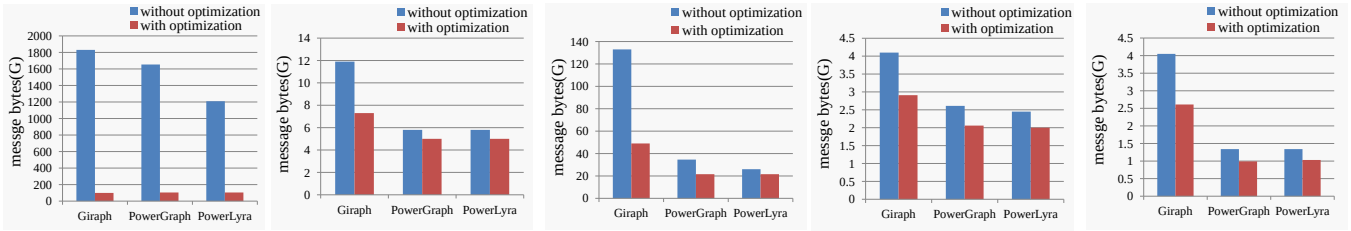

Figure 4: Comparison of communication overhead between without and with using optimization. (a) SSSP on USA-road. (b) CC on indochina. (c) PageRank on uk-2002. (d) Diameter on wiki-topcats. (e) Random Walk on live-journal.

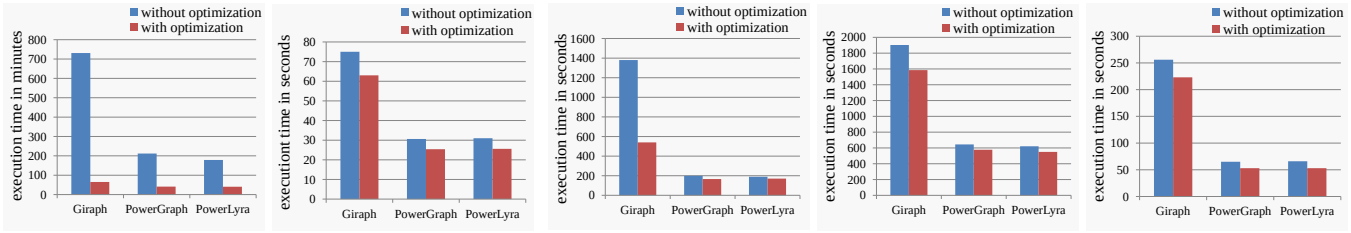

Figure 5: Comparison of runtime between without and with using optimization. (a) SSSP on USA-road. (b) CC on indochina. (c) PageRank on uk-2002. (d) Diameter on wiki-topcats. (e) Random Walk on live-journal.

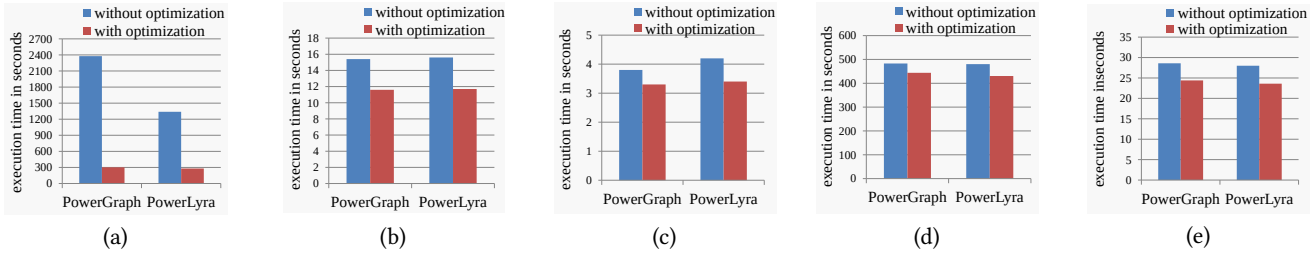

Figure 6: Comparison of runtime between without and with optimization on 1-computing node. (a) SSSP on USA-road-W. (b) CC on frwiki. (c) PageRank on web-google.(d) Diameter on web-google. (e) Random Walk on web-google.

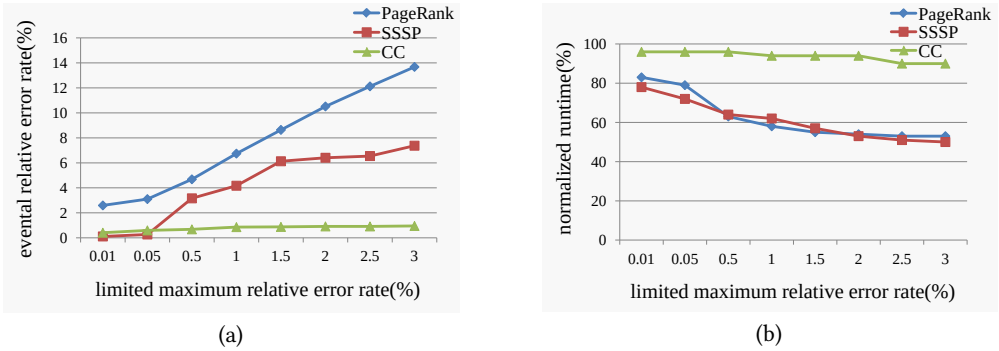

Figure 7: (a) Relationship between the limited per-message error rate and the whole application error. (b) Relationship between the limited per-message error rate and the runtime normalized to the runtime of the original implementation.

Table 6: Relative errors for applications implemented with our optimization

Application	Dataset	Relative error(%)
SSSP	USA-road-W	0.8772
SSSP	USA-road	1.6734
CC	frwiki	1.0618
CC	indochina	1.1923
PageRank	web-google	4.3483
PageRank	uk-2002	4.6043

that 99% of the pages in top 100 computed using the original algorithm also appear in the top 100 from using the revised algorithm and that 98.8% of the pages in top 1000 from the original algorithm also appear in the top 1000 from the revised algorithm. Overall, the results obtained via the optimized algorithms are expected to be close approximations of the original results, trading off a limited amount of overall accuracy for the benefit of better performance.

Figure 7(a) correlates the allowed per-message error rates with the eventual average per-vertex error rates, for SSSP, CC and PageRank operating on USA-road-NY[5], frwiki[14] and uk-2014[14] respectively. Here it can be observed that the eventual per-vertex error rate roughly correlates linearly with the per-message error rate, with the correlation ratio determined by how the per-message errors accumulate/compound at each global iteration of the algorithms. In particular, for CC and SSSP, the per-vertex error rates are the minimum of the errors of all the incoming messages, so their correlation ratios are lower than that of PageRank, where the per-vertex error equals to the sum of the errors of all the incoming messages.

Figure 7(b) further correlates the allowed per-message error rates with the runtime of the algorithms. Here again the relation is roughly linear, and a significant runtime reduction is observed even when allowing a small per-message error rate, say 0.5%. This is because that as the per-message allowed error rate increases, more communications can be potentially omitted as a bigger differences is allowed between the predicted and the actual messages, thereby reducing the amount of communication and overall runtime.

4.3 Effectiveness of curve-fitting method

A curve-fitting function was used to predict the growth of vertex values in our optimized PageRank algorithm. Figure 8 validates the effectiveness of this method when using a single version of the optimized algorithm to operate on a variety of different inputs, including uk-2014, in-2014, frwiki, dewiki and uk-2002[14], with the size of input data ranging from 250MB to 4.7GB. Here a single set of curve-fitting parameters, $\alpha = 0.1$ and $\beta = 0.5$, are used across all inputs. The overall runtime of the algorithm is reduced by 37%, 32%, 39%, 37% and 62% respectively while keeping the relative error rate under 6.1%. These results validate our hypothesis that for PageRank, while the accuracy of curve-fitting prediction will differ when used on different input data, the patterns of vertex updates are sufficiently similar across different inputs, so that reasonable performance improvements can be expected across all inputs.

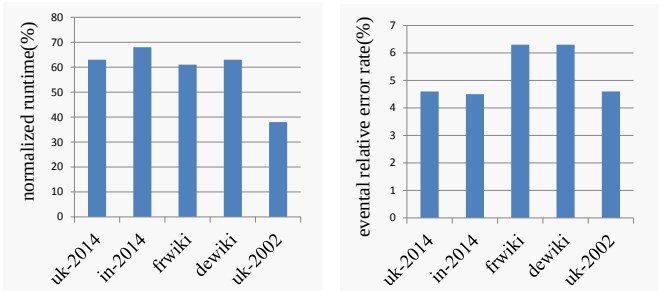

Figure 8: (a) Execution time normalized to the execution time of the original implementation without prediction. (b) Eventual relative error rate between the approximate results and the accurate results.

5 RELATED WORK

Pregel[16] is the first distributed parallel graph computing system using the Bulk Synchronous Parallel (BSP) model[25]. Apache Giraph[1] is developed as an open-source version of Pregel and is used by Facebook to analyze social networks. Maiter[29] introduces a delta-based accumulative iterative computation model, which iteratively updates values of vertices by accumulating the changes between iterations to avoid the negligible updates. GraphLab[15] is designed for asynchronous parallel graph computations in machine learning and allows the vertices to be processed asynchronously based on a scheduler. PowerGraph[7] introduces the concept of vertex-cuts, where the edges instead of vertices of an input graph are distributed to reduce the storage and communication requirement of large distributed power-law graphs. PowerLyra[4] provides a hybrid graph partitioning that combines edge-cut and vertex-cut and a hybrid graph computation that combines the synchronous and asynchronous execution.

A large collection of graph partitioning strategies[13, 19, 23, 24, 26] have been introduced to reduce communication overheads of distributed graph systems. Streaming partitioning[23, 24] is an online approach that dynamically places each vertex as it is loaded from storage based on the locations of its neighbors. Dynamic repartitioning[13, 19, 26] provides a way for the partitioning to self-adapt to balance computation, by monitoring the runtime characteristics of the system and migrating the vertices across workers during graph processing. This paper also aims to reduce the communication overhead of graph systems but does so by allowing the user to define a way to predict the content of transmitted data.

Speculative prediction of values is widely studied in paralleling algorithms. Gardner[6] modifies the particle swarm optimization(PSO) algorithm by speculatively concurrently evaluating the t+1th time-step with all possible results of iteration t. Jang[10] enhances variable-length decompression by speculating on the starting points of compressed blocks.

6 CONCLUSIONS

This paper presents a new optimization to reduce the communication overhead of vertex-centric distributed graph algorithms. The

optimization has been applied to five graph algorithms, Single-source shortest path, Connected Components, PageRank, Diameter and Random Walk, on three graph computing frameworks, Giraph[1], PowerGraph[7], and PowerLyra[4]. The performance of these algorithms with and without the optimization is systematically studied on Amazon EC2 Cloud, where our optimization has reduced the communication overhead by 39% and the execution time by 27% in average, while keeping the error rate under 5%.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China through grants No. 61640219 and by the National Science Foundation of USA through CCF-1421443.

REFERENCES

[1] Apache. 2012. Apache Giraph. http://giraph.apache.org/.
[2] Apache. 2012. Apache Hama. https://www.hama.com/.
[3] Ziv Baryossef and Lital Mashiach. 2008. Local approximation of PageRank and reverse PageRank. ACM Conference on Information and Knowledge Management, California, USA.
[4] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. European Conference on Computer Systems, Bordeaux, France.
[5] DIMACS. 2006. USA Road Network. http://www.dis.uniroma1.it/challenge9/download.shtml.
[6] Matthew Gardner, Andrew McNabb, and Kevin Seppi. 2012. A speculative approach to parallelization in particle swarm optimization. *Swarm Intelligence* 6, 2 (2012), 77–116.
[7] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. Operating Systems Design and Implementation(OSDI), Hollywood, CA.
[8] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. Operating Systems Design and Implementation(OSDI), Broomfield, CO.
[9] Imranul Hoque and Indranil Gupta. 2013. LFGraph: Simple and fast distributed graph analytics. ACM SIGOPS Conference on Timely Results in Operating Systems, PA, USA.
[10] Hakbeom Jang, Channoh Kim, and Jae W Lee. 2013. Practical speculative parallelization of variable-length decompression algorithms. *Languages, Compilers, and Tools for Embedded Systems* 48, 5 (2013), 55–64.
[11] Jeff Kahn, Nathan Linial, Noam Nisan, and Michael E Saks. 1989. On the cover time of random walks on graphs. *Journal of Theoretical Probability* 2, 1 (1989), 121–128.
[12] U Kang, Charalampos E Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. 2011. HADI: Mining Radii of Large Graphs. *ACM Transactions on Knowledge Discovery From Data* 5, 2 (2011), 8.
[13] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. ACM European Conference on Computer Systems, New York, USA.
[14] LAW. 2002. Large Graphs. http://law.di.unimi.it/datasets.php.
[15] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2010. GraphLab: a new framework for parallel machine learning. Uncertainty in Artificial Intelligence, California, USA.
[16] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. ACM SIGMOD International Conference on Management of data, Indiana, USA.
[17] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.
[18] Flajolet Philippe and Martin G. Nigel. 1985. Probabilistic Counting Algorithms for Data Base Applications. *J. Comput. System Sci.* 31, 2 (1985), 182–209.
[19] Semih Salihoglu and Jennifer Widom. 2013. Gps: A graph processing system. International Conference on Scientific and Statistical Database Management, Edinburgh, Scotland, UK.
[20] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. 2008. Estimating PageRank on graph streams. Symposium on Principles of Database Systems, Vancouver, Canada.
[21] Zechao Shang and Jeffrey Xu Yu. 2014. Auto-approximation of graph computing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1833–1844.
[22] Stanford. 2009. Large Network Dataset. https://snap.stanford.edu/.
[23] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. ACM SIGKDD international conference on Knowledge discovery and data mining, Beijing, China.
[24] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. ACM international conference on Web search and data mining, New York, USA.
[25] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.
[26] Luis M Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. 2014. Adaptive partitioning for large-scale dynamic graphs. International Conference on Distributed Computing Systems (ICDCS), Madrid, Spain.
[27] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. Sync or async: Time to fuse for distributed graph-parallel computation. ACM Sigplan Symposium on Principles and Practice of Parallel Programming, California, USA.
[28] Raphael Yuster. 2012. Approximate shortest paths in weighted graphs. *J. Comput. System Sci.* 78, 2 (2012), 632–637.
[29] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2014. Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel and Distributed Systems* 25, 8 (2014), 2091–2100.