GRAPE: Parallelizing Sequential Graph Computations

Wenfei Fan[1,2], Jingbo Xu[1,2], Yinghui Wu[3], Wenyuan Yu[2], Jiaxin Jiang[4]
[1]University of Edinburgh [2]Beihang University [3]Washington State University [4]Hong Kong Baptist University
{wenfei@inf, jingbo.xu@}ed.ac.uk, yinghui@eecs.wsu.edu, yuwenyuan@act.buaa.edu.cn, jxjian@comp.hkbu.edu.hk

ABSTRACT

We demonstrate GRAPE, a parallel GRAPh query Engine. GRAPE advocates a parallel model based on a simultaneous fixed point computation in terms of partial and incremental evaluation. It differs from prior systems in its ability to parallelize existing sequential graph algorithms as a whole, without the need for recasting the entire algorithms into a new model. One of its unique features is that under a monotonic condition, GRAPE parallelization guarantees to terminate with correct answers as long as the sequential algorithms "plugged in" are correct. We demonstrate its parallel computations, ease-of-use and performance compared with the start-of-the-art graph systems. We also demonstrate a use case of GRAPE in social media marketing.

1. INTRODUCTION

Graph queries have found prevalent use in transportation network analysis, knowledge extraction, Web mining, social networks and social marketing. Our familiar graph queries include (a) graph traversal, *e.g.,* shortest distance queries, (b) keyword search in graphs, (c) pattern matching via subgraph isomorphism or simulation. On real-life graphs that easily have billions of nodes and edges, graph queries are costly even for reachability (linear-time), not to mention subgraph isomorphism (NP-complete).

To support graph queries on large-scale graphs, several parallel systems have been developed [5,6,9,11]. These systems, however, require users to recast graph algorithms into their models. While graphs have been studied for decades and a number of sequential algorithms are already in place, to use Pregel [6], for instance, one has to "think like a vertex" and recast the entire existing algorithms into a vertex-centric model; similarly when programming with other systems, *e.g.,* [11], which adopts vertex-centric programming by treating blocks as vertices. The recasting is nontrivial for people who are not very familiar with their models. This makes these systems a privilege for experienced users only.

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

Table 1: Graph traversal on parallel systems

System	Category	Time(s)	Comm.(MB)
Giraph	vertex-centric	10126	1.02×10^5
GraphLab	vertex-centric	8586	1.02×10^5
Blogel	block-centric	226	2.8×10^3
GRAPE	auto-parallelization	10.5	0.05

Is it possible to have a system such that we can plug sequential graph algorithms into it, and it parallelizes the computations across multiple processors, without drastic degradation in performance or functionality of existing systems?

GRAPE. This motivates us to develop GRAPE [2], a parallel GRAPh query Engine. It has the following unique features that differ from previous parallel graph systems.

(1) Ease of programming. GRAPE provides a simple programming model. For a class $\mathcal{Q}$ of graph queries, users only need to provide three (existing) *sequential* (incremental) algorithms for $\mathcal{Q}$ with only minor changes. This makes parallel computations accessible to users who know conventional graph algorithms covered in undergraduate textbooks.

(2) Semi-automated parallelization. GRAPE *parallelizes* the sequential algorithms based on a combination of partial evaluation and incremental computation. It guarantees to terminate with correct answers under a monotonic condition, as long as the sequential algorithms plugged in are correct.

(3) Graph-level optimization. GRAPE inherits optimization strategies available for sequential algorithms and graphs, *e.g.,* indexing, compression and partitioning. These strategies are hard to implement for vertex-centric programs.

(4) Scale-up. In addition to the ease of programming, the performance of GRAPE is comparable to the state-of-the-art systems, such as vertex-centric systems Giraph [1](Pregel) and GraphLab [5], and block-centric Blogel [11], outperforming these systems in most of the cases [2]. As an example, Table 1 shows the performance of the systems for shortest-path queries (SSSP) over US road network with 24 processors.

As proof of concept, a primitive version of GRAPE is available [4]. Below we give an overview of GRAPE (Section 2), presenting its programming model (Section 2.1), parallel model (Section 2.2) and architecture (Section 2.3). The demo will walk through GRAPE and show how it supports "plug-and-play" of sequential algorithms for various classes of graph queries, compared with Giraph, GraphLab and Blogel (Section 3). We will also demonstrate how GRAPE is used in social media marketing with graph association rules [1].

[1]http://giraph.apache.org/

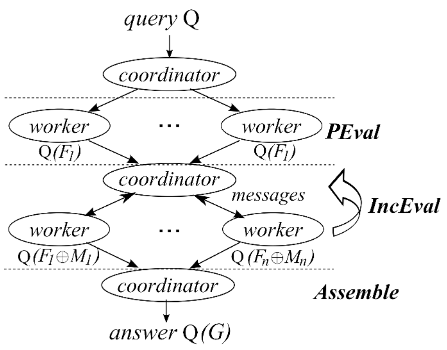


Figure 1: Workflow of GRAPE

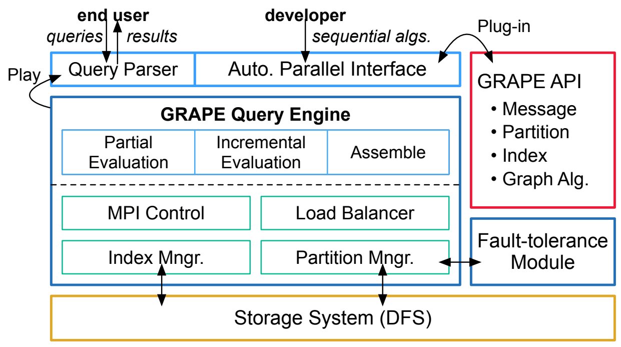


Figure 2: GRAPE Architecture

2. SYSTEM OVERVIEW

We start with the foundation and architecture of GRAPE.

2.1 Programming Model

For a class $\mathcal{Q}$ of graph queries, a GRAPE user only needs to specify *three* core functions with *sequential algorithms* for $\mathcal{Q}$.

PEval is any sequential algorithm for $\mathcal{Q}$ that given a query $Q \in \mathcal{Q}$ and a graph G, computes the answer $Q(G)$ to Q in G.

IncEval is any sequential incremental algorithm for $\mathcal{Q}$ that given Q, G, $Q(G)$ and updates M to G, computes changes ΔO to the old output $Q(G)$ such that $Q(G \oplus M) = Q(G) \oplus \Delta O$, where $G \oplus M$ denotes graph G updated by M.

Assemble collects partial answers that are computed locally at each worker by PEval and IncEval (see Section 2.2), and combines them into a complete answer; it is typically simple.

As will be seen Section 2.2, PEval and IncEval extend existing sequential algorithms only with two declarations.

2.2 Parallel Model

GRAPE adopts a parallel model based on a fixed point computation in terms of partial and incremental evaluation.

Workflow. GRAPE employs a *coordinator* P_0 and a set of n *workers* $P_1, \ldots, P_n$. It works on a graph G fragmented into $(F_1, \ldots, F_n)$ via a partition strategy $\mathcal{P}$ picked by the user. Each worker P_i maintains a fraction F_i ($i \in [1, n]$) of G.

Each fragment F_i has a set of *update parameters*, which are variables associated with "border nodes" of F_i, *e.g.,* nodes that have edges from or to another fragment F_j. The parameters are declared in PEval and inherited by IncEval, along with an aggregate function to resolve conflicts (when a variable is given multiple values by different workers). These are the only addition to existing sequential algorithms. Messages are automatically generated from update parameters for communication among workers (see below).

Given a query $Q \in \mathcal{Q}$, GRAPE posts the same Q to all the workers. As shown in Fig. 1, GRAPE computes $Q(G)$ in three phases following BSP (Bulk Synchronous Parallel [10]).

(1) Partial evaluation (**PEval**). In the first superstep, each processor P_i executes PEval against its local data F_i, to compute *partial answers* $Q(F_i)$ in parallel for all $i \in [1, n]$. Function PEval facilities data-partitioned parallelism via *partial evaluation*. After $Q(F_i)$ is computed, PEval generates a message at each P_i and sends it to coordinator P_0. The messages are simply update parameters whose values are *changed*.

(2) Incremental computation (**IncEval**). GRAPE iterates the following supersteps until it terminates. Each superstep has two steps, one at P_0 and the other at the workers. (a) Coordinator P_0 routes messages from the last superstep to workers, if there is any, and triggers the next superstep, like BSP. (b) Upon receiving message M_i, worker P_i *incrementally* computes $Q(F_i \oplus M_i)$ with IncEval, *by treating M_i as updates*, in parallel for $i \in [1, n]$. It sends a message to P_0 that encodes the changes to the update parameters of F_i.

(3) Termination (**Assemble**). At each superstep, coordinator P_0 checks whether for all $i \in [1, n]$, P_i is inactive, *i.e.,* P_i is done with its local computation, and there is no more change to any update parameter of F_i. If so, GRAPE pulls partial results from all workers, and computes $Q(G)$ by Assemble at P_0. It returns $Q(G)$ and terminates.

Fixed point. GRAPE supports a simultaneous fixed point operator $\phi(R_1, \ldots, R_n)$ on n fragments defined as:

$$\begin{aligned} R_i^0 &= \mathsf{PEval}(Q, F_i^0[\bar{x}_i]), \\ R_i^{r+1} &= \mathsf{IncEval}(Q, R_i^r, F_i^r[\bar{x}_i], M_i), \end{aligned}$$

where $i \in [1, n]$, r indicates a superstep, R_i^r denotes partial results in step r at worker P_i, $F_i^0 = F_i$, $F_i^r[\bar{x}_i]$ is fragment F_i at the end of superstep r carrying update parameters $\bar{x}_i$, and M_i indicates changes to $\bar{x}_i$. More specifically, (1) in the first superstep, PEval computes the partial answers R_i^0 ($i \in [1, n]$). (2) At superstep $r+1$, the partial answers R_i^{r+1} are incrementally updated by IncEval, taking Q, R_i^r and the messages M_i as input. (3) The computation proceeds until $R_i^{r_0+1} = R_i^{r_0}$ at a "fixed point" r_0. Assemble is then invoked to combine all partial answers $R_i^{r_0}$ and get $Q(G)$.

We demonstrate the following unique features of GRAPE, and invite interested reader to consult [2] for details.

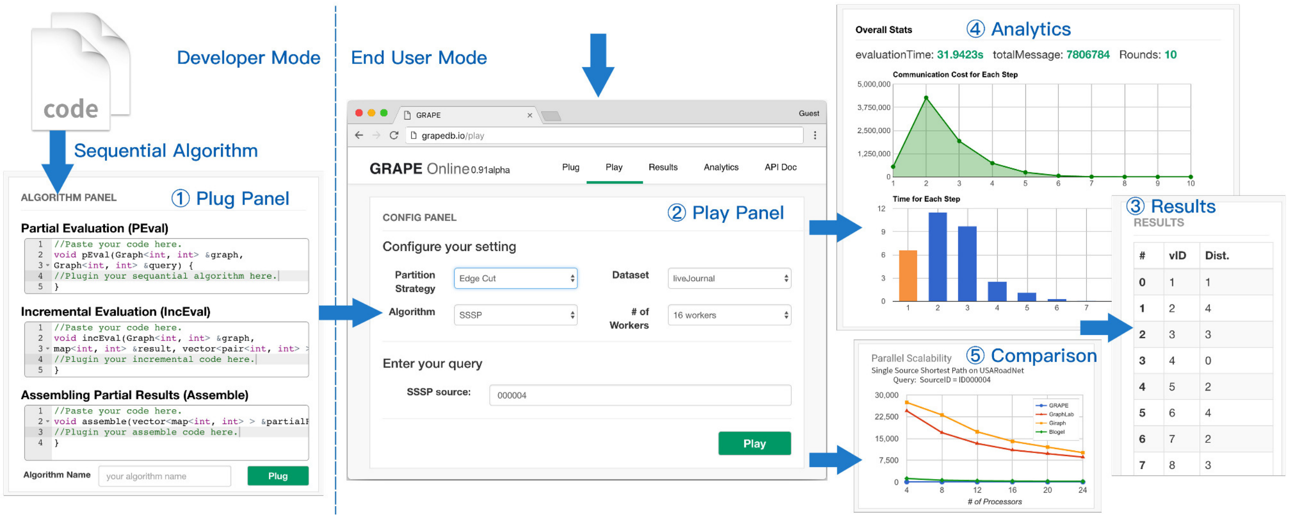
(1) GRAPE plugs in sequential algorithms as a whole, and executes these algorithms on entire graph fragments, inheriting all optimization strategies for the sequential algorithms.

(2) GRAPE reduces the costs of iterative graph computations by using IncEval, to minimize unnecessary recomputations. In particular, IncEval is *bounded* [7] if given F_i, Q, $Q(F_i)$ and messages M_i, it computes ΔO_i such that $Q(F_i \oplus M_i) = Q(F_i) \oplus \Delta O_i$, and moreover, its cost can be expressed as a function in $|M_i| + |\Delta O_i|$, the size of changes in the input and output, instead of $|F_i|$, no matter how big F_i is.

Example 1: For single-source shortest path (SSSP), one can readily use the following sequential algorithms.

(1) PEval is our familiar Dijkstra's algorithm [3] that computes the distance from s to each node v, denoted by an integer variable x_v (initially ∞ if $s \neq v$). The update parameters of fragment F_i consist of x_u's for all border nodes u.

PEval is executed at each worker in parallel. At the end, it collects the changed x_u values for its border nodes x_u, and sends the changes to P_0 as a message. Coordinator P_0 picks



ALGORITHM PANEL

Partial Evaluation (PEval)

```
//Paste your code here.
void pEval(Graph<int, int> &graph,
Graph<int, int> &query) {
//Plugin your sequantial algorithm here.
}
```

Incremental Evaluation (IncEval)

```
//Paste your code here.
void incEval(Graph<int, int> &graph,
map<int, int> &result, vector<pair<int, int> >
//Plugin your incremental code here.
}
```

Assembling Partial Results (Assemble)

```
//Paste your code here.
void assemble(vector<map<int, int> > &partialR
//Plugin your assemble code here.
}
```

Algorithm Name: your algorithm name — Plug

CONFIG PANEL

Configure your setting

Partition Strategy: Edge Cut — Dataset: liveJournal

Algorithm: SSSP — # of Workers: 16 workers

Enter your query

SSSP source: 000004

Play

Overall Stats

evaluationTime: 31.9423s totalMessage: 7806784 Rounds: 10

RESULTS

#	vID	Dist.
0	1	1
1	2	4
2	3	3
3	4	0
4	5	2
5	6	4
6	7	2
7	8	3

Figure 3: GRAPE **User Interface**

the *smallest value* for each x_u by using aggregate function min specified in PEval. The new values of x_u's are sent to corresponding workers P_i as message M_i.

(2) IncEval is a sequential incremental SSSP algorithm in [7]. It incrementally revises the variables affected by message M_i, and sends to P_0 the update parameters as in PEval.

(3) Assemble simply takes the union of partial results, *i.e.*, the value of x_v for each node v, the shortest distance.

We will demonstrate the following. (a) The update parameters decrease monotonically, and the GRAPE process terminates with correct answers. (b) The declarations of x_u and min are the only changes to the sequential algorithms of [3,7]. (c) The communication cost is confined to variables associated with border nodes whose values are changed, and are quite low. (d) IncEval is *bounded* [7]: its cost is decided by the size of message M_i and the nodes v in F_i with changed x_v values, not by $|F_i|$. The incremental steps effectively reduce the cost of iterative SSSP computation. □

Performance guarantees. As shown in [2], GRAPE has provable guarantees on correctness and generality. (1) *Assurance Theorem*. GRAPE guarantees to terminate with correct $Q(G)$ if PEval and IncEval are correct sequential algorithms for $\mathcal{Q}$, Assemble correctly combines partial results, and PEval and IncEval satisfy a monotonic condition, *i.e.*, their changes to update parameters follow a partial order defined on the domain of the variables, which is commonly observed in a wide range of graph computations. (2) *Simulation Theorem*. GRAPE optimally simulates parallel models MapReduce, BSP and PRAM. That is, all algorithms in MapRedue, BSP or PRAM with n workers (*e.g.*, those developed based on Pregel, Giraph, GraphLab and Blogel) can be simulated by GRAPE using n processors with the same number of supersteps and memory cost (see [2]).

2.3 System Architecture

GRAPE adopts a four-tier architecture depicted in Fig. 2.

(1) The top layer of GRAPE is an interface that allows developers to plug in sequential graph algorithms, and end users to parallelize the algorithms with GRAPE. We will elaborate how to plug and play sequential algorithms in Section 3.

(2) At the core of the system is a parallel query engine. It manages sequential algorithms registered in GRAPE API library, constructs parallel workflow for GRAPE programs, and executes the workflow for query answering.

(3) Underlying the query engine are (a) an *MPI Controller* (Message Passing Interface) for communications between coordinator and workers; it makes use of MPICH[2], a standard package that implements MPI for parallel and distributed systems such as GraphLab; (b) an *Index Manager* for loading indices, (c) a *Partition Manager* to partition graphs, and (d) a *Load Balancer* to balance workload across workers.

(4) The storage layer manages graph data in DFS (distributed file system). It is accessible to the query engine, Index Manager, Partition Manager and Load Balancer.

3. DEMONSTRATION OVERVIEW

The demonstration consists of two parts. (1) We walk through GRAPE to demonstrate its ease-of-use and performance compared with the state-of-the-art parallel graph query engines. (2) To further demonstrate its scalability, we demonstrate how GRAPE is used in social media marketing. The target audience of the demo includes anyone who is interested in large-scale graph querying and analytics.

Environment. Our prototype GRAPE [4] is implemented in C++. We demonstrate GRAPE and its applications in a cluster of 16 Aliyun ECS n2.large instances [3] (with one serving as the coordinator), each equipped with an Intel Xeon processor with 2.5GHz and 16G memory.

A walk through. We visualize and demonstrate how GRAPE supports "plug-and-play" of sequential algorithms, as shown in Fig. 3. Consider a class $\mathcal{Q}$ of graph queries.

(1) Plug (developers). As shown in Fig. 3(1), a developer specifies PEval, IncEval and Assemble for $\mathcal{Q}$ in the *plug panel.* These are sequential algorithms for $\mathcal{Q}$, denoted as a PIE program for $\mathcal{Q}$ and registered in the GRAPE library.

That is, one can plug existing algorithms into GRAPE for $\mathcal{Q}$. As shown in Example 1, the only changes to the algorithms are update parameters and aggregate functions.

(2) Play (end users). In the *play panel* (Fig. 3(2)), a user can pick a PIE program for $\mathcal{Q}$ from the GRAPE library, a graph G, a graph partition strategy $\mathcal{P}$, and a number n of

[2]https://www.mpich.org/
[3]https://intl.aliyun.com/

processors to work with. She can then enter queries $Q \in \mathcal{Q}$. GRAPE partitions G, parallelizes PEval and IncEval across n processors, and combines partial results to get $Q(G)$ with Assemble, following the parallel model of Section 2.2.

The Graph Partitioner of GRAPE provides several *built-in* vertex/edge cut partition strategies, including METIS, 1D/2D, and a streaming-style partition algorithm [8] that reduces cross edges. One is also allowed to plug new strategies into GRAPE as needed. As shown in Fig. 3(2), users only need to select a registered strategy from the library.

(3) Results. Query answers $Q(G)$ are displayed in the *result panel* (Fig. 3(3)). To demonstrate how GRAPE works, we have registered PIE programs for various graph queries in its library, including (1) single-source shortest paths (SSSP), (2) graph pattern matching via simulation (Sim) and subgraph isomorphism (SubIso), (3) keyword search in graphs (Keyword), (4) connected component detection (CC), and (5) collaborative filtering in machine learning (CF).

We invite the audience to compare sequential algorithms for (their own) query classes with their corresponding PIE programs. We also demonstrate the impact of graph partition strategies. For example, for SSSP, GRAPE takes 18.3 seconds and ships $7.5M$ messages with 16 nodes over a social network Livejournal partitioned with a best strategy METIS. It takes 30 seconds and ships $40M$ messages with stream-based partition in the same setting due to more cross edges.

(4) Analytics. The audience is invited to configure GRAPE, and observe its scalability by varying the number of workers, graph partition strategies, datasets and query classes. As shown in Fig. 3(4), we visualize the performance, and report the communication and computational costs for computing $Q(G)$. We also provide a fine-grained analysis, visualizing the performance of partial evaluation (PEval) and incremental steps (IncEval).

We also demonstrate how GRAPE supports graph-level optimization. GRAPE parallelizes sequential algorithms as a whole, and hence naturally supports optimization strategies developed for sequential algorithms, such as graph indexing, compression and load balancing in terms of graph partitions and workload estimates. For query class mentioned above, GRAPE supports optimized PIE programs. These are not easy to be supported by, *e.g.,* vertex-centric programming.

(5) Performance comparison. We visualize the computation and communication costs of different parallel graph query engines on real-life and synthetic graphs (Fig. 3(5)). We compare GRAPE with vertex-centric Giraph and GraphLab, and block-centric system Blogel [11]. We demonstrate that GRAPE achieves performance comparable the state-of-the-art systems at the very least, and illustrate the reasons.

Applications. Using real-life graphs (*e.g.,* Weibo), we demonstrate how GRAPE helps in social media marketing, an emerging application that is predicted to trump traditional marketing: "90% of customers trust peer recommendations versus 14% who trust advertising". We adopt *graph pattern association rules* (GPARs) [1] which extend conventional association rules by incorporating graph patterns, to identify regularities in social networks. A GPAR is of the form $Q(x, y) \Rightarrow p(x, y)$, where $Q(x, y)$ is a graph pattern, $p(x, y)$ is a predicate, and x and y are two designated nodes in Q. It says that if the topological condition specified by Q is satisfied, then x and y are likely to be associated with p.

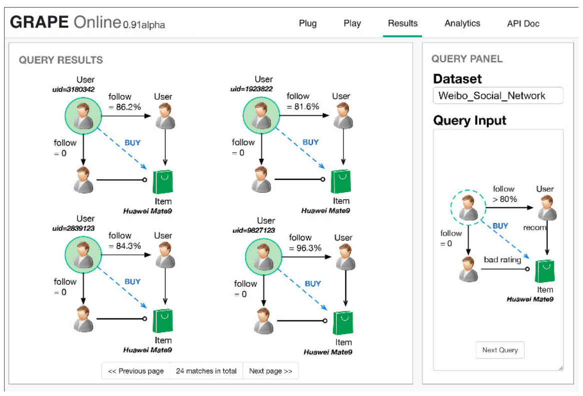


Figure 4: Social media marketing

Example 2: Figure 4 shows a GPAR φ: *if among the people followed by x_o, (a) at least 80% of them recommend Huawei Mate 9, and (b) no one gives it a bad rating, then the chances are that x_o will buy this mobile phone.* Hence we can recommend Huawei Mate 9 to x_o. Four potential customers suggested by φ are shown in the result panel of Fig. 4 $\square$

We demonstrate that given a set of GPARs, GRAPE efficiently finds potential customers ranked by confidence, by parallelizing PIE programs for subgraph isomorphism (SubIso). It offers a provable guarantee that the more workers are used, the faster it finds potential customers [1].

Acknowledgments. Fan is supported in part by ERC 652976, NSFC 61421003, 973 Program 2014CB340302, EPSRC EP/M025268/1, Shenzhen Peacock Program 1105100030834361, the Foundation for Innovative Research Groups of NSFC, Beijing Advanced Innovation Center for Big Data and Brain Computing. Wu is supported by NSF IIS-1633629, and Jiang by HKRGC GRF HKBU12232716.

4. REFERENCES

[1] W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns. *PVLDB*, 8(12):1502–1513, 2015.
[2] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, B. Zhang, Z. Zheng, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *SIGMOD*, 2017.
[3] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.
[4] GRAPE. *http://grapedb.io/*.
[5] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.
[6] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.
[7] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
[8] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230, 2012.
[9] Y. Tian, A. Balmin, S. A. Corsten, and J. M. Shirish Tatikonda. From "think like a vertex" to "think like a graph". *PVLDB*, 7(7):193–204, 2013.
[10] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
[11] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.