From Think Parallel to Think Sequential

Wenfei Fan[1,2], Yang Cao[1], Jingbo Xu[2], Wenyuan Yu[2], Yinghui Wu[3],
Chao Tian[1,2], Jiaxin Jiang[4], Bohan Zhang[5]
[1]Univ. of Edinburgh [2]Beihang Univ. [3]Washington State Univ. [4]Hong Kong Baptist Univ. [5]Peking Univ.
{wenfei@inf, yang.cao@, chao.tian@}ed.ac.uk, {xujb, yuwenyuan}@act.buaa.edu.cn, yinghui@eecs.wsu.edu,
jxjian@comp.hkbu.edu.hk, bohan@pku.edu.cn

ABSTRACT

This paper presents GRAPE, a parallel GRAPh Engine for graph computations. GRAPE differs from previous graph 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. Underlying GRAPE are a simple programming model, and a principled approach based on fixpoint computation with partial evaluation and incremental computation. Under a monotonic condition, GRAPE guarantees to converge at correct answers as long as the sequential algorithms are correct. We show how our familiar sequential graph algorithms can be parallelized by GRAPE. In addition to the ease of programming, we experimentally verify that GRAPE achieves comparable performance to the state-of-the-art graph systems, using real-life and synthetic graphs.

1. INTRODUCTION

There has been increasing demand for graph computations, *e.g.,* graph traversal, connectivity, pattern matching, and collaborative filtering. Indeed, graph computations have found prevalent use in mobile network analysis, pattern recognition, knowledge discovery, transportation networks, social media marketing and fraud detection, among other things. In addition, real-life graphs are typically big, easily having billions of nodes and trillions of edges [18]. With these comes the need for parallel graph computations. In response to the need, several parallel graph systems have been developed, *e.g.,* Pregel [25], GraphLab [16, 24], Trinity [29], GRACE [35], Blogel [37], Giraph++ [31], and GraphX [17].

However, users often find it hard to write and debug parallel graph programs using these systems. The most popular programming model for parallel graph algorithms is the vertex-centric model, pioneered by Pregel and GraphLab. For instance, to program with Pregel, one needs to "think like a vertex", by writing a user-defined function *compute(msgs)* to be executed at a vertex v, where v communicates with other vertices by message passing (*msgs*). Although graph computations have been studied for decades and a large number of sequential (single-machine) graph algorithms are already in place, to use Pregel, one has to recast the existing algorithms into vertex-centric programs. Trinity and GRACE also support vertex-centric programming. While Blogel and Giraph++ allow blocks to have their status as a "vertex" and support block-level communication, they still adopt the vertex-centric programming paradigm. GraphX also recasts graph computation into its distributed dataflow framework as a sequence of join and group-by stages punctuated by map operations, on Spark platform (see [36] for a survey). The recasting is nontrivial for users who are not very familiar with the parallel models. Moreover, none of the systems provides guarantee on the correctness or even termination of parallel programs developed in their models. These make the existing systems a privilege for experienced users only.

System	Category	Time(s)	Comm.(MB)
Giraph	vertex-centric	434.0	1.13×10^5
GraphLab	vertex-centric	41.7	1.07×10^5
Blogel	block-centric	112.3	1.23×10^5
GRAPE	think sequential	24.3	1.47×10^4

Table 1: Graph traversal on parallel systems

Is it possible to simplify parallel programming for graph computations, from "think parallel" to "think sequential"? That is, can we have a system that parallelizes existing sequential graph algorithms across a cluster of processors? Better yet, is there a general condition under which the parallelization guarantees to converge at correct answers as long as the sequential algorithms are correct? After all, the human's brain is not wired to think parallel.

To answer these questions, we develop GRAPE, a parallel GRAPh Engine. It differs from prior systems in the following.

(1) Ease of programming. GRAPE supports a simple programming model. For a class $\mathcal{Q}$ of graph queries, users only need to provide three sequential (incremental) algorithms for $\mathcal{Q}$, with *no need* to recast them into a new model, or revise the logic of the algorithms. This makes parallel computations accessible to users who know conventional graph algorithms covered in college textbooks.

(2) Parallelization. GRAPE *parallelizes* the computation across a cluster of processors, based on a fixpoint computation with partial evaluation and incremental computation. Under a monotonic condition, it guarantees to converge with correct answers as long as the three sequential algorithms provided are correct.

(3) Optimization. GRAPE inherits all optimization strategies available for sequential graph algorithms, *e.g.,* indexing, compression and partitioning. These are hard to implement for vertex programs.

(4) Scale-up. The ease of programming does not imply performance degradation compared with the state-of-the-art systems such as vertex-centric Giraph [3] (Pregel) and GraphLab, and block-centric Blogel. For instance, Table 1 shows the performance of the systems for shortest-path queries over Friendster [2] with 192 workers. GRAPE outperforms Giraph, GraphLab and Blogel in both response time and communication costs (see Section 4).

This paper presents the programming and parallel models of GRAPE (Section 2), shows how it parallelizes sequential algorithms (Section 3), and empirically evaluates GRAPE (Section 4).

2. GRAPE PARALLELIZATION

We present the programming paradigm and parallel model of GRAPE. Interested readers are invited to see [14] for details.

2.1 Graph Partition

We start with basic notations. We consider directed or undirected graphs $G = (V, E, L)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges; and (3) each node v in V (resp. edge $e \in E$) carries $L(v)$ (resp. $L(e)$), indicating its content, as found in social networks, knowledge bases and property graphs.

Partition strategy. Given a graph G and an integer m, a graph partition strategy $\mathcal{P}$ partitions G into *fragments* $\mathcal{F} = (F_1, \ldots, F_m)$. Each fragment $F_i = (V_i, E_i, L_i)$ is a subgraph of G that resides at processor P_i, for $i \in [1, m]$; and $E = \bigcup_{i\in[1,m]} E_i$, $V = \bigcup_{i\in[1,m]} V_i$. Under edge-cut partition [8, 9], denote by

- $F_i.I$ the set of nodes $v \in V_i$ such that there exists edge (v', v) *from* a node v' in F_j;
- $F_i.O$ the set of nodes v' in some F_j such that there exists an edge (v, v') from $v \in V_i$; and
- $\mathcal{F}.O = \bigcup_{i\in[1,m]} F_i.O$, and $\mathcal{F}.I = \bigcup_{i\in[1,m]} F_i.I$.

A cut edge from F_i to F_j has a copy in each of F_i and F_j ($i \neq j$). We refer to nodes in $F_i.I$ (or $F_i.O$) as *border nodes* of fragment F_i *w.r.t.* partition strategy $\mathcal{P}$. Note that $\mathcal{F}.I = \mathcal{F}.O$.

Under vertex-cut partition [22], $\mathcal{F}.O$ and $\mathcal{F}.I$ correspond to entry vertices and exit vertices, respectively.

2.2 Programming Paradigm

Consider a graph computation problem $\mathcal{Q}$. Using our familiar terms, we refer to an instance Q of $\mathcal{Q}$ as a *query* of $\mathcal{Q}$. To answer $Q \in \mathcal{Q}$ with GRAPE, a user only needs to specify three functions.

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

Assemble: a function Assemble that collects partial answers computed locally at each worker by PEval and IncEval, and assembles them into complete answer $Q(G)$. It is typically straightforward.

Functions PEval, IncEval and Assemble are referred to as *a* PIE *program for* $\mathcal{Q}$. Here PEval and IncEval are *existing sequential* (incremental) algorithms for $\mathcal{Q}$, with the following additions to PEval.

Update parameters. PEval declares *status variables* $\bar{x}$ for a set C_i of nodes and edges in a fragment F_i, which store contents of F_i or intermediate results of a computation. Here C_i is a set of nodes and edges within d-hops of the border nodes in F_i, *e.g.,* $F_i.O$, for an integer d. When $d = 0$, one may define C_i as, *e.g.,* $F_i.O$.

We denote by $C_i.\bar{x}$ the set of *update parameters* of F_i, which consists of status variables of the nodes and edges in C_i, *i.e.,* variables in $C_i.\bar{x}$ are the candidates to be updated.

Aggregate function. PEval also specifies a function f_{aggr}, *e.g.,* min, max, to resolve conflicts when multiple workers attempt to assign different values to the same update parameter.

The update parameters and aggregate function are specified in PEval and are shared by IncEval. As will be seen shortly, IncEval only needs to deal with changes ΔG to update parameters.

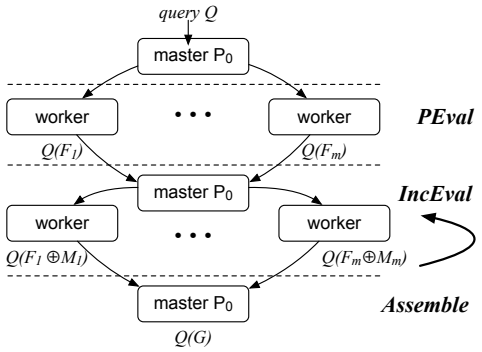


Figure 1: Workflow of GRAPE

2.3 Parallel Model

Given a partition strategy $\mathcal{P}$ and a PIE program ρ (PEval, IncEval, Assemble) for $\mathcal{Q}$, GRAPE parallelizes ρ as follows. It first partitions G into $(F_1, \ldots, F_m)$ with $\mathcal{P}$, and distributes fragments F_i's across m shared-nothing *virtual workers* $(P_1, \ldots, P_m)$. It maps m virtual workers to n physical workers. When $n < m$, multiple virtual workers mapped to the same worker share memory. Graph G is partitioned *once* for *all queries* $Q \in \mathcal{Q}$ on G.

We start with basic ideas behind GRAPE parallelization.

Partial evaluation. Given a function $f(s, d)$ and the s part of its input, *partial evaluation* is to specialize $f(s, d)$ *w.r.t.* the known input s [21]. That is, it performs the part of f's computation that depends only on s, and generates a partial answer, *i.e.,* a residual function f' that depends on the as yet unavailable input d. For each worker P_i in GRAPE, its local fragment F_i is its known input s, while the data residing at other workers accounts for the yet unavailable input d. As will be seen shortly, given a query $Q \in \mathcal{Q}$, GRAPE computes $Q(F_i)$ in parallel as partial evaluation.

Incremental evaluation. Workers exchange *changed values* of their local update parameters with each other. Upon receiving message M_i that consists of changes to the update parameters at fragment F_i, worker P_i treats M_i as *updates* to F_i, and *incrementally* computes changes ΔO_i to $Q(F_i)$ such that $Q(F_i \oplus M_i) = Q(F_i) \oplus \Delta O_i$. This is often more efficient than recomputing $Q(F_i \oplus M_i)$ starting from scratch, since in practice M_i is often small. Better still, the computation may be *bounded*: its cost can be expressed as a function in $|M_i| + |\Delta O_i|$, *i.e.,* the size of changes in the input and output, instead of $|F_i|$, no matter how big F_i is [12, 28].

Parallelization. We use (BSP) (Bulk Synchronous Parallel model [32]). Given a query $Q \in \mathcal{Q}$ at master P_0, GRAPE answers Q in the partitioned graph G. It posts the same Q to all the workers, and computes $Q(G)$ in three phases as follows, as shown in Fig. 1.

(1) Partial evaluation (**PEval**). In the first superstep, upon receiving query Q, each worker P_i applies function PEval to its local fragment F_i, to compute partial results $Q(F_i)$, in parallel ($i \in [1, m]$). After $Q(F_i)$ is computed, PEval generates a message at each worker P_i and sends it to master P_0. The message is simply the set $C_i.\bar{x}$ of update parameters at fragment F_i.

For each $i \in [1, m]$, master P_0 maintains update parameters $C_i.\bar{x}$. It deduces a message M_i to worker P_i based on the following *message grouping policy*. (a) For each status variable $x \in C_i.\bar{x}$, it collects the group S_x of values for x from all messages, and computes $x_{\mathsf{aggr}} = f_{\mathsf{aggr}}(S_x)$ by applying the aggregate function f_{aggr}. (b) Message M_i includes only those x_{aggr}'s such that $x_{\mathsf{aggr}} \neq x$, *i.e.,* only those *changed* values of the update parameters at F_i.

(2) Incremental computation (**IncEval**). GRAPE iterates the following supersteps until it terminates. Following BSP, each super-

Input: $F_i(V_i, E_i, L_i)$, source vertex s
Output: $Q(F_i)$ consisting of current $\mathsf{dist}(s, v)$ for all $v \in V_i$

Declaration: /*candidate set C_i is $F_i.O$*/
for each node $v \in V_i$, an integer variable $\mathsf{dist}(s, v)$;
message $M_i := \{\mathsf{dist}(s, v) \mid v \in F_i.O\}$;
aggregate function $f_{\mathsf{aggr}} = \mathsf{min}(\mathsf{dist}(s, v))$;

/*sequential algorithm for SSSP (pseudo-code)*/
1. initialize priority queue Que;
2. $\mathsf{dist}(s, s) := 0$;
3. **for each** v in V_i **do**
4. **if** $v! = s$ **then**
5. $\mathsf{dist}(s, v) := \infty$;
6. Que.addOrAdjust(s, $\mathsf{dist}(s, s)$);
7. **while** Que is not empty **do**
8. u := Que.pop() // pop vertex with minimal distance
9. **for each** child v of u **do** // only v that is still in Que
10. $alt := \mathsf{dist}(s, u) + L_i(u, v)$;
11. **if** $alt < \mathsf{dist}(s, v)$ **then**
12. $\mathsf{dist}(s, v) := alt$;
13. Que.addOrAdjust(v, $\mathsf{dist}(s, v)$);
14. $Q(F_i) := \{\mathsf{dist}(s, v) \mid v \in V_i\}$

Figure 2: Parallel SSSP: Partial evaluation PEval

step starts after the master P_0 receives messages (possibly empty) from *all workers* P_i for $i \in [1, m]$. A superstep has two steps itself, one at P_0 and the other at the workers.

(a) Master P_0 routes (nonempty) messages from the last superstep to workers, if there exists any.

(b) Upon receiving message M_i, worker P_i *incrementally* computes $Q(F_i \oplus M_i)$ by applying IncEval, and by *treating* M_i *as updates* to $C_i.\bar{x}$, in parallel for $i \in [1, m]$.

At the end of the process of IncEval, worker P_i sends a message to P_0 that encodes *updated values* of $C_i.\bar{x}$, if any. Upon receiving messages from all workers, master P_0 deduces message M_i to each worker P_i following the message grouping policy given above; it sends message M_i to worker P_i in the next superstep.

(3) Termination (**Assemble**). At each superstep, master P_0 checks whether for all $i \in [1, m]$, P_i is inactive, *i.e.,* P_i is done with its local computation, and there exists no more change to the update parameters of F_i. If so, GRAPE pulls partial results from all workers, and applies Assemble to group them together and get the final result at P_0, denoted by $\rho(Q, G)$. It returns $\rho(Q, G)$ and terminates.

Example 1: We show how GRAPE parallelizes the computation of Single Source Shortest Path (SSSP), a common graph computation problem. Consider a directed graph $G = (V, E, L)$ in which for each edge e, $L(e)$ is a positive number. The length of a path $(v_0, \ldots, v_k)$ in G is the sum of $L(v_{i-1}, v_i)$ for $i \in [1, k]$. For a pair (s, v) of nodes, denote by $\mathsf{dist}(s, v)$ the *distance* from s to v, *i.e.,* the length of a shortest path from s to v. Given graph G and a node s in V, SSSP computes $\mathsf{dist}(s, v)$ for all $v \in V$.

Under edge-cut partition [9], GRAPE takes the set $F_i.O$ of "border nodes" as C_i at each P_i (with edges across distinct fragments). The PIE program for SSSP consists of (1) our familiar Dijkstra's algorithm for SSSP [15] as PEval, (2) a sequential incremental algorithm of [27] as IncEval, and (3) a straightforward Assemble.

(1) PEval. As shown in Fig. 2, PEval (lines 1-14) is verbally identical to Dijsktra's algorithm [15]. One only needs to declare (a) status variable as an integer variable $\mathsf{dist}(s, v)$ for each node v, initially ∞ (except $\mathsf{dist}(s, s) = 0$); (b) update parameters as $C_i.\bar{x} = \{\mathsf{dist}(s, v) \mid v \in F_i.O\}$, *i.e.,* the status variables associated with nodes in $F_i.O$ at fragment F_i; and (c) min as an aggregate function f_{aggr}. If there are multiple values for the same $\mathsf{dist}(s, v)$, the smallest value is taken by the order on positive numbers.

At the end of its process, PEval sends $C_i.\bar{x}$ to master P_0. At P_0, GRAPE maintains $\mathsf{dist}(s, v)$ for all $v \in \mathcal{F}.O = \mathcal{F}.I$. Upon receiving messages from all workers, it takes the smallest value for each $\mathsf{dist}(s, v)$. It finds those variables with smaller $\mathsf{dist}(s, v)$ for $v \in F_j.O$, groups them into message M_j, and sends M_j to P_j.

Input: $F_i(V_i, E_i, L_i)$, partial result $Q(F_i)$, message M_i
Output: $Q(F_i \oplus M_i)$

Declaration: message $M_i = \{\mathsf{dist}(s, v) \mid v \in F_i.O,\ \mathsf{dist}(s, v)$ decreased$\}$;

1. initialize priority queue Que;
2. **for each** $\mathsf{dist}(s, v)$ in M_i **do**
3. Que.addOrAdjust(v, $\mathsf{dist}(s, v)$);
4. **while** Que is not empty **do**
5. u := Que.pop() /* pop vertex with minimum distance*/
6. **for each** children v of u **do**
7. $alt := \mathsf{dist}(s, u) + L_i(u, v)$;
8. **if** $alt < \mathsf{dist}(s, v)$ **then**
9. $\mathsf{dist}(s, v) := alt$;
10. Que.addOrAdjust(v, $\mathsf{dist}(s, v)$);
11. $Q(F_i) := \{\mathsf{dist}(s, v) \mid v \in V_i\}$

Figure 3: Parallel SSSP: Incremental evaluation IncEval

(2) IncEval. We give IncEval in Fig. 3. It is the sequential incremental algorithm for SSSP in [28] that is mildly revised to deal with changed $\mathsf{dist}(s, v)$ for v in $F_i.I$ (deduced by leveraging $\mathcal{F}.I = \mathcal{F}.O$). Using a queue Que, it starts with changes in M_i, propagates the changes to affected area, and updates the distances (see [28]). The partial result now consists of the revised distances (line 11). At the end of the process, it sends to master P_0 the updated values of those status variables in $C_i.\bar{x}$, as in PEval. It applies the aggregate function min to resolve conflicts.

Following [28], one can show that IncEval is *bounded*: its cost is determined by the sizes of "updates" $|M_i|$ and the changes to the output. This reduces the cost of iterative computation of SSSP.

(3) Assemble simply takes $Q(G) = \bigcup_{i \in [1,n]} Q(F_i)$, the union of the shortest distance for each node in each F_i.

The process converges at correct $Q(G)$. Updates to $C_i.\bar{x}$ are "monotonic": the value of $\mathsf{dist}(s, v)$ for each node v is computed from the active domain of G and does not increase. Moreover, $\mathsf{dist}(s, v)$ is the shortest distance from s to v as warranted by the sequential algorithms [15, 28] (PEval and IncEval). □

Fixpoint. The GRAPE parallelization of the PIE program can be modeled as a simultaneous fixpoint operator $\phi(R_1, \ldots, R_m)$ defined on m fragments. It starts with PEval for partial evaluation, and conducts incremental computation by taking with IncEval as the intermediate consequence operator, as follows:

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

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

Convergence. The correctness of the fixpoint computation is characterized as follows. Given a graph computation problem $\mathcal{Q}$, (a) the sequential algorithm PEval for $\mathcal{Q}$ is correct if for all queries $Q \in \mathcal{Q}$ and graphs G, it terminates and returns $Q(G)$; (b) the sequential incremental algorithm IncEval for $\mathcal{Q}$ is *correct* if it correctly updates old output $Q(G)$ to $Q(G \oplus M)$, by computing the changes ΔO to be applied to $Q(G)$, for changes (messages) M to update parameters; (c) Assemble is correct for $\mathcal{Q}$ *w.r.t.* partition strategy $\mathcal{P}$ if it correctly computes $Q(G)$ by assembling the partial answers from all workers, when GRAPE with PEval, IncEval and $\mathcal{P}$ terminates.

We say that GRAPE *correctly parallelizes* a PIE program ρ with partition strategy $\mathcal{P}$ if for all $Q \in \mathcal{Q}$ and graphs G, GRAPE guarantees to reach a fixpoint such that $\rho(Q, G) = Q(G)$.

It is shown [14] that under BSP, GRAPE correctly parallelizes a PIE program ρ for a graph computation problem $\mathcal{Q}$ with any partition strategy $\mathcal{P}$ if (a) PEval and IncEval of ρ are correct sequential algorithms for $\mathcal{Q}$, and (b) Assemble correctly combines partial results, and (c) PEval and IncEval satisfy a monotonic condition. The condition is as follows: for all status variables $x \in C_i.\bar{x}$, $i \in [1, m]$, (a) the values of x are from a finite set computed from the active domain of G and (b) there exists a partial order p_x on the values of x such that IncEval updates x in the order of p_x. That is, x draws values from a finite domain (condition (a) above), and x is updated "monotonically" following p_x (condition (b)).

Simulating other models. The simple parallel model of GRAPE does not come with a price of degradation in the functionality. Following [33], we say that parallel model $\mathcal{M}_1$ can *optimally simulate* model $\mathcal{M}_2$ if there is a compilation algorithm that transforms any program with cost C on $\mathcal{M}_2$ to a program with cost $O(C)$ on $\mathcal{M}_1$.

As shown in [14], GRAPE optimally simulates parallel models MapReduce [10], BSP [32] and PRAM (Parallel Random Access Machine) [33]. That is, all algorithms in these models with n workers can be simulated by GRAPE using n processors with the same number of supersteps and the same complexity. (2) We have shown that the simulation result above holds in the message-passing model described above, referred to as the designated message model in [14]. Hence, algorithms developed for graph systems based on MapReduce or BSP, *e.g.,* Pregel, GraphLab and Blogel, can be migrated to GRAPE without extra complexity.

Features. GRAPE has the following unique features.

(1) As shown in Fig. 4, to program with GRAPE, one only needs to provide a PIE program in the "plug" panel of GRAPE, which consists of (existing) sequential algorithms with minor changes. Given a partition strategy $\mathcal{P}$, a graph G, a query Q and the number m of processors in the "play" panel, GRAPE parallelizes the algorithms.

GRAPE aims to help users develop parallel programs, especially those who are more familiar with conventional sequential programming. This said, programming with GRAPE still requires to declare update parameters and design an aggregate function.

(2) Under a monotone condition, GRAPE parallelization guarantees to converge at the correct answer as long as the sequential algorithms are correct. This works regardless of partitioning strategy used, not limited to edge-cut and vertex-cut. Nonetheless, different strategies may yield partitions with various degrees of skewness and stragglers, which have an impact on the performance.

(3) GRAPE optimally simulates MapReduce, BSP and PRAM.

(4) GRAPE inherits existing optimization techniques developed for sequential graph algorithms, since it executes sequential algorithms on graph fragments, which are graphs themselves.

(5) GRAPE reduces the costs of iterative graph computations by using IncEval, to minimize unnecessary recomputations. While the speedup is more evident when IncEval is bounded [28], localizable or relatively bounded [11], these properties are not necessary.

There have been methods for incrementalizing graph algorithms, to get incremental algorithms from their batch counterparts [7].

3. PROGRAMMING WITH GRAPE

We next outline PIE programs for graph pattern matching (Sim), connectivity (CC) and collaborative filtering (CF), under edge-cut. PIE programs under vertex-cut can be developed similarly.

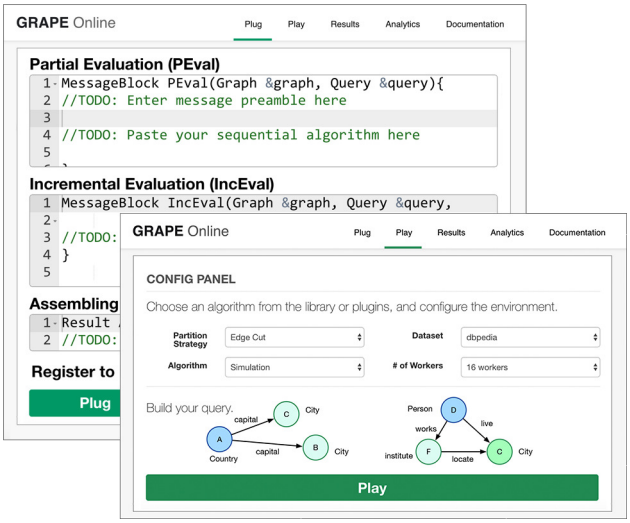


Figure 4: Programming Interface of GRAPE

Graph simulation (Sim). A *graph pattern* is a graph $Q = (V_Q, E_Q, L_Q)$, in which (a) V_Q is a set of *query nodes*, (b) E_Q is a set of *query edges*, and (c) each node u in V_Q carries a label $L_Q(u)$.

A graph G *matches* a pattern Q via *simulation* if there is a binary relation $R \subseteq V_Q \times V$ such that (a) for each query node $u \in V_Q$, there exists a node $v \in V$ such that $(u, v) \in R$, and (b) for each pair $(u, v) \in R$, $L_Q(u) = L(v)$, and for each query edge (u, u') in E_Q, there exists an edge (v, v') in graph G such that $(u', v') \in R$.

It is known that if G matches Q, then there exists a *unique maximum* relation [20], referred to as $Q(G)$. If G does not match Q, $Q(G)$ is the empty set. Given a directed graph G and a pattern Q, graph simulation is to compute the maximum relation $Q(G)$.

We show how GRAPE parallelizes graph simulation.

(1) PEval. GRAPE takes the sequential simulation algorithm of [20] as PEval to compute $Q(F_i)$ in parallel. PEval declares a Boolean status variable $x_{(u,v)}$ for each node u in V_Q and each node v in fragment F_i, indicating whether v matches u, initialized true. It takes $F_i.I$ as candidate set C_i. For each node $u \in V_Q$, PEval computes a set $\mathsf{sim}(u)$ of candidate matches v in F_i, and iteratively removes from $\mathsf{sim}(u)$ those nodes that violate the simulation condition (see [20] for details). At the end of the process, PEval sends $C_i.\bar{x} = \{x_{(u,v)} \mid u \in V_Q, v \in F_i.I\}$ to master P_0.

At master P_0, GRAPE maintains $x_{(u,v)}$ for all $v \in \mathcal{F}.I$. Upon receiving messages from all workers, it changes $x_{(u,v)}$ to false if it is false in *one of* the messages. This is specified by min as f_{aggr}, taking the order false $\prec$ true. GRAPE finds those variables that become false, groups them into messages M_j, and sends M_j to P_j.

(2) IncEval is the sequential incremental graph simulation algorithm of [12] in response to edge deletions. If $x_{(u,v)}$ is changed to false by message M_i, it is treated as deletion of "cross edges" to $v \in F_i.O$. It starts with changed status variables in M_i, propagates the changes to affected area, and removes from sim matches that become invalid (see [12] for details). The partial result is now the revised sim relation. At the end of the process, IncEval sends to P_0 updated values of status variables in $C_i.\bar{x}$, as in PEval.

IncEval is *semi-bounded* [12]: its cost is decided by the sizes of "updates" $|M_i|$ and changes to the affected area necessarily checked by all incremental algorithms for Sim, not by $|F_i|$.

(3) Assemble simply takes $Q(G) = \bigcup_{i \in [1,n]} Q(F_i)$, the union of all partial matches, *i.e.,* relation sim at each fragment F_i.

(4) Correctness is warranted by the convergence condition of GRAPE, as the sequential algorithms [12, 20] (PEval and IncEval)

are correct, and updates to $C_i.\bar{x}$ are monotonic: $x_{(u,v)}$ is initially true for each border node v, and is changed at most once to false.

Graph connectivity (CC). Given an undirected graph G, CC computes all connected components of G, referred to as *CC*s.

(1) PEval declares an integer variable $v.\mathsf{cid}$ for each node v in fragment F_i, initialized as its node id. It uses a standard sequential traversal (*e.g.,* DFS) to compute the local *CC*s of F_i and determines $v.\mathsf{cid}$ for each $v \in F_i$. For each local *CC* C, (a) PEval creates a "root" node v_c carrying the minimum node id in C as $v_c.\mathsf{cid}$, and (b) links all the nodes in C to v_c, and sets their cid as $v_c.\mathsf{cid}$. These can be completed in one pass of the edges of F_i via DFS. At the end of process, PEval sends $\{v.\mathsf{cid} \mid v \in F_i.I\}$ to master P_0.

At master P_0, GRAPE maintains $v.\mathsf{cid}$ for each all $v \in \mathcal{F}.I$. It updates $v.\mathsf{cid}$ by taking the smallest cid if multiple cids are received, by taking min as f_{aggr} in PEval. It groups the border nodes with updated cids into messages M_j, and sends M_j to P_j.

(2) IncEval incrementally updates the cids of the nodes in F_i upon receiving M_i. The message M_i sent to P_i consists of $v.\mathsf{cid}$ with updated (smaller) values of its border nodes v. For each v in M_i, IncEval (a) finds the root v_c of v, and (b) for v_c and all the border nodes linked to it, directly changes their cids to $v.\mathsf{cid}$.

Note that IncEval is *bounded*: it takes $O(|M_i|)$ time to identify the root nodes, and $O(|\mathsf{AFF}|)$ time to update cids by following the direct links from the root nodes, where AFF consists of only those nodes with their cid *changed*, independent of $|F_i|$.

(3) Assemble first updates the cid of each node to the cid of its linked root node. It then merges all the nodes having the same cids in a single bucket, and returns all buckets as *CC*s.

(4) Correctness. It is easy to see that the process terminates since the cids of the nodes are monotonically decreasing by aggregate function f_{aggr} until no changes can be made. Moreover, it correctly merges two local *CC*s by propagating smaller component ids.

Collaborative filtering (CF). CF takes as input a bipartite graph G that includes users U and products P, and a set of weighted edges $E \subseteq U \times P$ [23]. (1) Each user $u \in U$ (resp. product $p \in P$) carries latent factor vector $u.f$ (resp. $p.f$). (2) Each edge $e = (u, p)$ in E carries a weight $r(e)$, estimated as $u.f^T * p.f$ ($\emptyset$ for "unknown") that encodes a rating from user u to product p. The *training set* E_T refers to edge set $\{e \mid r(e) \neq \emptyset, e \in E\}$, *i.e.,* all the known ratings. Given these, CF computes the missing factor vectors $u.f$ and $p.f$ to minimize an error function $\epsilon(f, E_T) = \mathsf{min} \sum_{((u,p)\in E_T)} (r(u,p) - u.f^T p.f)^2 + \lambda(\|u.f\|^2 + \|p.f\|^2)$. This is typically carried out by the stochastic gradient descent (SGD) algorithm [23], which iteratively (1) predicts error $\epsilon(u,p) = r(u,p) - u.f^T * p.f$, for each $e = (u,p) \in E_T$, and (2) updates $u.f$ and $p.f$ accordingly to minimize $\epsilon(f, E_T)$.

GRAPE parallelizes CF by adopting SGD [23] as PEval, and the incremental algorithm ISGD of [34] as IncEval, using master P_0 to synchronize the shared factor vectors $u.f$ and $p.f$.

(1) PEval. It sets $C_i = F_i.I$ and declares status variable $v.x = (v.f, t)$ for $v \in C_i$, where $v.f$ is the factor vector of v (initially $\emptyset$), and t bookkeeps a timestamp at which $v.f$ is lastly updated. PEval is essentially the sequential SGD of [23]. It processes a "mini-batch" of training examples independently of others, to compute prediction error $\epsilon(u,p)$, and updates factor vectors f by a magnitude proportional to γ in the opposite direction of the gradient as:

$$u.f^t = u.f^{t-1} + \gamma(\epsilon(u,p) * v.f^{t-1} - \lambda * u.f^{t-1}); \quad (1)$$

$$p.f^t = p.f^{t-1} + \gamma(\epsilon(u,p) * u.f^{t-1} - \lambda * p.f^{t-1}). \quad (2)$$

At the end of its process, PEval sends messages M_i that consists of updated $v.x$ for each $v \in C_i = F_i.O$ to master P_0.

At P_0, GRAPE maintains $v.x = (v.f, t)$ for all border nodes $v \in \mathcal{F}.I = \mathcal{F}.O$. Upon receiving updated values $(v.f', t')$ with $t' > t$, it changes $v.f$ to $v.f'$, *i.e.,* it takes max as aggregate function f_{aggr} on timestamps. GRAPE then groups the updated vectors into messages M_j, and sends M_j to P_j as usual.

(2) IncEval is the incremental algorithm ISGD of [34]. Upon receiving message M_i at worker P_i, it computes $F_i \oplus M_i$ by treating M_j as updates to factor vectors of nodes in $F_i.I$, and only modifies affected factor vectors as in PEval based solely on new observations. It sends the updated vectors in C_i as in PEval.

(3) Assemble simply takes the union of all the factor vectors of nodes from the workers (to be used for recommendation).

(4) Correctness. The convergence condition in a sequential SGD algorithm [23, 34] is specified either as a predetermined maximum number of supersteps (*e.g.,*GraphLab), or when $\epsilon(f, E_T)$ is smaller than a threshold. In either case, GRAPE correctly infers CF models guaranteed by the correctness of SGD and ISGD, and by monotonic updates with the latest changes as in sequential SGD algorithms.

4. PERFORMANCE STUDY

We have implemented GRAPE [13]. We next empirically evaluate its efficiency and communication cost, using real-life and synthetic graphs. We compared the performance of GRAPE with three systems: Giraph (an open-source version of Pregel), GraphLab, and Blogel (the fastest block-centric system we are aware of).

Experimental setting. We used five real-life graphs of different types, including (1) traffic [5], an (undirected) US road network with 23 million nodes (locations) and 58 million edges; (2) UKWeb [6], a large Web graph with 133 million nodes and 5 billion edges; (3) Friendster [2], a social network with 65 million users and 1.8 billion relations; (4) DBpedia [1], a knowledge base with 5 million entities and 54 million edges, and in total 411 distinct labels; and (5) movieLens [4], a dense recommendation network (bipartite graph) with 20 million movie ratings (as weighted edges) between a set of 138000 users and 27000 movies. To test Sim with unlabeled Friendster, we generated 100 random node labels. We also randomly assigned weights to all graphs for testing SSSP.

Queries. We randomly generated the following queries. (a) We sampled 10 source nodes in each graph, and constructed an SSSP query for each node. (b) We generated 20 pattern queries for Sim, controlled by $|Q| = (|V_Q|, |E_Q|)$, the number of nodes and edges, respectively, using labels drawn from the graphs.

We remark that GRAPE can process query load without reloading the graph, but GraphLab, Giraph and Blogel need to reload the graph each time a query is issued, which is costly over large graphs.

Algorithms. We implemented the PIE programs for those query classes given in Sections 2 and 3. We used XtraPuLP [30] as the default graph partition strategy. We adopted basic sequential algorithms for all the systems without further optimization.

We also implemented algorithms for the queries for Giraph, GraphLab and Blogel. We used "default" code provided by the systems when available, and made our best efforts to develop "optimal" algorithms otherwise (see [14] for more details). We implemented synchronized algorithms for both GraphLab and Giraph for the ease of comparison. We expect the observed relative performance trends to hold on other similar graph systems.

We deployed the systems on a cluster of up to 12 machines, each with 16 threads of Intel Xeon 2.2GHz, and 128G memory. On each thread, a worker is deployed (thus in total 192 workers). Each experiment was run 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluated the efficiency of GRAPE by varying the number n of workers used, from 64 to 192. For SSSP and CC, we experimented with UKWeb, traffic and Friendster. For Sim, we used over Friendster and DBpedia. We used movieLens for CF as its application in movie recommendation.

(1) SSSP. Figures 5a-5c report the performance of the four systems for SSSP over traffic, UKWeb and Friendster, respectively. From the results we can see the following.

(a) GRAPE outperforms Giraph, GraphLab and Blogel by 14842, 3992 and 756 times, respectively, over traffic with 192 workers (Fig 5a). In the same setting, it is 556, 102 and 36 times faster over UKWeb (Fig. 5b), and 18, 1.7 and 4.6 times faster over Friendster (Fig. 5c). These tell us that by simply parallelizing sequential algorithms without further optimization, GRAPE already outperforms the state-of-the-art systems in response time.

The improvement of GRAPE over all the systems on traffic is much larger than on Friendster and UKWeb. (i) For Giraph and GraphLab, this is because synchronous vertex-centric algorithms take more supersteps to converge on graphs with larger diameters, *e.g.,* traffic. With 192 workers, Giraph take 10749 supersteps over traffic and 161 over UKWeb; similarly for GraphLab. In contrast, GRAPE is not vertex-centric and it takes 31 supersteps on traffic and 24 on UKWeb. (ii) Blogel also takes more (1690) supersteps over traffic than over UKWeb (42) and Friendster (23). It generates more blocks over traffic (with larger diameter) than UKWeb and Friendster. Since Blogel treats blocks as vertices, the benefit of parallelism is degraded with more blocks. (iii) GRAPE reduces redundant computation by the use of incremental IncEval.

(b) In all cases, GRAPE takes less time when n increases. On average, it is 1.4, 2.3 and 1.5 times faster for n from 64 to 192 over traffic, UKWeb and Friendster, respectively. (i) Compared with the results in [14] using less workers, this improvement degrades a bit. This is mainly because the larger number of fragments leads to more communication overhead. On the other hand, such impact is significantly mitigated by IncEval that only ships changed update parameters. (ii) In contrast, Blogel does not demonstrate such consistency in scalability. It takes more time on traffic when n is larger. When n varies from 160 to 192, it takes longer over Friendster. Its communication cost dominates the parallel cost as n grows, "canceling out" the benefit of parallelism. (iii) GRAPE has scalability comparable to GraphLab over Friendster and scales better over UKWeb and traffic. Giraph has better improvement with larger n, but with constantly higher cost (see (a)) than GRAPE.

(c) GRAPE significantly reduces supersteps. It takes on average 22 supersteps, while Giraph, GraphLab and Blogel take 3647, 3647 and 585 supersteps, respectively. This is because GRAPE runs sequential algorithms over fragmented graphs with cross-fragment communication only when necessary, and IncEval ships only *changes* to status variables. In contrast, Giraph, GraphLab and Blogel pass vertex-vertex (vertex-block) messages.

(2) CC. Figures 5d-5f report the performance for CC, and tell us the following. (a) Both GRAPE and Blogel substantially outperform Giraph and GraphLab. For instance, when $n = 192$, GRAPE is on average 12094 and 1329 times faster than Giraph and GraphLab, respectively. (b) Blogel is faster than GRAPE in some cases, *e.g.,* 3.5s vs. 17.9s over UKWeb when $n = 192$. This is because Blogel embeds the computation of CC in its graph partition phase as precomputation, while this graph partition cost (on average 357 seconds using its built-in Voronoi partition) is *not* included in its response time. In other words, without taking advantage of precomputation, the performance of GRAPE is already comparable to the near "optimal" case reported by Blogel.

(3) Sim. Fixing $|Q| = (6, 10)$, *i.e.,* patterns Q with 6 nodes and 10 edges, we evaluated graph simulation over DBpedia and Friendster. As shown in Figures 5g-5h, (a) GRAPE consistently outperforms Giraph, GraphLab and Blogel over all queries. It is 109, 8.3 and 45.2 times faster over Friendster, and 136.7, 5.8 and 20.8 times faster over DBpedia on average, respectively, when $n = 192$. (b) GRAPE scales better with the number n of workers than the others. (c) GRAPE takes at most 21 supersteps, while Giraph, GraphLab and Blogel take 38, 38 and 40 supersteps, respectively. This empirically validates the convergence guarantee of GRAPE under monotonic status-variable updates and its positive effect on reducing parallel and communication cost.

*(4) Collaborative filtering (*CF*).* We used movieLens [4] with a training set $|E_T| = 90\%|E|$. We compared GRAPE with the built-in CF code in GraphLab, and with CF programs implemented for Giraph and Blogel. Note that CF favors "vertex-centric" programming since each node only needs to exchange data with their neighbors, as indicated by that GraphLab and Giraph outperform Blogel. Nonetheless, Figure 5i shows that GRAPE is on average 4.1, 2.6 and 12.4 times faster than Giraph, GraphLab and Blogel, respectively. Moreover, it scales well with n.

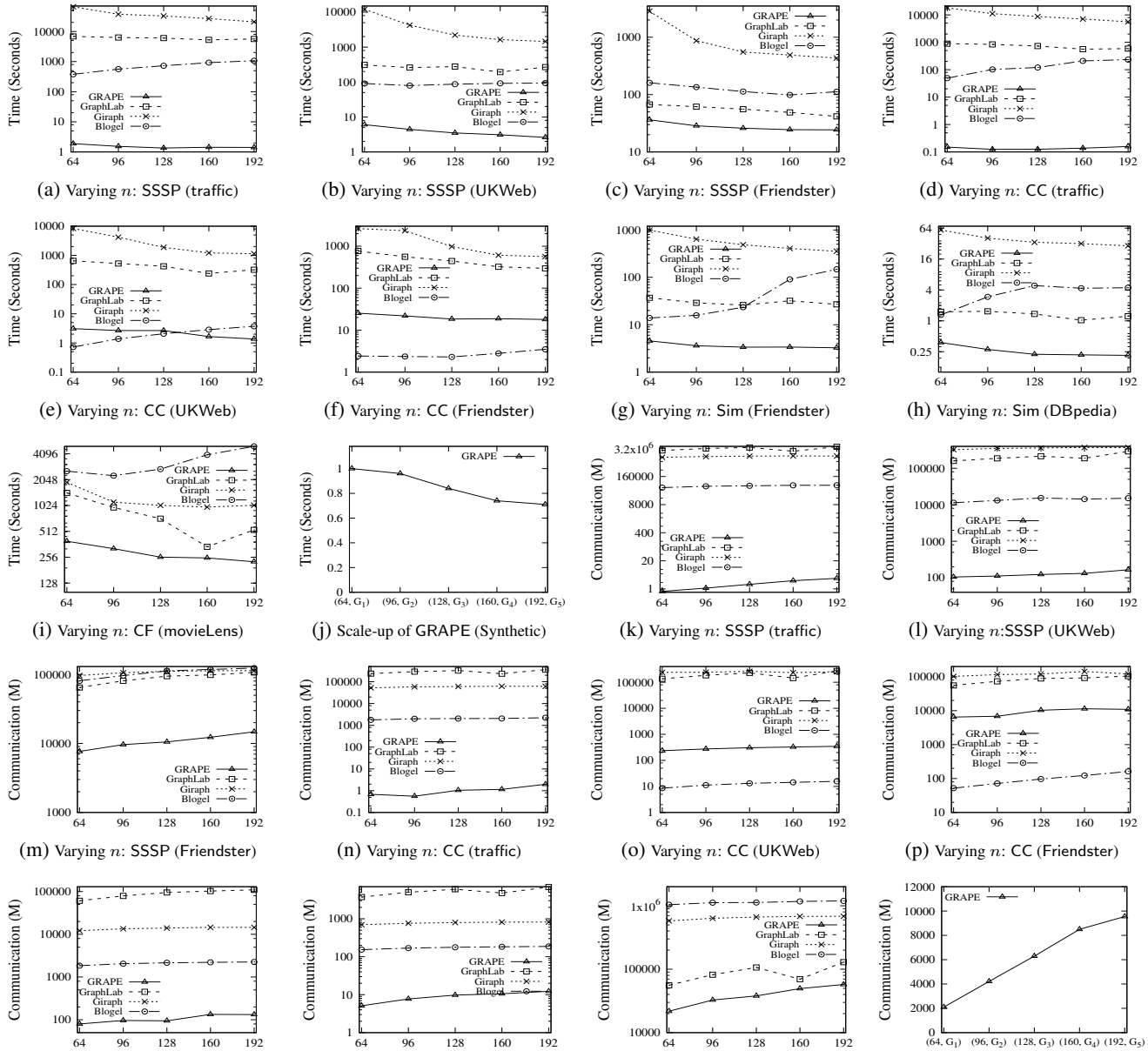
(5) Scale-up of GRAPE. The speed-up of a system may degrade over more workers [26]. We thus evaluate the scale-up of GRAPE, which measures the ability to keep the same performance when both the size of graph G (denoted as $(|V|, |E|)$) and the number n of workers increase proportionally. We varied n from 64 to 192, and for each n, deployed GRAPE over a synthetic graph. The graph size varies from $(50M, 500M)$ to $(250M, 2.5B)$ (denoted as G_5), with fixed ratio between edge number and node number and proportional to n . The scale up at *e.g.,* $(128, G_3)$ is the ratio of the time using 64 workers over G_1 to its counterpart using 128 workers over G_3. As shown in Fig. 5j, GRAPE preserves a reasonable scale-up (close to linear scale-up, the optimal scale-up).

Compared to single-threaded computation, GRAPE incurs extra communication overhead, just like other parallel systems. However, large graphs such as UKWeb are beyond the capacity of a single machine, and parallel computation is a must for such graphs.

Exp-2: Communication cost. The communication cost (in bytes) reported by Giraph, GraphLab and Blogel depends on their own implementation of message blocks and protocols [19]. For a fair comparison, we tracked the total bytes sent by each machine during a run, by monitoring the system file /proc/net/dev, following [19].

In the same setting as Exp-1, Figures 5l-5t report the communication costs of the systems. We observe that Giraph and GraphLab ship roughly the same amount of messages. GRAPE ships much less data than Giraph and GraphLab. On datasets excluding traffic, with 192 workers, it ships on average 0.095%, 0.62%, 0.3%, and 26.2% of the data shipped for SSSP, Sim, CC and CF by Giraph (GraphLab), respectively, and reduces cost up to 6 orders of magnitude on traffic! While it ships more data than Blogel for CC due to the precomputation of Blogel, it only ships 1.9%, 6.2% and 4.8% of the data shipped by Blogel for SSSP, Sim and CF, respectively.

(1) SSSP. Figures 5k-5m show that both GRAPE and Blogel incur communication costs that are orders of magnitudes less than those of GraphLab and Giraph. This is because vertex-centric programming incurs a large number of inter-vertex messages. Both block-centric programs (Blogel) and PIE programs (GRAPE) effectively



(a) Varying n: SSSP (traffic) (b) Varying n: SSSP (UKWeb) (c) Varying n: SSSP (Friendster) (d) Varying n: CC (traffic)

(e) Varying n: CC (UKWeb) (f) Varying n: CC (Friendster) (g) Varying n: Sim (Friendster) (h) Varying n: Sim (DBpedia)

(i) Varying n: CF (movieLens) (j) Scale-up of GRAPE (Synthetic) (k) Varying n: SSSP (traffic) (l) Varying n:SSSP (UKWeb)

(m) Varying n: SSSP (Friendster) (n) Varying n: CC (traffic) (o) Varying n: CC (UKWeb) (p) Varying n: CC (Friendster)

(q) Varying n: Sim (Friendster) (r) Varying n: Sim (DBpedia) (s) Varying n: CF (movieLens) (t) SSSP (Synthetic)

Figure 5: Efficiency and communication cost of GRAPE

reduce unnecessary messages, and trigger inter-block messages only when necessary. Moreover, GRAPE ships 0.9% and 10% of the data shipped by Blogel over UKWeb and Friendster, respectively. Indeed, GRAPE ships only updated values. This significantly reduces the amount of messages that need to be shipped.

(2) CC. Figures 5n-5p show similar improvement of GRAPE over GraphLab and Giraph. It ships on average 0.17% of the data shipped by Giraph and GraphLab. As Blogel precomputes CC (see Exp-1(2)), it ships little data. This said, GRAPE is not far worse than the near "optimal" case of Blogel, sometimes better.

(3) Sim. Figures 5q and 5r report the communication cost for graph simulation over Friendster and DBpedia, respectively. One can see that GRAPE ships substantially less data, *e.g.,* on average 0.9%, 0.1% and 4.9% of the data shipped by Giraph, GraphLab and Blogel, respectively. Observe that the communication cost of Blogel is much higher than that of GRAPE, even though it adopts inter-block communication. This shows that the extension of vertex-centric to block-centric by Blogel has limited improvement for more complex queries. GRAPE works better than these systems by employing incremental IncEval to reduce excessive messages.

(4) CF. Figure 5s reports the result for CF over movieLens. On average, GRAPE ships 5.6%, 43.3% and 3.2% of the data shipped by Giraph, GraphLab and Blogel, respectively.

(5) Communication cost (synthetic). In the same setting as Figure 5j, Figure 5t reports the communication cost for SSSP over large synthetic graphs. It takes higher cost over larger graphs and more workers due to increased "border nodes", as expected. The results for other algorithms are consistent and hence not shown.

Summary. We find the following. (1) Over traffic [5], GRAPE is on average 4, 3 and 2 orders of magnitude faster than Giraph, GraphLab and Blogel for SSSP, respectively, with 192 processors,

due to the large diameter of the graph. On other real-life graphs excluding traffic, GRAPE is on average 484, 36 and 15 times faster than the three systems for SSSP, 151, 6.8 and 16 times for Sim, and 4.6, 2.6 and 12.4 times for CF, respectively, when the number of workers ranges from 64 to 192. For CC, it is 1377 and 212 times faster than Giraph and GraphLab, respectively, and is comparable to the "optimal" case of Blogel. (2) In the same setting (excluding traffic), GRAPE ships on average 0.07%, 0.12% and 1.7% of the data shipped across machines by Giraph, GraphLab and Blogel for SSSP, 0.89%, 0.14% and 4.9% for Sim, 5.6%, 43.3% and 3.2% for CF, respectively. When traffic is also included, GRAPE outperforms these systems by up to 6 orders of magnitude in communication cost for SSSP. For CC, it incurs 0.23% and 0.3% of data shipment of Giraph and GraphLab, and is comparable with "optimized" Blogel. (3) GRAPE demonstrates good scale-up when using more workers, since its incremental computation mitigates the impact of more border nodes and fragments. Moreover, incremental steps effectively reduce unnecessary recomputation.

5. CONCLUSION

The main objective of GRAPE is to simplify parallel programming for graph computations, from "think parallel" to "think sequential". For users who are used to conventional programming, they can start with (existing) sequential algorithms, add declarations for handling messages, and let GRAPE parallelize the computation across a cluster of machines. Moreover, GRAPE guarantees to converge at correct answers under a general condition as long as it is provided with correct sequential algorithms, and it inherits optimization strategies developed for sequential graph algorithms.

As proof of concept (PoC), we have deployed and evaluated GRAPE at three companies. At a large online payment company, GRAPE serves as the graph computing infrastructure supporting its financial risk control system. The company employs graphs in which vertices denote customers, and edges represent transactions and associations with other customers; it needs to evaluate the customers and assign a credit. The company used to deploy its system on Neo4j + Hive + Spark. However, none of the systems can process the tasks alone; the workflow spans three systems and takes 15 minutes on average for a single query. In contrast, GRAPE provides a unified solution for this scenario. It supports real-time ad-hoc queries without the need to couple with other systems. It improves the performance of financial risk analyses: it is 9.0 times faster in graph batch ingesting and streaming, 128.8 times faster in association analysis, and is faster by up to 5 orders of magnitude in batch processing of real-life business applications.

GRAPE works well for other applications. We have also carried out PoC at a big-data service company and a telecommunication service company. The results are consistent and very promising.

We are currently extending GRAPE to support a new parallel model that adaptively switches between synchronous and asynchronous models, to reduce stragglers and stale computations.

Acknowledgments. Fan, Cao, Xu and Yu are supported in part by 973 Program 2014CB340302, ERC 652976, EPSRC EP/M025268/1, NSFC 61421003 and 61602023, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Wu is supported in part by NSF IIS-1633629.

6. REFERENCES

[1] DBpedia. *http://wiki.dbpedia.org/Datasets.*
[2] Friendster. *https://snap.stanford.edu/data/com-Friendster.html.*
[3] Giraph. *http://giraph.apache.org/.*
[4] Movielens. *http://grouplens.org/datasets/movielens/.*
[5] Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml.*
[6] UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/*, 2006.
[7] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, CMU, 2005.
[8] K. Andreev and H. Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.
[9] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *SIGKDD*, pages 1456–1465, 2014.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.
[11] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.
[12] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.
[13] W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang. GRAPE: Parallelizing sequential graph computations. *PVLDB*, 10(12):1889–1892, 2017.
[14] 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.
[15] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3), 1987.
[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USENIX*, 2012.
[17] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.
[18] I. Grujic, S. Bogdanovic-Dinic, and L. Stoimenov. Collecting and analyzing data from E-Government Facebook pages. In *ICT Innovations*, 2014.
[19] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *VLDB*, 7(12), 2014.
[20] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[21] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3), 1996.
[22] M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.
[23] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.
[24] 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.
[25] 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.
[26] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what cost? In *HotOS*, 2015.
[27] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
[28] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.
[29] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.
[30] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, 2017.
[31] 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), 2013.
[32] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
[33] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Vol A*. 1990.
[34] J. Vinagre, A. M. Jorge, and J. Gama. Fast incremental matrix factorization for recommendation with positive-only feedback. In *UMAP*, 2014.
[35] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.
[36] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2), 2017.
[37] 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.