GRAPE: Conducting Parallel Graph Computations without Developing Parallel Algorithms

Wenfei Fan[1,2], Jingbo Xu[1,2], Xiaojian Luo[2], Yinghui Wu[3], Wenyuan Yu[2], Ruiqi Xu[1]
[1]*University of Edinburgh* [2]*Beihang University* [3]*Washington State University*
{wenfei@inf, jingbo.xu@}ed.ac.uk, luoxiaojian@buaa.edu.cn yinghui@eecs.wsu.edu
yuwenyuan@act.buaa.edu.cn, Ruiqi.Xu@ed.ac.uk

Abstract

Developing parallel graph algorithms with correctness guarantees is nontrivial even for experienced programmers. Is it possible to parallelize existing sequential graph algorithms, without recasting the algorithms into a parallel model? Better yet, can the parallelization guarantee to converge at correct answers as long as the sequential algorithms provided are correct? GRAPE *tackles these questions, to make parallel graph computations accessible to a large group of users. This paper presents (a) the parallel model of* GRAPE*, based on partial evaluation and incremental computation, and (b) a performance study, showing that* GRAPE *achieves performance comparable to the state-of-the-art systems.*

1 Introduction

The need for graph computations is evident in transportation network analysis, knowledge extraction, Web mining, social network analysis and social media marketing, among other things. Graph computations are, however, costly in real-life graphs. For instance, the social graph of Facebook has billions of nodes and trillions of edges [16]. In such a graph, it is already expensive to compute shortest distances from a single source, not to mention graph pattern matching by subgraph isomorphism, which is intractable in nature.

To support graph computations in large-scale graphs, several parallel systems have been developed, *e.g.,* Pregel [23], GraphLab [22], Giraph++ [29], GraphX [15], GRACE [33], GPS [27] and Blogel [34], based on MapReduce [8] and (variants of) BSP (Bulk Synchronous Parallel) models [30]. These systems, however, do not allow us to reuse existing sequential graph algorithms, which have been studied for decades and are well optimized. To use Pregel [23], for instance, one has to "think like a vertex" and recast existing algorithms into a vertex-centric model; similarly when programming with other systems, *e.g.,* [34], which adopts vertex-centric programming by treating blocks as vertices. The recasting is nontrivial for people who are not very familiar with the parallel models. This makes parallel graph computations a privilege of experienced users only.

Is it possible to make parallel graph computations accessible to users who only know conventional graph algorithms covered in undergraduate textbooks? Can we have a system such that given a graph computation problem, we can "plug in" its existing sequential algorithms for the problem as a whole, without recasting or "thinking in parallel", and the system automatically parallelizes the computation across multiple processors? Moreover, can the system guarantee that the parallelization terminates and converges at correct answers as long as the sequential algorithms plugged in are correct? Furthermore, can the system inherit optimization techniques

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

well developed for sequential graph algorithms, such as indexing and pruning? Better yet, despite the ease of programming, can the system achieve performance comparable to the state-of-the-art parallel graph systems?

These questions motivate us to develop GRAPE, a parallel GRAPh Engine [13]. GRAPE advocates a parallel model based on simultaneous fixpoint computation that starts with partial evaluation and takes incremental computation as the intermediate consequence operator. It answers all the questions above in the affirmative. As a proof of concept, we have developed a preliminary implementation of GRAPE [12].

This paper presents (a) a brief introduction to the parallel model of GRAPE, and (b) a performance study of the system, using real-life larger than those that [13] experimented with.

2 Parallelizing Sequential Graph Algorithms

We present the programming model and parallel computation model of GRAPE, illustrating how GRAPE parallelizes sequential graph algorithms. We encourage the interested reader to consult [13] for more details.

2.1 Programming Model

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 queries $Q \in \mathcal{Q}$ with GRAPE, a user only needs to specify three functions as follows.

PEval: an 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: an incremental algorithm for $\mathcal{Q}$ that given Q, G, $Q(G)$ and updates ΔG to G, 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 that collects partial answers computed locally at each worker by PEval and IncEval (see Section 2.2), and assembles them into complete answer $Q(G)$. It is typically straightforward.

The three 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* $\vec{x}$ for a set of nodes and edges. As will be seen shortly, these variables are the candidates to be updated by incremental steps IncEval.
- *Aggregate function*. PEval 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.

From these we can see that programming with GRAPE is as simple as MapReduce, if not simpler. Indeed, GRAPE asks the users to provide three functions, where PEval and IncEval are existing sequential algorithms without recasting them into a new model, unlike MapReduce, and Assemble is typically a simple function.

Graph partition. GRAPE supports data-partitioned parallelism. It allows users to pick an (existing) graph partition strategy $\mathcal{P}$ registered in GRAPE, and partitions a (possibly big) graph G into smaller fragments.

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

Given a number m, strategy $\mathcal{P}$ partitions G into *fragments* $\mathcal{F} = (F_1, \ldots, F_m)$ such that each $F_i = (V_i, E_i, L_i)$ is a subgraph of G, $E = \bigcup_{i \in [1,m]} E_i$, and $V = \bigcup_{i \in [1,m]} V_i$. Here F_i is a *subgraph of* G if $V_i \subseteq V$, $E_i \subseteq E$, and for each node $v \in V_i$ (resp. edge $e \in E_i$), $L_i(v) = L(v)$ (resp. $L_i(e) = L(e)$).

Here $\mathcal{P}$ can be any partition strategy, *e.g.,* vertex-cut [20] or edge cut [6]. When $\mathcal{P}$ is edge-cut, denote by

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

For vertex-cut, $\mathcal{F}.O$ and $\mathcal{F}.I$ correspond to exit vertices and entry vertices, respectively.

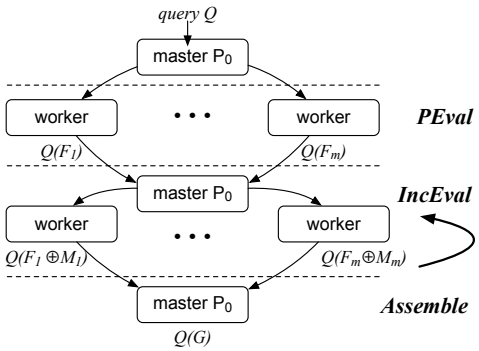


Figure 1: Workflow of GRAPE

2.2 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.

Partial evaluation and incremental computation. We start with basic ideas behind GRAPE parallelization.

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 [19]. 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.

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, and so is O_i. Better still, the incremental 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 [11, 26].

Model. Following BSP [30], given a query $Q \in \mathcal{Q}$ at master P_0, GRAPE answers Q in the partitioned graph G. It posts the same query 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 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 of update parameters at fragment* F_i, denoted by $C_i.\bar{x}$. More specifically, $C_i.\bar{x}$ consists of status variables associated with a set C_i of nodes and edges within d-hops of nodes in $F_i.O$. Here d is an integer determined by query Q only, specified in function PEval. In particular, when $d = 0$, C_i is $F_i.O$.

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 set 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 *changed* parameter values.

(2) Incremental computation (**IncEval**). GRAPE iterates the following supersteps until it terminates. Following BSP, each superstep 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*, in parallel for $i \in [1, m]$.

At the end of the process of IncEval, 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 any update parameter of F_i. If so, GRAPE pulls partial results from all workers, and applies Assemble to group together the partial results and get the final result at P_0, denoted by $\rho(Q, G)$. It returns $\rho(Q, G)$ and terminates.

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 IncEval as the intermediate consequence operator, as follows:

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

where $i \in [1, m]$, 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 $\bar{x}_i$, and M_i is a message indicating changes to $\bar{x}_i$. 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 $i \in [1, m]$. Function Assemble is then invoked to combine all partial answers $R_i^{r_0}$ and get the final answer $\rho(Q, G)$.

Convergence. To characterize the correctness of the fixpoint computation, we use the following notations. (a) A sequential algorithm PEval for $\mathcal{Q}$ is *correct* if given all queries $Q \in \mathcal{Q}$ and graphs G, it terminates and returns $Q(G)$. (b) A sequential incremental algorithm IncEval for $\mathcal{Q}$ is *correct* if given all $Q \in \mathcal{Q}$, graphs G, old output $Q(G)$ and updates ΔG to G, it computes changes ΔO to $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$. (c) Assemble is *correct for $\mathcal{Q}$ w.r.t. $\mathcal{P}$* if when GRAPE with PEval, IncEval and $\mathcal{P}$ terminates at superstep r_0, $\mathsf{Assemble}(Q(F_1[\bar{x}_1^{r_0}]), \ldots, Q(F_m[\bar{x}_m^{r_0}])) = Q(G)$, where $\bar{x}_i^{r_0}$ denotes the values of parameters $C_i.\bar{x}_i$ at round r_0. (d) We say that GRAPE *correctly parallelizes* a PIE program ρ with a partition strategy $\mathcal{P}$ if for all queries $Q \in \mathcal{Q}$ and graphs G, GRAPE guarantees to reach a fixpoint such that $\rho(Q, G) = Q(G)$.

It is shown [13] that under BSP, GRAPE correctly parallelizes a PIE program ρ for a graph computation problem $\mathcal{Q}$ if (a) its PEval and IncEval 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 variables $x \in C_i.\bar{x}$, $i \in [1, m]$, (a) the values of x are 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)), and x is updated "monotonically" following p_x (condition (b)).

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 $v \in V$.

Adopting edge-cut partition [6], GRAPE takes the set $F_i.O$ of "border nodes" as C_i at each worker P_i, *i.e.,* nodes with edges across different fragments. The PIE program of SSSP in GRAPE specializes three functions: (1) a standalone sequential algorithm for SSSP as PEval, *e.g.,* our familiar Dijkstra's algorithm [14], to compute $Q(F_i)$ as partial evaluation; (2) a bounded sequential incremental algorithm of [25] as IncEval that computes $Q(F_i \oplus M_i)$, where messages M_i include updated (smaller) $\mathsf{dist}(s, u)$ (due to new "shortcut" from s) for border nodes u; and (3) Assemble that takes a union of the partial answers $Q(F_i)$ as $Q(G)$.

(1) *PEval*. As shown in Fig. 2a, PEval (lines 1-14) is verbally identical to Dijsktra's algorithm [14]. One only

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 $v \in V_i$

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

/*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 Q
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. $Q(F_i) := \{\mathsf{dist}(s, v) \mid v \in V_i\}$

(a) PEval for SSSP

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

Declare: 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 **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. $Q(F_i) := \{\mathsf{dist}(s, v) \mid v \in V_i\}$

(b) IncEval for SSSP

Figure 2: GRAPE for SSSP

needs to declare status variable as an integer variable $\mathsf{dist}(s, v)$ for each node v, initially ∞ (except $\mathsf{dist}(s, s)$ = 0) and (a) update parameters (in message M_i) 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 (b) min as an aggregate function (**aggregateMsg**). If there are multiple values for the same $\mathsf{dist}(s, v)$, the smallest value is taken by the linear order on integers.

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.

(2) *IncEval*. We give IncEval in Fig. 2b. It is the sequential incremental algorithm for SSSP in [26], in response to 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 [26]). 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.

The only changes to the algorithm of [26] are underlined in Fig. 2b. Following [26], 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 (the While and For loops).

(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 GRAPE 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 decreases or remains unchanged. There are finitely many such variables. Moreover, $\mathsf{dist}(s, v)$ is the shortest distance from s to v as warranted by the sequential algorithms [14, 26] (PEval and IncEval). □

Expressive power. The simple parallel model of GRAPE does not come with a price of degradation in the functionality. More specifically, following [31], we say that a parallel model $\mathcal{M}_1$ can *optimally simulate* model $\mathcal{M}_2$ if there exists a compilation algorithm that transforms any program with cost C on $\mathcal{M}_2$ to a program with cost $O(C)$ on $\mathcal{M}_1$. The cost includes computational cost and communication cost.

As shown in [13], GRAPE optimally simulates parallel models MapReduce [8], BSP [30] and PRAM (Parallel Random Access Machine) [31]. That is, all algorithms in MapRedue, BSP or PRAM with n workers can be simulated by GRAPE using n processors with the same number of supersteps and memory cost. Hence algorithms developed for graph systems based on MapReduce or BSP, *e.g.,* Pregel, GraphLab and Blogel, can be readily migrated to GRAPE without extra cost. We have established a stronger result, *i.e.,* the simulation result

above holds in the message-passing model described above, referred to as the designated message model in [13].

As promised, GRAPE has the following unique features. (1) For a graph computation problem $\mathcal{Q}$, GRAPE allows one to plug in existing sequential algorithms PEval and IncEval, subject to minor changes, and it automatically parallelizes the computation. (2) Under a monotone condition, the parallelization guarantees to converge at the correct answer in any graph, as long as the sequential algorithms PEval and IncEval are correct. (3) MapReduce, BSP and PRAM [31] can be optimally simulated by GRAPE. (4) GRAPE easily capitalizes on existing optimization techniques developed for sequential graph algorithms, since it plugs in sequential algorithms as a whole, and executes these algorithms on entire graph fragments. (5) GRAPE reduces the costs of iterative graph computations by using IncEval, to minimize unnecessary recomputations. The speedup is particularly evident when IncEval is bounded [26], localizable or relatively bounded (see [9] for the latter two criteria).

2.3 Programming with GRAPE

As examples, we outline PIE programs for graph pattern matching (Sim and SubIso), connectivity (CC) and collaborative filtering (CF). We will conduct experimental study with this variety of computations. We adopt edge-cut [6] for the PIE programs below. PIE programs under vertex-cut [20] can be developed similarly.

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 [18], referred to as $Q(G)$. If G does not match Q, $Q(G)$ is the empty set. Graph simulation is in $O((|V_Q| + |E_Q|)(|V| + |E|))$ time [10, 18].

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. GRAPE takes the sequential simulation algorithm of [18] as PEval to compute $Q(F_i)$ in parallel. PEval declares a Boolean variable $x_{(u,v)}$ for each query node u in V_Q and each node v in F_i, indicating whether v matches u, initialized true. It takes the set $F_i.I$ of "border nodes" as candidate set C_i, and $C_i.\bar{x}_{(u,v)}$ as the set of update parameters at F_i. 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. That is, it uses min as the aggregate function, taking the order false $\prec$ true. GRAPE identifies those variables that become false, groups them into messages M_j, and sends M_j to P_j.

IncEval is the *semi-bounded* sequential incremental algorithm of [11]: its cost is decided by the sizes of "updates" $|M_i|$ and changes necessarily checked by all incremental algorithms for Sim, not by $|F_i|$.

The process guarantees to terminate since the update parameters $C_i.\bar{x}$'s are monotonically updated. At this point, Assemble simply takes $Q(G) = \bigcup_{i \in [1,n]} Q(F_i)$, the union of the simulation relation at each F_i.

Subgraph isomorphism. Here a *match* of pattern Q in graph G is a subgraph of G that is isomorphic to Q, and the problem is to compute the set $Q(G)$ of all matches of Q in G. The problem is intractable.

GRAPE parallelizes subgraph isomorphism with *two* supersteps, one for PEval and the other for IncEval. (a) PEval is a sequential Breadth First Search (BFS) algorithm that "fetches" a set $\bigcup_{v \in F_i.I} N_{d_Q}(v)$ of nodes and edges at F_i in parallel. Here $N_{d_Q}(v)$ is the set of nodes and edges that are within d_Q hop of v, and d_Q is the diameter of pattern Q, *i.e.,* the length of the *undirected* shortest path between any two nodes in Q. PEval declares such d_Q-neighbors as $C_i.\bar{x}$ at each F_i, and identifies the update parameters via BFS. (b) IncEval is simply VF2, the sequential algorithm of [7] for subgraph isomorphism. It computes $Q(F_i \oplus M_i)$ at each worker P_i in parallel, in fragment F_i extended with d_Q-neighbor of each node in $F_i.I$. (c) Assemble takes the union of all partial matches computed by IncEval from all workers. The correctness is assured by VF2 and the locality of subgraph isomorphism: a pair (v, v') of nodes in G is in a match of Q only if v is in the d_Q-neighbor of v'.

When message $\bigcup_{v \in F_i.I} N_{d_Q}(v)$ is large, GRAPE further paralellizes PEval by expanding $N_{d_Q}(v)$ step by step, one hop at each step, to reduce communication cost and stragglers. More specifically, IncEval is revised

such that it identifies new "border nodes", and finds the 1-hop neighbor of each border node via incremental BFS to propagate the changes. This proceeds until $\bigcup_{v \in F_i.I} N_{d_Q}(v)$ is in place, and at this point VF2 is triggered.

Graph connectivity (CC). Given an undirected graph G, CC computes all connected components of G. It picks a sequential CC algorithm as PEval, *e.g.,* DFS. It declares an integer variable $v.\mathsf{cid}$ for each node v, recording the component in which v belongs. The update parameters at fragment F_i include $v.\mathsf{cid}$ for all $v \in F_i.I$. At each fragment F_i, PEval computes its local connected components and creates their ids. It exchanges $v.\mathsf{cid}$ for all border nodes $v \in F_i.I$ with neighboring fragments. Given message M_i consisting of the changed $v.\mathsf{cid}$'s for border nodes, IncEval incrementally updates local components in fragment F_i. It merges two components whenever possible, by using min as the aggregate function that takes the smallest component id. The process proceeds until no more changes can be made. At this point, Assemble merges all the nodes having the same cid into a single connected component, and returns all the connected components.

The process terminates since the cids of the nodes are monotonically decreasing. Moreover, IncEval is *bounded*: its cost is a function of the number of $v.\mathsf{cid}$'s whose values are changed in response to M_i.

Collaborative filtering (CF). CF takes as input a bipartite graph G that includes users U and products P, and a set of weighted edges $E \subseteq U \times P$ [21]. (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 $\epsilon(f, E_T) = \min \sum_{((u,p)\in E_T)} (r(u,p) - u.f^T p.f)^2 + \lambda(\|u.f\|^2 + \|p.f\|^2)$. This is typically carried out by the stochastic gradient descent (SGD) algorithm [21], 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 [21] as PEval, and the incremental algorithm ISGD of [32] as IncEval, using master P_0 to synchronize the shared factor vectors $u.f$ and $p.f$. More specifically, PEval declares $v.x = (v.f, t)$ for each node v (initially $\emptyset$), and t bookkeeps a timestamp at which $v.f$ is lastly updated. At fragment F_i, the update parameters include $v.x$ for all $v \in F_i.O$. At P_0, GRAPE maintains $v.x = (v.f, t)$ for all $v \in \mathcal{F}.I = \mathcal{F}.O$. Upon receiving updated $(v.f', t')$ with $t' > t$, it changes $v.f$ to $v.f'$, *i.e.,* takes max as the aggregate function on timestamps. Given M_i at worker P_i, IncEval operates on $F_i \oplus M_i$ by treating M_i as updates to factor vectors of nodes in $F_i.I$, and only modifies affected vectors. The process proceeds until $\epsilon(f, E_T)$ becomes smaller than a threshold, or after a predefined number of step. Assemble simply takes the union of all the vectors from the workers. As long as the sequential SGD algorithm terminates, the PIE program converges at the correct answer since the updates are monotonic with the latest changes as in the sequential SGD.

3 Performance Study

We next empirically evaluate GRAPE, for its (1) efficiency, and (2) communication cost, using real-life graphs larger than those that [13] experimented with. We evaluated the performance of GRAPE compared with 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) movieLens [3], a dense recommendation network (bipartite graph) that has 20 million movie ratings (as weighted edges) between a set of 138000 users and 27000 movies; (2) UKWeb [5], a large Web graph with 133 million nodes and 5 billion edges, (3) DBpedia [1], a knowledge base with 5 million entities and 54 million edges, and in total 411 distinct labels, (4) Friendster [2], a social network with 65 million users and 1.8 billion relations; and (5) traffic [4], an (undirected) US road network with 23 million nodes (locations) and 58 million edges. To make use of unlabeled Friendster for Sim and SubIso, we assigned up to 100 random labels to nodes. We also randomly assigned weights to all the 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 and SubIso, 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 workload without reloading the graph, but GraphLab, Giraph and Blogel require the graph to be reloaded each time a single query is issued, which is costly over large graphs.

Algorithms. We implemented the core functions PEval, IncEval and Assemble for those query classes given in Sections 2.3, registered in the API library of GRAPE. We used XtraPuLP [28] as the default graph partition strategy. We adopted basic sequential algorithms for all the systems without 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 [13] 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 processors (Intel Xeon 2.2GHz) and 128G memory (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 worker used, from 64 to 192. For SSSP and CC, we experimented with UKWeb, traffic and Friendster. For Sim and SubIso, we used over Friendster and DBpedia. We used movieLens for CF as its application in movie recommendation.

(1) SSSP. Figures 3a-3c report the performance of the systems for SSSP over Friendster, UKWeb and traffic, respectively. The results on other graphs are consistent (not shown). From the results we can see the following.

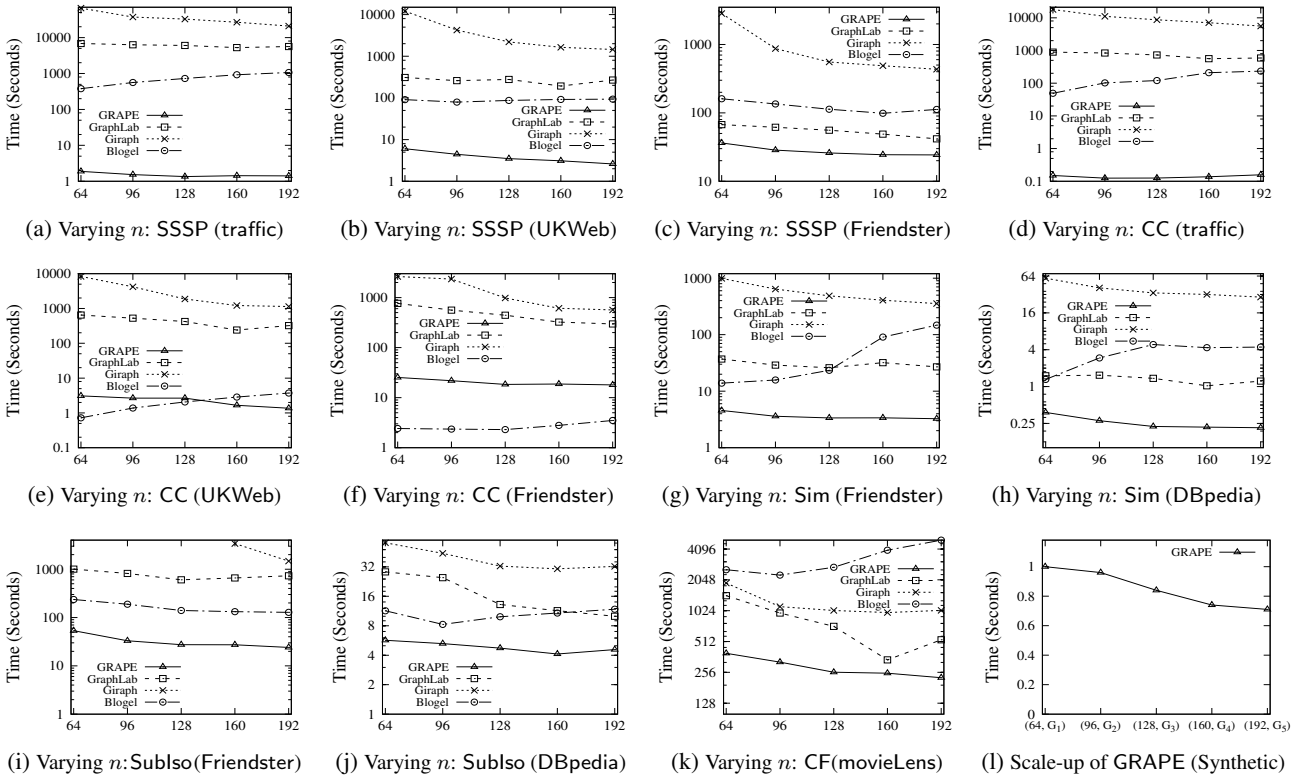
(a) GRAPE outperforms Giraph, GraphLab and Blogel by 14842, 3992 and 756 times, respectively, over traffic with 192 workers (Fig 3a). In the same setting, it is 556, 102 and 36 times faster over UKWeb (Fig. 3b), and 18, 1.7 and 4.6 times faster over Friendster (Fig. 3c). By simply parallelizing sequential 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. As Blogel treats blocks as vertex, the benefit of parallelism is degraded with more blocks.

(b) In all cases, GRAPE take 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 [13] 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 3d-3f report the performance for CC detection, and tell us the following. (a) Both GRAPE and Blogel substantially outperform Giraph and GraphLab. For instance, when $n = 192$, GRAPE is on average



(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:SubIso(Friendster) (j) Varying n: SubIso (DBpedia) (k) Varying n: CF(movieLens) (l) Scale-up of GRAPE (Synthetic)

Figure 3: Efficiency of GRAPE

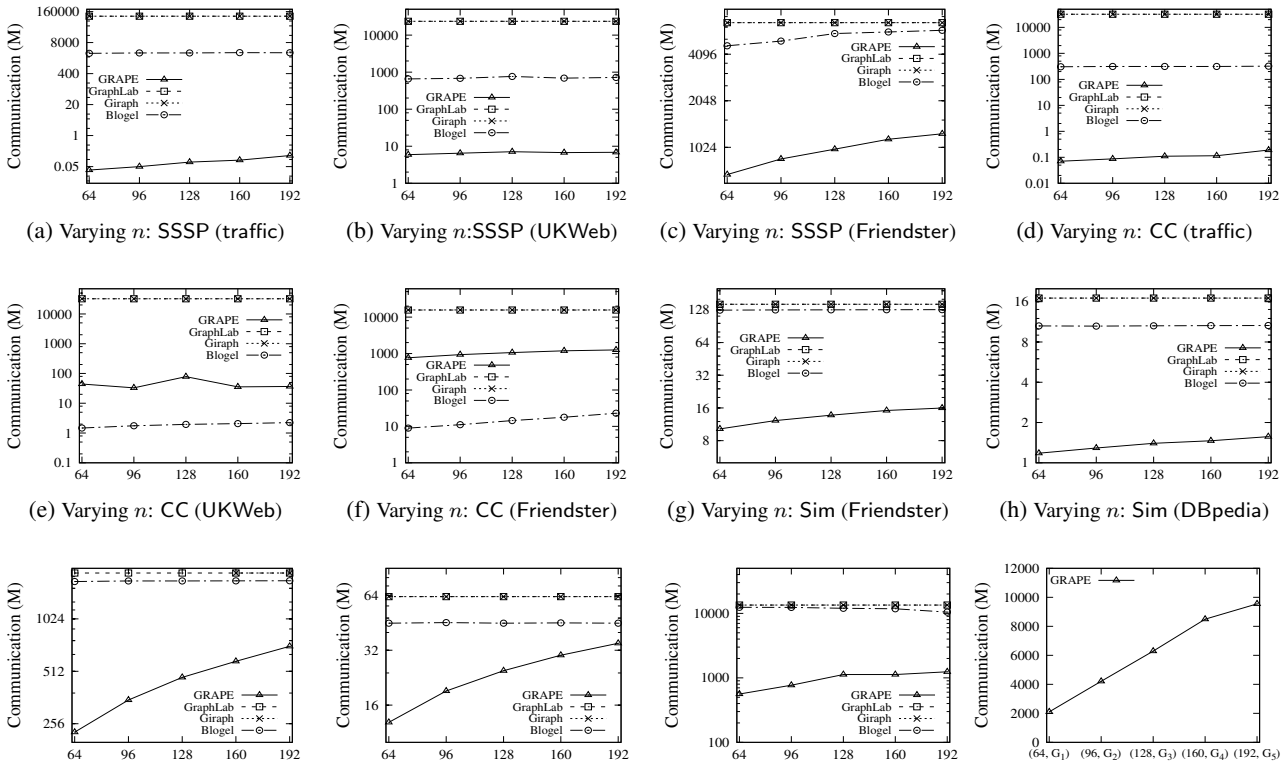
12094 and 1329 times faster than Giraph and GraphLab, respectively. (b) Blogel is faster than GRAPE in some cases, *e.g.,* 3.5s vs. 17.9s over Friendster when $n = 192$. This is because Blogel embeds the computation of CC in its graph partition phase as precomputation, while this graph partition cost (on average 357 seconds using its built-in Voronoi partition) is *not* included in its response time. In other words, without 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 3g-3h, (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) SubIso. Fixing $|Q| = (3, 5)$, we evaluated subgraph isomorphism. As shown in Figures 3i-3j over Friendster and DBpedia, respectively, GRAPE is on average 34, 16 and 4 times faster than Giraph, GraphLab and Blogel when $n = 192$. It is 2.2 and 1.3 times faster when n varies from 64 to 192 over Friendster and DBpedia, respectively. This is comparable with GraphLab that is 1.4 and 2.8 times faster, respectively.

*(5) Collaborative filtering (*CF*).* We used movieLens [3] with a training set $|E_T| = 90\%|E|$. We compared GRAPE with the built-in CF in GraphLab, and with CF 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 3k 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.

(6) Scale-up of GRAPE. The speed-up of a system may degrade over more workers [24]. 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



(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:SubIso(Friendster) (j) Varying n: SubIso (DBpedia) (k) Varying n: CF(movieLens) (l) SSSP (Synthetic)

Figure 4: Communication costs

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. 3l, GRAPE preserves a reasonable scale-up (close to linear scale-up, the optimal scale-up).

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 [17], where Giraph ships less data (in bytes) than GraphLab over specific datasets and queries, and vice versa for others. For a fair and consistent comparison, we report the total number of exchanged messages.

In the same setting as Exp-1, Figure 4 reports the communication costs of the systems. We observe that Giraph and GraphLab ship roughly the same amount of messages. GRAPE incurs much less cost than Giraph and GraphLab. For SSSP with 192 workers, it ships on average 10%, 0.0017%, 45%, and 9.3% of the data shipped for Sim, CC, SubIso and CF by Giraph (GraphLab), respectively, and reduces their cost by 6 orders of magnitude! While it ships more data than Blogel for CC due to the precomputation of Blogel, it only ships 13.5%, 54%, 0.014% and 16% of the data shipped by Blogel for Sim, SubIso, SSSP and CF, respectively.

(1) SSSP. Figures 4a-4c show that both GRAPE and Blogel incurs 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 reduce unnecessary messages, and trigger inter-block messages only when necessary. We also observe that GRAPE ships 0.94% and 17.9% 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 4d-4f show similar improvement of GRAPE over GraphLab and Giraph for CC. It ships on average 0.0017% of the data shipped by Giraph and GraphLab. As Blogel precomputes CC (see Exp-1(2)), it ships little data. Nonetheless, GRAPE is not far worse than the near "optimal" case of Blogel, sometimes better.

(3) Sim. Figures 4g and 4h 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 8.5%, 8.5% and 11.4% 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) SubIso. Figures 4i and 4j report the results for SubIso over Friendster and DBpedia, respectively. On average, GRAPE ships 26%, 26% and 31% of the data shipped by Giraph, GraphLab and Blogel, respectively.

(5) CF. Figure 4k reports the result for CF over movieLens. On average, GRAPE ships 6.5%, 6.5% and 7.3% of the data shipped by Giraph, GraphLab and Blogel, respectively.

(6) Communication cost (synthetic). In the same setting as Figure 3l, Figure 4l reports the communication cost for SSSP. It takes more cost over larger graphs and more workers due to increased "border nodes", as expected.

Summary. Our experimental study verifies the findings in [13] using larger real-life graphs, with some new observations. (1) Over large-scale graphs, GRAPE remains comparable to state-of-the-art systems (Giraph, GraphLab, Blogel) by automatically parallelizing sequential algorithms. (2) GRAPE preserves reasonable scalability and demonstrates good scale-up when using more workers because its incremental computation mitigates the impact of more border nodes and fragments. In contrast, Blogel may take longer time with larger number of workers since its communication cost cancels out the improvement from block-centric parallelism. (3) GRAPE outperforms Giraph, GraphLab and Blogel in communication costs, by orders of magnitude on average.

4 Conclusion

We contend that GRAPE is promising for making parallel graph computations accessible to a large group of users, without degradation in performance or functionality. We have developed stronger fundamental results in connection with the convergence and expressive power of GRAPE, which will be reported in a later publication. We are currently extending GRAPE to support a variant of asynchronous parallel model (ASP).

Acknowledgments. Fan and Xu are supported in part by ERC 652976 and EPSRC EP/M025268/1. Fan, Luo and Yu are supported by Shenzhen Peacock Program 1105100030834361, NSFC 61421003, 973 Program 2014CB340302, Beijing Advanced Innovation Center for Big Data and Brain Computing, and NSFC Foundation for Innovative Research Groups. Wu is supported by NSF IIS 1633629 and Google Faculty Research Award.

References

[1] DBpedia. *http://wiki.dbpedia.org/Datasets.*

[2] Friendster. *https://snap.stanford.edu/data/com-Friendster.html.*

[3] Movielens. *http://grouplens.org/datasets/movielens/.*

[4] Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml.*

[5] UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/*, 2006.

[6] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *SIGKDD*, pages 1456–1465, 2014.

[7] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI*, 26(10):1367–1372, 2004.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[9] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.

[10] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractability to polynomial time. In *PVLDB*, 2010.

[11] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.

[12] W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang. GRAPE: Parallelizing sequential graph computations. In *VLDB (demo)*, 2017. *http://grapedb.io/*.

[13] 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.

[14] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.

[15] 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*, pages 599–613, 2014.

[16] I. Grujic, S. Bogdanovic-Dinic, and L. Stoimenov. Collecting and analyzing data from E-Government Facebook pages. In *ICT Innovations*, 2014.

[17] 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.

[18] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[19] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3), 1996.

[20] M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.

[21] Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[22] 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.

[23] 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.

[24] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what cost? In *HotOS*, 2015.

[25] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[26] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.

[27] S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, 2013.

[28] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, 2017.

[29] 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.

[30] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[31] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Vol A*. 1990.

[32] J. Vinagre, A. M. Jorge, and J. Gama. Fast incremental matrix factorization for recommendation with positive-only feedback. In *UMAP*, 2014.

[33] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[34] 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.