Parallelizing Sequential Graph Computations

Wenfei Fan[1,2], Jingbo Xu[1,2], Yinghui Wu[3], Wenyuan Yu[2], Jiaxin Jiang[4], Zeyu Zheng[5], Bohan Zhang[5], Yang Cao[1], Chao Tian[1,2]

[1]Univ. of Edinburgh [2]Beihang Univ. [3]Washington State Univ. [4]Hong Kong Baptist Univ. [5]Peking Univ.

{wenfei@inf, jingbo.xu@, yang.cao, chao.tian@}ed.ac.uk, yinghui@eecs.wsu.edu, yuwenyuan@act.buaa.edu.cn, jxjian@comp.hkbu.edu.hk, {bohan, zeyu}@pku.edu.cn

ABSTRACT

This paper presents GRAPE, a parallel system for graph computations. GRAPE differs from prior systems in its ability to parallelize existing sequential graph algorithms as a whole. Underlying GRAPE are a simple programming model and a principled approach, based on partial evaluation and incremental computation. We show that sequential graph algorithms can be "plugged into" GRAPE with minor changes, and get parallelized. As long as the sequential algorithms are correct, their GRAPE parallelization guarantees to terminate with correct answers under a monotonic condition. Moreover, we show that algorithms in MapReduce, BSP and PRAM can be optimally simulated on 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.

Keywords

Parallel model; graph computation; partial evaluation; incremental evaluation; scalability

1. INTRODUCTION

Several parallel systems have been developed for graph computations, *e.g.,* Pregel [35], GraphLab [34], Giraph++ [44] and Blogel [50]. These systems, however, require users to recast graph algorithms into their models. While graphs have been studied for decades and a number of sequential algorithms are already in place, to use Pregel, for instance, one has to "think like a vertex" and recast the existing algorithms into a vertex-centric model; similarly when programming with other systems. The recasting is nontrivial for people who are not very familiar with the parallel models. This makes these systems a privilege for experienced users only.

Is it possible to have a system such that we can "plug" sequential graph algorithms into it as a whole (subject to minor changes), and it parallelizes the computation across multiple processors, without drastic degradation in performance or functionality of existing systems?

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA

DOI: http://dx.doi.org/10.1145/3035918.3035942

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

Table 1: Graph traversal on parallel systems

GRAPE. To answer this question, we develop GRAPE, a parallel GRAPh Engine for graph computations such as traversal, pattern matching, connectivity and collaborative filtering. It differs from prior graph 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 minor additions. There is *no need* to revise the logic of the existing algorithms, and it substantially reduces the efforts to "think in parallel". This makes parallel computations accessible to users who know conventional graph algorithms covered in undergraduate textbooks.

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

(3) Graph-level optimization. GRAPE inherits all optimization strategies available for sequential algorithms and graphs, *e.g.,* indexing, compression and partitioning. These strategies 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: 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 (SSSP) over US road network [8], with 24 processors. GRAPE outperforms Giraph, GraphLab and Blogel in both response time and communication costs (see Section 7 for more results).

A principled approach. To see how GRAPE achieves these, we present its underlying principles. Consider a graph G that is partitioned into fragments $(F_1, \ldots, F_n)$, and distributed across n processors $(P_1, \ldots, P_n)$, respectively. Given a query $Q \in \mathcal{Q}$ and a fragmented G, GRAPE computes the answer $Q(G)$ to Q in G based on the following.

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 [28]. 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 processor P_i in GRAPE,

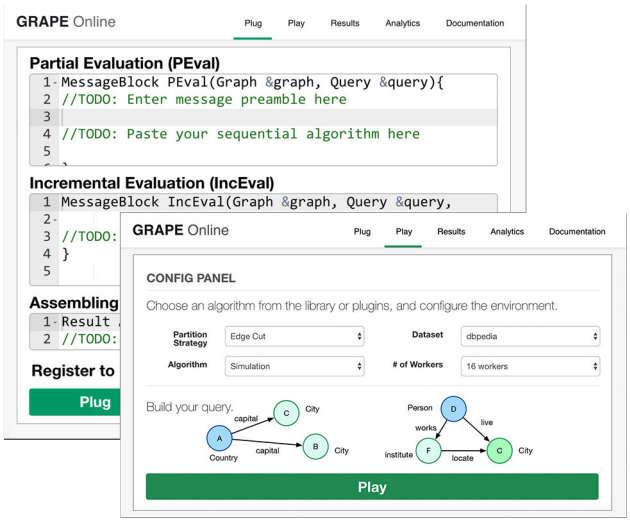


Figure 1: Programming Interface of GRAPE

its local fragment F_i is its known input s, while the data residing at other processors is the yet unavailable input d. GRAPE computes $Q(F_i)$ *in parallel as partial evaluation*.

Incremental computation. GRAPE exchanges selected partial results as messages between processors, and computes $Q(F_i \oplus M_i)$, by treating message M_i to P_i as *updates* to certain status variables associated with nodes and edges in F_i. It *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 typically small, and so is O_i. Better still, the incremental computation may be *bounded*: its cost depends only on sizes of the changes M_i to input F_i and changes ΔO_i to output $Q(F_i)$, not on the size $|F_i|$ of the entire F_i [22,40].

Workflow. Based on these, GRAPE works as follows.

(1) Plug. GRAPE offers a simple programming interface as shown in Fig. 1. For a class $\mathcal{Q}$ of graph queries, users specify three functions: PEval, IncEval and Assemble in the algorithm panel. They are *sequential algorithms* for Q, for partial evaluation, incremental computation and combining partial results, respectively. They can be picked from a library of graph algorithms; the only addition is a specification of messages for communication between processors.

(2) Play. In the configuration panel, users may pick a graph partition strategy and the number n of processors (Fig. 1). Given a query $Q \in \mathcal{Q}$ and a partitioned graph G, GRAPE parallelizes PEval, IncEval and Assemble across n processors, and computes $Q(G)$ in three phases as shown in Fig. 2.

(a) Each processor P_i first executes PEval against its local data F_i, to compute *partial answers* $Q(F_i)$ in parallel. This facilities data-partitioned parallelism via *partial evaluation*.

(b) Then each P_j may exchange partial results with other processors via synchronous message passing. Upon receiving message M_i, P_i *incrementally* computes $Q(F_i \oplus M_i)$ by IncEval, operating on local F_i "updated" by M_i.

(c) The incremental step iterates until no further updates M_i can be made to any F_i. At this point, Assemble pulls partial answers $Q(F_i \oplus M_i)$ for $i \in [1, n]$ and assembles $Q(G)$.

That is, GRAPE parallelizes sequential algorithms as a whole, and conducts a simultaneous fixpoint computation. It guarantees to reach a fixpoint under a monotonic condition if the sequential algorithms are correct for $\mathcal{Q}$. Moreover, it minimizes iterative recomputation by IncEval, and supports graph-level optimization over F_i.

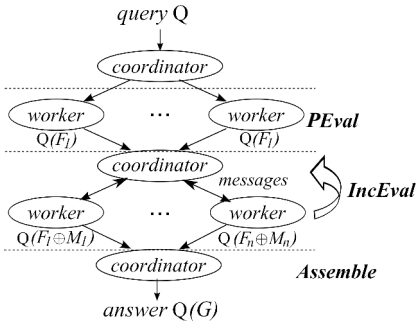


Figure 2: Workflow of GRAPE

Example 1: Consider Single Source Shortest Path (SSSP). Given a graph G with edges labeled with weights, and a source node s in G (as a query Q), it is to find $Q(G)$ including the shortest distance dist(s, v) from s to all nodes v in G.

Using GRAPE, one can pick our familiar Dijkstra's algorithm [23] as PEval, and a bounded sequential incremental algorithm of [39] as IncEval. The only addition is that for each fragment F_i, an integer variable dist(s, v) is declared for each node v, initially ∞ (except dist$(s, s) = 0$). As shown in Fig. 2, PEval first computes $Q(F_i)$; it then repeats IncEval to compute $Q(F_i \oplus M_i)$, where messages M_i include updated (smaller) dist(s, u) (due to new "shortcut" from s) for border nodes u, *i.e.,* nodes with edges across different fragments. GRAPE guarantees the termination of the fixpoint computation, when no more dist(s, v) can be changed to a smaller value. At this point, Assemble takes a union of $Q(F_i)$ as $Q(G)$, which is provably correct (see Section 3 for details).

That is, we take sequential algorithms as PEval, IncEval and Assemble, and specify variables dist(s, v) for updating border nodes. GRAPE takes care of details such as message passing, load balancing and fault tolerance. There is *no need* to recast the entire algorithms into a new model. □

Contributions. We propose GRAPE, from foundation to implementation, to parallelize sequential graph algorithms.

(1) We introduce the parallel model of GRAPE, by combining partial and (bounded) incremental evaluation (Section 3). We also present the programming model of GRAPE. We show how to plug in *existing* sequential algorithms for GRAPE to parallelize the entire algorithms, in contrast to parallelization of instructions or operators [36, 41].

(2) We prove two fundamental results (Section 4): (a) Assurance Theorem guarantees GRAPE to terminate with correct answers under a monotonic condition when its input sequential algorithms are correct; and (b) Simulation Theorem shows that MapReduce [17], BSP (Bulk Synchronous Parallel) [46] and PRAM (Parallel Random Access Machine) [47] can be optimally simulated by GRAPE. Hence algorithms for existing graph systems can be migrated to GRAPE.

(3) We show that a variety of graph computations can be readily parallelized in GRAPE (Section 5). These include graph traversal (shortest path queries SSSP), pattern matching (via graph simulation Sim and subgraph isomorphism SubIso), connected components (CC), and collaborative filtering (CF in machine learning). We show how GRAPE easily parallelizes their sequential algorithms with minor revisions.

(4) We outline an implementation of GRAPE (Section 6). We show how GRAPE supports parallelization, message passing, fault tolerance and consistency. We also show how easily GRAPE implements optimization such as indexing, compression and dynamic grouping, which are not supported by the state-of-the-art vertex-centric and block-centric systems.

(5) We experimentally evaluate GRAPE (Section 7), compared with (a) Giraph, a open-source version of Pregel, (b) GraphLab, an asynchronous vertex-centric system, and (c) Blogel, the fastest block-centric system we are aware of. Over real-life graphs, we find that in addition to the ease of programming, GRAPE achieves comparable performance to the state-of-the-art systems. For instance, (a) GRAPE is 323, 274 and 7.9 times faster than Giraph, GraphLab and Blogel for SSSP, 2.7, 2.6 and 1.7 times for Sim, 1.7, 1.4 and 1.7 times for SubIso, and 1.9, 1.4 and 3.8 times for CF on average, respectively, when the number of processors ranges from 4 to 24. (b) In the same setting, GRAPE ships on average 5.6%, 5.6% and 10% of the data shipped by Giraph, GraphLab and Blogel for SSSP, 1.3%, 1.3% and 1.6% for Sim, 4.7%, 4.7% and 6.5% for SubIso, and 8.1%, 8.1% and 8.7% for CF, respectively. (c) Incremental steps effectively reduce the cost and improve the performance of Sim by 2.6 times. (d) Optimization strategies for sequential algorithms remain effective for GRAPE and improve Sim by 2 times on average.

Related work. The related work is categorized as follows.

Parallel models and systems. Several parallel models have been studied for graphs, *e.g.,* PRAM [47], BSP [46] and MapReduce [17]. PRAM abstracts parallel RAM access over shared memory. BSP models parallel computations in supersteps (including local computation, communication and a synchronization barrier) to synchronize communication among workers. Pregel [35] (Giraph [3]) implements BSP with vertex-centric programming, where a superstep executes a user-defined function at each vertex in parallel. GraphLab [34] revises BSP to pass messages asynchronously. Block-centric models [44,50] extend vertex-centric programming to blocks, to exchange messages among blocks.

Popular graph systems also include GraphX [25], GRACE [49], GPS [42], etc. GraphX [25] recasts graph computation in its distributed dataflow framework as a sequence of join and group-by stages punctuated by map operations over Spark platform. GRACE [49] provides an operator-level, iterative programming model to enhance synchronous BSP with asynchronous execution. GPS [42] implements Pregel with extended APIs and partition strategies. All these systems require recasting of sequential algorithms.

GRAPE adopts the synchronization mechanism of BSP. As opposed to the prior systems, (a) GRAPE aims to parallelize existing sequential algorithms, by combining partial evaluation and incremental computation. (b) As opposed to MapReduce, it highlights data-partitioned parallelism via graph fragmentation. For iterative computations, it does not need to ship the entire state of the graphs in each round [35]. (c) The vertex-centric model of Pregel (synchronized) is a special case of GRAPE, when each fragment is limited to a single vertex. The communications of Pregel are via "interprocessor" messages, and a message from a node often has to go through several supersteps to reach another node. GRAPE reduces excessive messages and scheduling cost of Pregel, since communications within the same fragment are local. GRAPE also facilitates graph-level optimizations that are hard to implement in vertex-centric systems; similarly for GraphLab (asynchronized). (d) Closer to GRAPE are block-centric models [44, 50]. However, the programming interface of [44] is still vertex-centric, and [50] is a mix of vertex-centric and block-centric programming (V-compute and B-compute). The B-compute interface is essentially vertex-centric programming, by treating each block as a vertex. Users have to recast existing sequential algorithms into a new model. In contrast, GRAPE "plugs in" sequential algorithms PEval and IncEval from GRAPE library, and applies them to blocks without recasting. None of the prior systems uses (bounded) incremental steps to speed up iterative computations. No one provides assurance on termination and correctness of parallel graph computations.

Partial evaluation has been studied for certain XML [14] and graph queries [22]. There has also been a host of work on incremental graph computation (*e.g.,* [22,40]). This work makes a first effort to provide a uniform model by combining partial evaluation and incremental computation together, to parallelize sequential graph algorithms as a whole.

Parallelization of graph computations. A number of graph algorithms have been developed in MapReduce, vertex-centric models and others [22,51]. In contrast, GRAPE aims to parallelize existing sequential graph algorithms, without revising their logic and work flow. Moreover, parallel algorithms for MapReduce, BSP (vertex-centric or not) and PRAM can be easily migrated to GRAPE (Section 4.2).

Prior work on automated parallelization has focused on the instruction or operator level [37, 41] by breaking dependencies via symbolic and automata analyses. There has also been work at data partition level [52], to perform multi-level partition ("parallel abstraction") and enable locality-optimized access to adapt to different parallel abstraction. In contrast, GRAPE aims to parallelize sequential algorithms as a whole. It is to make parallel computation accessible to end users, while [37, 41, 52] target experienced developers of parallel algorithms. There have also been tools for translating imperative code to MapReduce, *e.g.,* word count [38]. GRAPE advocates a different approach, by parallelizing the runs of sequential graph algorithms to benefit from data-partitioned parallelism, without translation. This said, the techniques of [37, 38, 41, 52] are complementary to GRAPE.

Simulation results. Prior work has mostly focused on simulations between variants of PRAM with different memory management strategies, to characterize bounds of slowdown for deterministic or randomized solutions [26]. There has also been recent work on simulation of PRAM on MapReduce and BSP [29]. We present optimal deterministic simulation results of MapReduce, BSP and PRAM on GRAPE, adopting the notion of optimal simulations of [47].

2. PRELIMINARIES

We start with a review of basic notations.

Graphs. 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 $L(v)$ (resp. $L(e)$), indicating its content, as found in social networks, knowledge bases and property graphs.

Graph $G' = (V', E', L')$ is called a *subgraph of* G if $V' \subseteq V$, $E' \subseteq E$, and for each node $v \in V'$ (resp. each edge $e \in E'$), $L'(v) = L(v)$ (resp. $L'(e) = L(e)$).

symbols	notations
$\mathcal{Q}$, Q	a class of graph queries, query $Q \in \mathcal{Q}$
G	graph, directed or undirected
P_0, P_i	P_0: coordinator; P_i: workers ($i \in [1, n]$)
$\mathcal{P}$	graph partition strategy
$G_{\mathcal{P}}$	the fragmentation graph of G via $\mathcal{P}$
$\mathcal{F}$	fragmentation $(F_1, \ldots, F_n)$
M_i	messages designated to worker P_i

Table 2: Notations

Subgraph G' is said to be *induced by* V' if E' consists of all the edges in G whose endpoints are both in V'.

Partition strategy. Given a number m, a strategy $\mathcal{P}$ partitions graph 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, $E = \bigcup_{i\in[1,m]} E_i$, $V = \bigcup_{i\in[1,m]} V_i$, and F_i resides at processor P_i. 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 some F_j ($i \neq j$); and
- $\mathcal{F}.O = \bigcup_{i\in[1,m]} F_i.O$, $\mathcal{F}.I = \bigcup_{i\in[1,m]} F_i.I$; $\mathcal{F}.O = \mathcal{F}.I$.

In vertex-cut partition [32], $\mathcal{F}.O$ and $\mathcal{F}.I$ correspond to entry vertices and exit vertices, respectively. We refer to nodes in $F_i.I \cup F_i.O$ as the *border nodes* of F_i *w.r.t.* $\mathcal{P}$.

The *fragmentation graph* $G_{\mathcal{P}}$ of G via $\mathcal{P}$ is an index such that given each node v in $\mathcal{F}.O$ (or $\mathcal{F}.I$), $G_{\mathcal{P}}(v)$ retrieves a set of $(i \mapsto j)$ if $v \in F_i.O$ and $v \in F_j.I$ with $i \neq j$. As will be seen shortly, $G_{\mathcal{P}}$ helps us deduce the directions of messages.

The notations of this paper are summarized in Table 2.

3. PROGRAMMING WITH GRAPE

We start with the parallel model of GRAPE, and then show how to program with GRAPE. Following BSP [46], GRAPE employs a *coordinator* P_0 and a set of m *workers* $P_1, \ldots, P_m$.

3.1 The Parallel Model of GRAPE

Given a partition strategy $\mathcal{P}$ and sequential PEval, IncEval and Assemble for a class $\mathcal{Q}$ of graph queries, GRAPE parallelizes the computations as follows. It first partitions G into $(F_1, \ldots, F_m)$ with $\mathcal{P}$, and distributes 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. It also constructs fragmentation graph $G_{\mathcal{P}}$. Note that G is partitioned *once* for *all queries* $Q \in \mathcal{Q}$ posed on G.

Parallel model. Given $Q \in \mathcal{Q}$, GRAPE computes $Q(G)$ in the partitioned G as shown in Fig. 2. Upon receiving Q at coordinator P_0, GRAPE posts the same Q to all the workers. It adopts synchronous message passing following BSP [46]. Its parallel computation consists of three phases.

(1) Partial evaluation (**PEval**). In the first superstep, upon receiving Q, each worker P_i computes partial results $Q(F_i)$ locally at F_i using PEval, in parallel ($i \in [1, m]$). It also identifies and initializes a set of update parameters for each F_i that records the status of its border nodes. At the end of the process, it generates a message from the update parameters at each P_i and sends it to coordinator P_0 (see Section 3.2).

(2) Incremental computation (**IncEval**). GRAPE iterates the following supersteps until it terminates. Each superstep has two steps, one at P_0 and the other at the workers.

(2.a) Coordinator. Coordinator 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 is no pending message designated for P_i. If so, GRAPE invokes Assemble and terminates (see below). Otherwise, P_0 routes messages from the last superstep to workers (Section 3.2), and triggers the next superstep.

(2.b) Workers. Upon receiving message M_i, worker P_i *incrementally* computes $Q(F_i \oplus M_i)$ with IncEval, by treating M_i as updates, in parallel for all $i \in [1, m]$. It automatically finds the changes to the update parameters in each F_i, and sends the changes as a message to P_0 (see Section 3.3).

GRAPE supports data-partitioned parallelism by *partial evaluation* on local fragments, in parallel by all workers. Its *incremental step* (2.b) speeds up iterative graph computations by reusing the partial results from the last superstep.

(3) Termination (**Assemble**). The coordinator P_0 decides to terminate if there is no change to any update parameters (see (2.a) above). If so, P_0 pulls partial results from all workers, and computes $Q(G)$ by Assemble. It returns $Q(G)$.

We now introduce the programming model of GRAPE. For a class $\mathcal{Q}$ of graph queries, one only needs to provide three core functions PEval, IncEval and Assemble (see Plug Panel of Fig. 1), referred to as *a* PIE *program*. These are conventional sequential algorithms, and can be picked from Library API of GRAPE. We next elaborate a PIE program.

3.2 PEval: Partial Evaluation

PEval takes a query $Q \in \mathcal{Q}$ and a fragment F_i of G as input, and computes partial answers $Q(F_i)$ at worker P_i in parallel for all $i \in [1, m]$. It may be any existing sequential algorithm $\mathcal{T}$ for $\mathcal{Q}$, extended with the following:

- *partial result* kept in a designated variable; and
- *message specification* as its interface to IncEval.

Communication between workers is conducted via messages, defined in terms of *update parameters* as follows.

(1) Message preamble. PEval (a) declares *status variables* $\vec{x}$, and (b) specifies a set C_i of nodes and edges relative to $F_i.I$ or $F_i.O$. The status variables associated with C_i are denoted by $C_i.\bar{x}$, referred to as the *update parameters* of F_i.

Intuitively, variables in $C_i.\bar{x}$ are the candidates to be updated by incremental steps. In other words, messages M_i to worker P_i are *updates* to the values of variables in $C_i.\bar{x}$.

More specifically, C_i is specified by an integer d and S, where S is either $F_i.I$ or $F_i.O$. That is, C_i is the set of nodes and edges within d-hops of nodes in S. If $d = 0$, C_i is $F_i.I$ or $F_i.O$. Otherwise, C_i may include nodes and edges from other fragments F_j of G (see an example in Section 5).

The variables are declared and initialized in PEval. At the end of PEval, it sends the values of $C_i.\bar{x}$ to coordinator P_0.

(2) Message segment. PEval may specify function **aggregateMsg**, to resolve conflicts when multiple messages from different workers attempt to assign different values to the same update parameter (variable). When such a strategy is not provided, GRAPE picks a default exception handler.

(3) Message grouping. GRAPE deduces updates to $C_i.\vec{x}$ for $i \in [1, m]$, and treats them as messages exchanged among workers. More specifically, at coordinator P_0, GRAPE identifies and maintains $C_i.\bar{x}$ for each worker P_i. Upon receiving messages from P_i's, GRAPE works as follows.

(a) Identifying C_i. It deduces C_i for $i \in [1, m]$ by referencing fragmentation graph $G_{\mathcal{P}}$, and C_i remains unchanged in the entire process. It maintains update parameters $C_i.\bar{x}$ for F_i.

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 $v \in V_i$

Message preamble: (designated) /*candidate set C_i is $F_i.O$*/
for each node $v \in V_i$, an integer variable $\mathsf{dist}(s,v)$

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

Message segment: $M_i := \{\mathsf{dist}(s,v) \mid v \in F_i.O\}$;
aggregateMsg = $\mathsf{min}(\mathsf{dist}(s,v))$;

Figure 3: PEval **for** SSSP

(b) Composing M_i. For messages from each P_i, GRAPE (i) identifies variables in $C_i.\bar{x}$ with *changed values*; (ii) deduces their designations P_j by referencing $G_{\mathcal{P}}$; if $\mathcal{P}$ is edge-cut, the variable tagged with a node v in $F_i.O$ will be sent to worker P_j if v is in $F_j.I$ (*i.e.,* if $i \mapsto j$ is in $G_{\mathcal{P}}(v)$); similarly for v in $F_i.I$; if $\mathcal{P}$ is vertex-cut, it identifies nodes shared by F_i and F_j ($i \neq j$); and (iii) it combines all changed variables values designated to P_j into a single message M_j, and sends M_j to worker P_j in the next superstep for all $j \in [1,m]$.

If a variable x is assigned a set S of values from different workers, function **aggregateMsg** is applied to S to resolve the conflicts, and its result is taken as the value of x.

These are automatically conducted by GRAPE, which minimizes communication costs by passing only *updated* variable values. To reduce the workload at the coordinator, alternatively each worker may maintain a copy of $G_{\mathcal{P}}$ and deduce the designation of its messages in parallel.

Example 2: We show how GRAPE parallelizes SSSP. 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 *shortest 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, GRAPE computes $\mathsf{dist}(s,v)$ for all $v \in V$. It adopts edge-cut partition [13]. It deduces $F_i.O$ by referencing $G_{\mathcal{P}}$ and stores $F_i.O$ at each fragment F_i.

As shown in Fig. 3, PEval (lines 1-14) is *verbally identical* to Dijsktra's sequential algorithm [23]. The *only changes* are message preamble and segment (underlined). It declares an integer variable $\mathsf{dist}(s,v)$ for each node v, initially ∞ (except $\mathsf{dist}(s,s) = 0$). It specifies min as **aggregateMsg** to resolve conflicts: if there are multiple values for the same $\mathsf{dist}(s,v)$, the smallest value is taken by the linear order on integers. The update parameters are $C_i.\bar{x} = \{\mathsf{dist}(s,v) \mid v \in F_i.O\}$.

At the end of its process, PEval sends $C_i.\bar{x}$ to coordinator 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 values, deduces their destinations by referencing $G_{\mathcal{P}}$, 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: $Q(F_i \oplus M_i)$

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

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

Figure 4: IncEval **for** SSSP

3.3 IncEval: Incremental Evaluation

Given query Q, fragment F_i, partial results $Q(F_i)$ and message M_i (updates to $C_i.\bar{x}$), IncEval computes $Q(F_i \oplus M_i)$ incrementally, making maximum reuse of the computation of $Q(F_i)$ in the last round. Each time after IncEval is executed, GRAPE treats $F_i \oplus M_i$ and $Q(F_i \oplus M_i)$ as F_i and $Q(F_i)$, respectively, for the next round of incremental computation.

IncEval can take any existing sequential incremental algorithm $\mathcal{T}_\Delta$ for $\mathcal{Q}$. It shares the message preamble of PEval. At at the end of the process, it identifies *changed values* to $C_i.\bar{x}$ at each F_i, and sends the changes as messages to P_0. At P_0, GRAPE composes messages as described in 3(b) above.

Boundedness. Graph computations are typically iterative. GRAPE reduces the costs of iterative computations by promoting *bounded incremental algorithms* for IncEval.

Consider an incremental algorithm $\mathcal{T}_\Delta$ for $\mathcal{Q}$. Given G, $Q \in \mathcal{Q}$, $Q(G)$ and updates M to G, it computes ΔO such that $Q(G \oplus M) = Q(G) \oplus \Delta O$, where ΔO denotes changes to the old output $O(G)$. It is said to be *bounded* if its cost can be expressed as a function in the size of $|\mathsf{CHANGED}| = |\Delta M| + |\Delta O|$, *i.e.,* the size of changes in the input and output [21,40]. Intuitively, $|\mathsf{CHANGED}|$ represents the updating costs inherent to the incremental problem for $\mathcal{Q}$ itself. For a bounded IncEval, its cost is determined by $|\mathsf{CHANGED}|$, not by the size $|F_i|$ of entire F_i, no matter how big $|F_i|$ is.

Example 3: Continuing with Example 2, we give IncEval in Fig. 4. It is the sequential incremental algorithm for SSSP in [40], in response to changed $\mathsf{dist}(s,v)$ for v in $F_i.I$ (here M_i includes changes to $\mathsf{dist}(s,v)$ for $v \in F_i.I$ deduced from $G_{\mathcal{P}}$). Using a queue Que, it starts with M_i, propagates the changes to affected area, and updates the distances (see [40]). The partial result is now the revised distances (line 11).

At the end of the process, IncEval sends to coordinator P_0 updated values of those status variables in $C_i.\bar{x}$, as in PEval. It applies **aggregateMsg** min to resolve conflicts.

The only changes to the algorithm of [40] are underlined in Fig. 4. Following [40], 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). □

3.4 Assemble Partial Results

Function Assemble takes partial results $Q(F_i \oplus M_i)$ and fragmentation graph $G_{\mathcal{P}}$ as input, and combines $Q(F_i \oplus M_i)$ to get $Q(G)$. It is triggered when no more changes can be made to update parameters $C_i.\bar{x}$ for any $i \in [1,m]$.

Example 4: Continuing with Example 3, Assemble (not shown) for SSSP 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 terminates with correct $Q(G)$. The 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. Furthermore, $\mathsf{dist}(s,v)$ is the shortest distance from s to v, as warranted by the correctness of the sequential algorithms [23, 40] (PEval and IncEval). □

Putting these together, one can see that a PIE program parallelizes a graph query class $\mathcal{Q}$ provided with a sequential algorithm $\mathcal{T}$ (PEval) and a sequential incremental algorithm $\mathcal{T}_\Delta$ (IncEval) for $\mathcal{Q}$. Assemble is typically a straightforward sequential algorithm. A large number of sequential (incremental) algorithms are already in place for various $\mathcal{Q}$. Moreover, there have been methods for incrementalizing graph algorithms, to get incremental algorithms from their batch counterparts [9,18]. Thus GRAPE makes parallel graph computations accessible to a large group of end users.

In contrast to existing graph systems, GRAPE plugs in $\mathcal{T}$ and ΔT as a whole, and confines communication specification to the message segment of PEval. Users do not have to think "like a vertex" [34, 35, 44, 50] when programming. As opposed to vertex-centric and block-centric systems, GRAPE runs sequential algorithms on entire fragments. Moreover, IncEval employs incremental evaluation to reduce cost, which is a unique feature of GRAPE. Note that IncEval speeds up iterative computations by minimizing unnecessary recomputation of $Q(F_i)$, *no matter whether it is bounded or not.*

3.5 GRAPE API

GRAPE provides a declarative programming interface for users to plug in the sequential algorithms as UDFs (user-defined functions). Upon receiving (sequential) algorithms, GRAPE registers them as stored procedures in its API library (Fig. 1), and maps them to a query class $\mathcal{Q}$.

In addition, GRAPE can simulate MapReduce. More specifically, GRAPE supports two types of messages:

- *designated* messages from one worker to another; and
- *key-value* pairs (*key, val*), to simulate MapReduce.

The messages generated by PEval and IncEval are marked *key-value* or *designated*. The messages we have seen so far are designated, and GRAPE automatically identifies their destinations at coordinator P_0, as described in Section 3.2.

If the messages are marked *key-value*, GRAPE automatically recognizes the key and value segments by parsing the message declaration in PEval and IncEval. Following MapReduce, it groups the messages by keys at coordinator P_0, and distributes them across m workers, to balance the workload.

4. FOUNDATION OF GRAPE

Below we present the correctness guarantees of the parallel model of GRAPE, and demonstrate the power of GRAPE.

4.1 Correctness of Parallel Model

Intuitively, GRAPE supports a simultaneous fixpoint operator $\phi(R_1, \ldots, R_m)$ over m fragments defined as:

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

where $i \in [1,m]$, r indicates a superstep, R_i^r denotes partial results in step r at worker P_i, $F_i^0 = F_i$, $F_i^r[\bar{x}_i]$ is fragment F_i at the end of superstep r carrying update parameters $C_i.\bar{x}_i$, and M_i indicates changes to $C_i.\bar{x}_i$. The computation proceeds until it reaches r_0 when $R_i^{r_0} = R_i^{r_0+1}$. At this point, $\mathsf{Assemble}(G_\mathcal{P}, R_1^{r_0}, \ldots, R_m^{r_0})$ is computed and returned.

We next prove a correctness guarantee for the simple model with *designated messages*. We start with notations.

(1) We say that GRAPE with PEval, IncEval and $\mathcal{P}$ *terminates* if for all queries $Q \in \mathcal{Q}$ and all graphs G, there exists r_0 such that at superstep r_0, $R_i^{r_0} = R_i^{r_0+1}$ for all $i \in [1,m]$.

(2) Denote by $G[\bar{x}]$ a graph G with update parameters $\bar{x}$. We say that PEval is *correct for* $\mathcal{Q}$ if for all $Q \in \mathcal{Q}$ and graphs G, $\mathsf{PEval}(Q, G[\bar{x}])$ returns $Q(G[\bar{x}])$. Similarly, IncEval is *correct for* $\mathcal{Q}$ if $\mathsf{IncEval}(Q, G[\bar{x}], M, Q(G[\bar{x}]))$ returns $Q(G[\bar{x} \oplus M])$, where $\bar{x} \oplus M$ denotes $\bar{x}$ updated by M.

We say that 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.

(3) We say that PEval and IncEval satisfy the *monotonic condition w.r.t.* $\mathcal{P}$ if for all variables $x \in C_i.\bar{x}$, $i \in [1,m]$ (a) the values of x are computed from values in 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.

Intuitively, condition (a) says that x draws values from a finite domain, and condition (b) says that x is updated "monotonically" following p_x. These ensure that GRAPE parallelization with PEval, IncEval and $\mathcal{P}$ terminate.

For instance, $\mathsf{dist}(s,v)$ in Example 2 can only be changed in the decreasing order (*i.e.,* min for **aggregateMsg**).

Theorem 1 [Assurance Theorem]*: Consider sequential algorithms* PEval, IncEval, Assemble *for a graph query class* $\mathcal{Q}$*, and a partition strategy* $\mathcal{P}$*. If (a)* PEval *and* IncEval *satisfy the monotonic condition* w.r.t. $\mathcal{P}$*, and (b)* PEval, IncEval *and* Assemble *are correct for* $\mathcal{Q}$ w.r.t. $\mathcal{P}$*, then* GRAPE *with* PEval, IncEval, Assemble *and* $\mathcal{P}$ *guarantees to terminate and correctly compute* $Q(G)$ *for all* $Q \in \mathcal{Q}$ *and graphs* G*.* □

Proof: It suffices to show the following. For all Q and G, (1) there exists $r_{(Q,G)}$ such that GRAPE terminates at superstep $r_{(Q,G)}$ with deterministic values $\bar{x}_i^{r_{(Q,G)}}$ for update parameters $C_i.\bar{x}$ in fragments F_i of G for $i \in [1,m]$; and (2) IncEval correctly computes partial answers $Q(F_i[\bar{x}_i^{r_{(Q,G)}}])$ on F_i at superstep $r_{(Q,G)}$. For if these hold, then GRAPE always terminates with correct answers by the correctness of Assemble. We show (1) and (2) by induction on superstep r in the run for Q and G (see Appendix A for details). □

4.2 The Power of GRAPE

GRAPE can readily switch to other parallel models.

Following [47], 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 and communication cost. For GRAPE, these are measured by the running time of PEval, IncEval and Assemble on all the processors, and by the total size of the messages passed among all the processors in the entire process.

We show that GRAPE optimally simulates popular parallel models MapReduce [17], BSP [46] and PRAM [47].

Theorem 2 [Simulation Theorem]*: (1) all BSP algorithms with* n *workers in* t *supersteps can be optimally sim-*

ulated on GRAPE *with n workers in t supersteps, without extra cost in each superstep;*

(2) all MapReduce programs using n processors can be optimally simulated by GRAPE *using n processors; and*

(3) all CREW PRAM algorithms using $O(P)$ total memory, $O(P)$ processors and t time can be run in GRAPE *in $O(t)$ supersteps using $O(P)$ processors with $O(P)$ memory.* □

As a consequence, all algorithms developed for graph systems based on MapReduce and/or BSP can be readily migrated to GRAPE without much extra cost, including Pregel [35], GraphX [25], Giraph++ [44] and Blogel [50].

Below we outline a proof (see Appendix A for details).

Bulk-Synchronous parallel model. A BSP algorithm proceeds in supersteps. Each superstep consists of an input phase, a local computation phase and an output phase. The workers are synchronized between supersteps. The cost of a superstep is expressed as $w + gh + l$, where (a) w is the maximum number of operations by any worker within the superstep; (b) h is the maximum amount of messages sent/received by any workers; (c) g is the communication throughput ratio, or bandwidth inefficiency; and (d) l is the communication latency or synchronization periodicity. We define the throughput and latency for GRAPE similarly.

For Theorem 2(1), each worker of BSP is simulated by a worker in GRAPE. PEval is defined to perform the same local computation in the first superstep of BSP, IncEval simulates the actions of each worker in the later supersteps of the BSP algorithm, and Assemble collects and combines the partial results. Message routing and synchronization control adopt the same strategy of BSP, via designated messages, where the coordinator acts as the synchronization router. One can verify that the simulation does not incur extra cost.

In particular, Pregel [35] assigns a virtual worker to each node (single-vertex fragments). GRAPE reduces its excessive messages, supports graph-level optimization, and employs incremental steps to speed up iterative computation.

MapReduce. A MapReduce program is specified by a Map function and a Reduce function [17]. Its computation is a sequence of map, shuffle and reduce steps that operate on a set of key-value pairs. Its cost is measured in terms of (a) N: the *number of rounds* of map-shuffle-reduce conducted in the process, (b) S_i: the *communication cost* of round i, as the sum of the sizes of input and output for all reducers, and (c) H_i: the *computational cost* of round i, as the sum of the time taken by each mapper and reducer in round i.

For Theorem 2(2), GRAPE uses PEval to perform the map phase of the first map-shuffle-reduce round, and two supersteps (IncEval) to simulate each later round, one for map and the other for reduce, via key-value messages (see Section 3.5). We provide a compilation function that given Map and Reduce functions, constructs (a) PEval as the Map function, (b) IncEval by invoking Map for odd supersteps and Reduce for even supersteps, and (c) Assemble by simply taking a union of partial results. By induction on the round N of MapReduce, one can verify that the transformed GRAPE process has running time $O(C)$, where C is the parallel running time of the MapReduce computation.

There exist more efficient compilation algorithms by combining multiple MapReduce tasks into a single GRAPE superstep. Moreover, GRAPE employs (bounded) IncEval to reduce MapReduce cost for iterative graph computations.

Parallel Random Access Machine. PRAM consists of a number of processors sharing memory, and any processor can access any memory cell in unit time. The computation is synchronous. In one unit time, each processor can read one memory location, execute a single operation and write into one memory location. PRAM is further classified for access policies of shared memory, *e.g.,* CREW PRAM indicates concurrent read and exclusive write (see [47] for details).

It is known that a CREW PRAM algorithm using t time with $O(P)$ total memory and $O(P)$ processors can be simulated by a MapReduce algorithm in $O(t)$ rounds using at most $O(P)$ reducers and memory [29]. By Theorem 2(2), each MapReduce algorithm in r rounds can be simulated by GRAPE in $2r$ supersteps. From these Theorem 2(3) follows.

5. GRAPH COMPUTATIONS IN GRAPE

We have seen how GRAPE parallelizes graph traversal SSSP (Section 3). We next show how GRAPE parallelizes existing sequential algorithms for a variety of graph computations. We take pattern matching, connectivity and collaborative filtering as examples (Sections 5.1–5.3, respectively).

5.1 Graph Pattern Matching

We start with graph pattern matching commonly used in, *e.g.,* social media marketing and knowledge base expansion.

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 $L_Q(u)$.

We study two semantics of graph pattern matching.

Graph simulation. 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 $(u, v) \in R$, referred to as a *match* of u; and
(b) for each pair $(u, v) \in R$, (a) $L_Q(u) = L(v)$, and (b) for each query edge (u, u') in E_q, there exists an edge (v, v') in graph G such that $(u', v') \in R$.

Graph pattern matching via graph simulation is as follows.

- Input: A directed graph G and a pattern Q.
- Output: The unique maximum relation $Q(G)$.

It is known that if G matches Q, then there exists a *unique maximum* relation [27], referred to as $Q(G)$. If G does not match Q, $Q(G)$ is the empty set. Moreover, $Q(G)$ can be computed in $O((|V_Q| + |E_Q|)(|V| + |E|))$ time [19, 27].

We show how GRAPE parallelizes graph simulation. Like SSSP, it adopts an edge-cut partition strategy.

(1) PEval. GRAPE takes the sequential simulation algorithm of [27] as PEval to compute $Q(F_i)$ in parallel. Its message preamble declares a Boolean status 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 $F_i.I$ 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 [27] 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 coordinator P_0.

At coordinator 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 **aggregateMsg**, taking the order false $\prec$ true. GRAPE identifies those variables that become false, deduces their destinations by referencing $G_{\mathcal{P}}$ and $\mathcal{F}.I = \mathcal{F}.O$, groups them into messages M_j, and sends M_j to P_j.

(2) IncEval is the sequential incremental graph simulation algorithm of [21] 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 [21] for details). The partial result is now the revised sim relation. At the end of the process, IncEval sends to coordinator P_0 updated values of those status variables in $C_i.\bar{x}$, as in PEval.

IncEval is *semi-bounded* [21]: 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 (sim at each F_i).

(4) The correctness is warranted by Theorem 1: the sequential algorithms [21, 27] (PEval and IncEval) are correct, and the "monotonic" updates to $C_i.\bar{x}$: $x_{(u,v)}$ is initially true for each border node v, and is changed at most once to false.

Subgraph isomorphism. We next parallelize subgraph isomorphism, under which a *match* of pattern Q in graph G is a subgraph of G that is isomorphic to Q. Graph pattern matching via subgraph isomorphism is to compute the set $Q(G)$ of all matches of Q in G. It is intractable: it is NP-complete to decide whether $Q(G)$ is nonempty.

GRAPE parallelizes VF2, the sequential algorithm of [16] for subgraph isomorphism. It adopts a default edge-cut graph partition strategy $\mathcal{P}$. It has *two* supersteps, one for PEval and the other for IncEval, outlined as follows.

(1) PEval identifies update parameter $C_i.\bar{x}$. It declares a status variable x_{id} with each node and edge, to store its id. It specifies the d_Q-neighbor $N_{d_Q}(v)$ of each node $v \in F_i.I$, where d_Q is the diameter of pattern Q, *i.e.,* the length of the shortest path between any two nodes in Q, and $N_d(v)$ is the subgraph of G induced by the nodes within d hops of v.

At P_0, $C_i.\bar{x}$ is identified for each fragment F_i (this can be done in parallel by workers as remarked in Section 3.2). Message M_i is composed and sent to P_i, including all nodes and edges in $C_i.\bar{x}$ that are from fragments F_j with $j \neq i$. The values of variables in $C_i.\bar{x}$ (the ids) will not be changed, and thus no partial order is defined on their values.

(2) IncEval is VF2. It computes $Q(F_i \oplus M_i)$ at each worker P_i in parallel, on fragment F_i extended with d_Q-neighbor of each node in $F_i.I$. IncEval sends no messages since the values of variables in $C_i.\bar{x}$ remain unchanged. As a result, IncEval is executed once, and hence two supersteps suffice.

(3) Assemble simply takes the union of all partial matches computed by IncEval from all workers.

(4) The correctness of the process 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'.

5.2 Graph Connectivity

We next study graph connectivity. We parallelize sequential algorithms for computing connected components (CC).

Consider an undirected graph G. A subgraph G_s of G is a *connected component* of G if (a) it is connected, *i.e.,* for any pair (v, v') of nodes in G_s, there exists a path between v to v', and (b) it is maximum, *i.e.,* adding any node to G_s makes the induced subgraph no longer connected.

- Input: An undirected graph $G = (V, E, L)$.
- Output: All connected components of G.

It is known that CC is in $O(|G|)$ time [10].

GRAPE partitions G by edge-cut. It picks a sequential CC algorithm as PEval. At each fragment F_i, PEval computes its local connected components and creates their ids. The component ids of the border nodes are exchanged with neighboring fragments. The (changed) ids are then used to incrementally update local components in each fragment by IncEval, which simulates a "merging" of two components whenever possible, until no more changes can be made.

(1) PEval declares an integer status variable v.cid for each node v in fragment F_i, initialized as its node id.

PEval uses a standard sequential traversal (*e.g.,* DFS) to compute the local connected components of F_i and determines v.cid for each $v \in F_i$. For each local component C, (a) PEval creates a "root" node v_c carrying the minimum node id in C as v_c.cid, and (b) links all the nodes in C to v_c, and sets their cid as v_c.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 coordinator P_0.

At P_0, GRAPE maintains v.cid for each all $v \in \mathcal{F}.I$. It updates v.cid by taking the smallest cid if multiple cids are received, by taking min as **aggregateMsg** in the message segment of PEval. It groups the nodes with updated cids into messages M_j, and sends M_j to P_j by referencing $G_{\mathcal{P}}$.

(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.cid with updated (smaller) values. For each v in M_i, IncEval (a) finds the root v_c of v, and (b) for v_c and all the nodes linked to it, directly changes their cids to v.cid.

The incremental computation of 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 link from the root nodes, where AFF consists of only those nodes with their cid *changed*. Hence, it avoids redundant local traversal, and makes the complexity of IncEval independent of $|F_i|$.

(3) Assemble merges all the nodes having the same cids in a bucket as a single connected component, and returns all the connected components as a set of buckets.

(4) Correctness. The process terminates as the cids of the nodes are monotonically decreasing, until no changes can be made. Moreover, it correctly merges two local connected components by propagating the smaller component id.

5.3 Collaborative Filtering

As an example of machine learning, we consider collaborative filtering (CF) [33], a method commonly used for inferring user-product rates in social recommendation. It takes as input a bipartite graph G that includes users U and products P, and a set of weighted edges $E \subseteq U \times P$. (1) Each user $u \in U$ (resp. product $p \in P$) carries (unknown) latent factor vector $u.f$ (resp. $p.f$). (2) Each edge $e = (u, p)$ in E carries a weight $r(e)$, estimated as $u.f^T * p.f$ (possibly $\emptyset$ *i.e.,* "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. The CF problem is as follows.

- Input: Directed bipartite graph G, training set E_T.
- Output: The missing factor vectors $u.f$ and $p.f$ that minimizes an error function $\epsilon(f, E_T)$, estimated as $\mathsf{min} \sum_{((u,p)\in E_T)} (r(u,p) - u.f^T p.f) + \lambda(\|u.f\|^2 + \|p.f\|^2)$.

That is, CF predicts all the unknown ratings by learning the factor vectors that "best fit" E_T. A common practice

to approach CF is to use stochastic gradient descent (SGD) algorithm [33], 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 towards minimizing $\epsilon(f, E_T)$.

GRAPE parallelizes CF by edge-cut partitioning E_T (as a bipartite graph). It adopts SGD [33] as PEval and an incremental algorithm ISGD [48] as IncEval, using coordinator P_0 to synchronize the shared factor vectors $u.f$ and $p.f$.

(1) PEval. It declares a status variable $v.x = (v.f, t)$ for each node v, where $v.f$ is the factor vector of v (initially $\emptyset$), and t bookkeeps a timestamp at which $v.f$ is lastly updated. The candidate set is $C_i = F_i.O$. PEval is essentially the sequential SGD algorithm of [33]. It processes a "mini-batch" of training examples independently of others, to compute the prediction error $\epsilon(u,p)$, and update local factor vectors f in the opposite direction of the gradient as:

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

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

At the end of its process, PEval sends messages M_i that contains updated $v.x$ for $v \in C_i$ to coordinator P_0.

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

(2) IncEval is algorithm ISGD of [48]. Upon receiving 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 the 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 [33,48] is specified either as a predetermined maximum number of supersteps (as in 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.

6. IMPLEMENTATION OF GRAPE

We next outline an implementation of GRAPE.

Architecture overview. GRAPE adopts a four-tier architecture depicted in Fig. 5, described as follows.

(1) Its top layer is a user interface. As shown in Fig. 1, GRAPE supports interactions with (a) developers who specify and register sequential PEval, IncEval and Assemble as a PIE program for a class $\mathcal{Q}$ of graph queries (the plug panel); and (b) end users who plug-in PIE programs from API library, pick a graph G, enter queries $Q \in \mathcal{Q}$, and "play" (the play panel). GRAPE parallelizes the PIE program, computes $Q(G)$ and displays $Q(G)$ in result and analytics consoles.

(2) At the core of the system is a parallel query engine. It manages sequential algorithms registered in GRAPE API, makes parallel evaluation plans for PIE programs, and executes the plans for query answering (see Section 3.1). It also enforces consistency control and fault tolerance (see below).

(3) Underlying the query engine are (a) an *MPI Controller* (message passing interface) for communications between coordinator and workers, (b) an *Index Manager* for loading indices, (c) a *Partition Manager* to partition graphs, and (d) a *Load Balancer* to balance workload (see below).

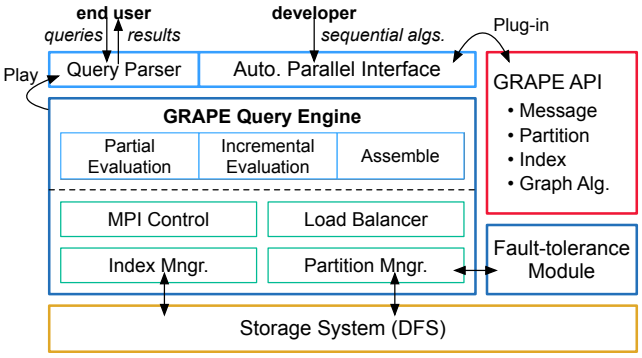


Figure 5: GRAPE **Architecture**

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

Message passing. The MPI Controller of GRAPE makes use of a standard MPI for parallel and distributed programs. It currently adopts MPICH [6], which is also the basis of other systems such as GraphLab [34] and Blogel [50]. It generates messages and coordinates messages in synchronization steps using standard MPI primitives. It supports both designated messages and key-value pairs (see Section 3).

Graph partition. The Graph Partitioner supports a variety of built-in partition algorithms. Users may pick (a) METIS, a fast heuristic algorithm for sparse graphs [30], (b) vertex cut and edge cut partitions [24] for graphs with small vertex cut-set and edge cut-set, respectively, (c) 1-D and 2-D partitions [12], which distribute vertex and adjacent matrix to the workers, respectively, emphasizing on maximizing the parallelism of graph traversal, and (d) a fast streaming-style partition strategy [43] that assigns edges to high degree nodes to reduce cross edges. New data partition strategies can also be plugged into GRAPE.

Graph-level optimization. In contrast to prior graph systems, GRAPE supports data-partitioned parallelism by parallelizing the runs of sequential algorithms. Hence all optimization strategies developed for sequential (batch and incremental) algorithms can be readily plugged into GRAPE, to speed up PEval and IncEval over graph fragments. As examples, below we outline some optimization strategies.

(1) Indexing. Any indexing structure effective for sequential algorithm can be computed offline and directly used to optimize PEval, IncEval and Assemble, without recasting. GRAPE supports indices including (1) 2-hop index [15] for reachability queries; and (2) neighborhood-index [31] for candidate filtering in graph pattern matching. Moreover, new indices can be "plugged" into GRAPE API library.

(2) Compression. GRAPE adopts *query preserving compression* [20] at the fragment level. Given a query class $\mathcal{Q}$ and a fragment F_i, each worker P_i computes a smaller F_i^c offline via a compression algorithm, such that for any query Q in $\mathcal{Q}$, $Q(F_i)$ can be computed from F_i^c without decompressing F_i^c, *regardless* of what sequential PEval and IncEval are used. As shown in [20], this compression scheme is effective for graph pattern matching and graph traversal, among others.

(3) Dynamic grouping. GRAPE dynamically group a set of border nodes by adding a "dummy" node, and sends messages from the dummy nodes in batches, instead of one by one. This effectively reduces the amount of message communication in each synchronization step.

Load balancing. GRAPE groups computation tasks into work units, and estimates the cost at each virtual worker P_i in terms of the fragment size $|F_i|$ at P_i, the number of border nodes in F_i, and the complexity of computation $\mathcal{Q}$. Its Load Balancer computes an assignment of the work units to physical workers, to minimize both computational cost and communication cost (recall from Section 3 that GRAPE employs m virtual workers and n physical workers, and $m > n$). The bi-criteria objective makes it easy to deal with skewed graphs, when a small fraction of nodes are adjacent to a large fraction of the edges in G, as found in social graphs.

To the best of our knowledge, these optimization strategies are not supported by the state-of-the-art vertex-centric and block-centric systems. Indexing and query-preserving compression for sequential algorithms do not carry over to vertex programs, and block-centric programming essentially treats blocks as vertices rather than graphs. Moreover, dynamic grouping does not help vertex-level synchronization.

Fault tolerance. GRAPE employs an arbitrator mechanism to recover from both worker failures and coordinator failures (*a.k.a.* single-point failures). It reserves a worker P_a as arbitrator, and a worker S_c' as a standby coordinator. It keeps sending heart-beat signals to all workers and the coordinator. In case of failure, (a) if a worker fails to respond, the arbitrator transfers its computation tasks to another worker; and (b) if the coordinator fails, it activates the standby coordinator S_c' to continue computation.

Consistency. Multiple workers may update copies of the same status variable. To cope with this, (a) GRAPE allows users to specify a conflict resolution policy as function **aggregateMsg** in PEval (Section 3.2), *e.g.,* min for SSSP and CC (Section 5), based on a partial order on the domain of status variables, *e.g.,* linear order on integers. Based on the policy, inconsistencies are resolved in each synchronization step of PEval and IncEval processes. Moreover, Theorem 1 guarantees the consistency when the policy satisfies the monotonic condition. (b) GRAPE also supports default exception handlers when users opt not to specify **aggregateMsg**. In addition, GRAPE allows users to specify generic consistency control strategies and register them in GRAPE API library.

We are also implementing a lightweight transaction controller, to support not only queries but also updates such as insertions and deletions of nodes and edges. When the load is light, it adopts non-destructive updates of functional databases [45]. Otherwise, it switches to multi-version concurrency control [11] that keeps track of timestamps and versions, as also adopted by existing distributed systems.

7. EXPERIMENTAL STUDY

We next empirically evaluate GRAPE, for its (1) efficiency, (2) effectiveness of incremental steps, and (3) compatibility with optimization techniques developed for sequential algorithms, using real-life graphs. We also report in Appendix its (4) communication costs, (5) scalability with larger synthetic graphs, and (6) ease of programming. To focus on the main idea, we compared GRAPE with prior graph systems by plugging existing sequential algorithms into a preliminary implementation of GRAPE, without optimization.

Experimental setting. We start with graphs and queries.

Datasets. We used three real-life graphs of different types, including (1) liveJournal [7], a social network with 4.8 million entities and 68 million relationships, with 100 labels and 18293 connected components; (2) DBpedia [2], a knowledge base with 28 million entities of 200 types and 33.4 million edges of 160 types; and (3) traffic [8], a US road network with 23 million nodes (locations) and 58 million edges.

To evaluate collaborative filtering (CF), we used another real-life dataset movieLens [5], which has 10 million movie ratings (as weighted edges) between a set of 71567 users and 10681 movies; these make a bipartite graph G for CF.

We also used larger synthetic graphs (see Appendix B).

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 (see Section 5).

Algorithms. We implemented the core functions PEval, IncEval and Assemble given in Sections 3 and 5 for these query classes, registered in the API library of GRAPE. We used METIS [30] as the default graph partition strategy. We adopted basic sequential algorithms, and only used optimized Sim to demonstrate how GRAPE inherits optimization strategies developed for sequential algorithms (Exp-3).

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; the code is available at [4] for interested reader. As an example, we provide the code for SSSP in Appendix. As GraphLab supports both synchronized and asynchronized models [34], 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.

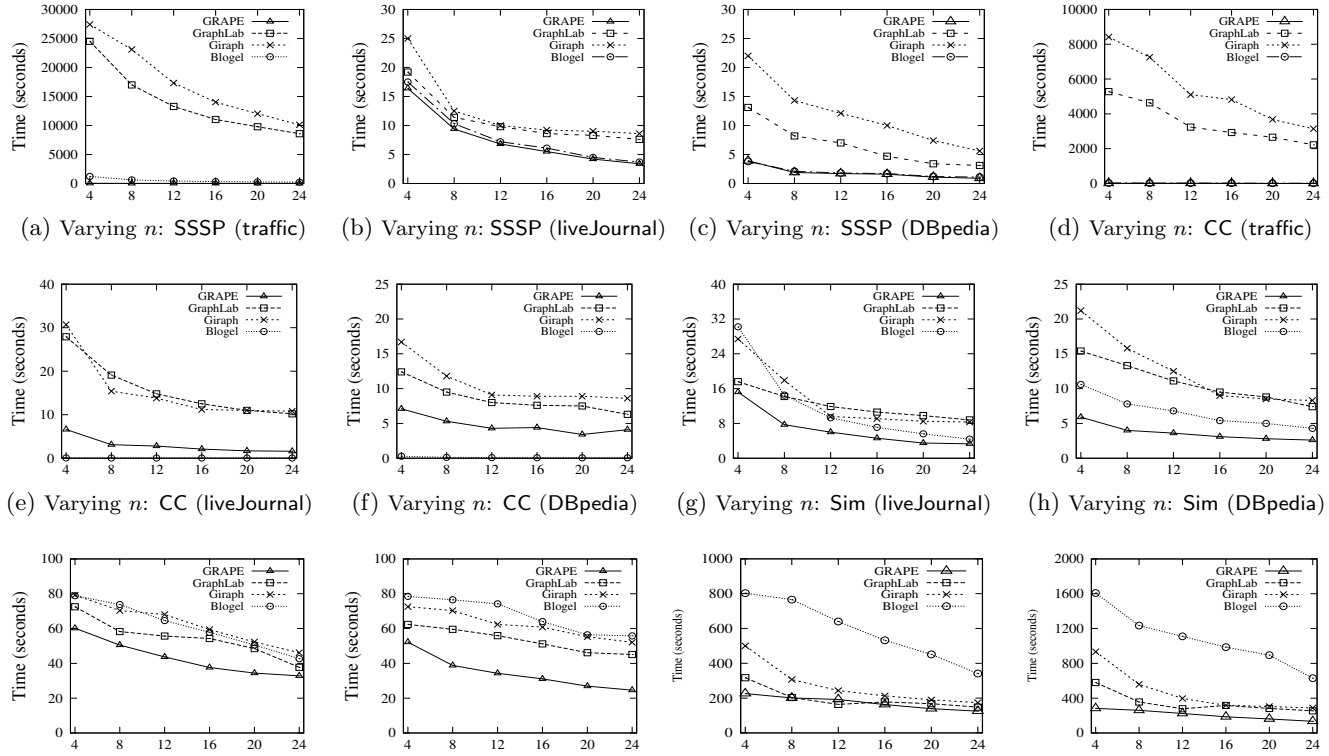
We deployed the systems on Aliyun ECS n2.large instances [1], each powered by an Intel Xeon processor with 2.5GHz and 16G memory. We used up to 24 instances. We used ECS since its average inter-connection speed is close to real-life large-scale distributed systems. Each experiment was run 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluated the efficiency of GRAPE over real-life graphs by varying the number n of processors used, from 4 to 24. We compared its performance with Giraph, GraphLab and Blogel. For SSSP and CC, we experimented with all three real-life datasets. For Sim and SubIso, we evaluated the queries over liveJournal and DBpedia, since these queries are meaningful on labeled graphs only, while traffic does not carry labels.

(1) SSSP. Figures 6(a)-6(c) report the performance of the systems for SSSP over traffic, liveJournal and DBpedia, respectively. From the results we can see the following.

(a) GRAPE outperforms Giraph, GraphLab and Blogel by 964, 818 and 22 times, respectively, over traffic with 24 processors (Fig 6(a)). In the same setting, it is 2.5, 2.2 and 1.1 times



(a) Varying n: SSSP (traffic) (b) Varying n: SSSP (liveJournal) (c) Varying n: SSSP (DBpedia) (d) Varying n: CC (traffic)

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

(i) Varying n: SubIso (liveJournal) (j) Varying n: SubIso (DBpedia) (k) Varying n: CF ($|E_T|$=90%$|E|$) (l) Varying n: CF ($|E_T|$=50%$|E|$)

Figure 6: Efficiency of GRAPE

faster over liveJournal (Fig. 6(b)), and 6.2, 3.4 and 1.2 times faster over DBpedia (Fig. 6(c)). By simply parallelizing sequential algorithms without further optimization, GRAPE is comparable to the state-of-the-art systems in response time.

Note that the improvement of GRAPE over Giraph and GraphLab on traffic is much larger than on liveJournal and DBpedia. This is because vertex-centric algorithms take more supersteps to converge on graphs with large diameters, *e.g.,* traffic. Giraph takes 10752 supersteps over traffic, while 18 over liveJournal; similarly for GraphLab. In contrast, GRAPE is not vertex-centric and is more robust; it takes 18 supersteps on traffic and 10 on liveJournal.

(b) All systems take less time when n increases, and GRAPE scales well with n. The speedup of GRAPE compared to Giraph and GraphLab becomes larger when more processors are used; *e.g.,* GRAPE is 818 times faster than Giraph with 4 processors, and is 964 times faster with 24 processors. On average, GRAPE is 4 times faster for n from 4 to 24, while it is 3 times for Giraph, 3.2 times for GraphLab and 5 times for Blogel. These verify the parallel scalability of GRAPE.

(c) GRAPE ships on average $9 * 10^{-6}$%, 6.4% and 0.05% of the data shipped by Giraph, $9 * 10^{-6}$%, 6.4% and 0.05% of GraphLab, and $3.5 * 10^{-4}$%, 24% and 0.94% of Blogel, over traffic, liveJournal and DBpedia, respectively (Figures 8(a)-8(c); see Appendix B for details). In particular, GRAPE significantly reduces supersteps. It takes on average 12 supersteps, while Giraph, GraphLab and Blogel take 10752, 10752 and 1673 supersteps, respectively. This is because GRAPE runs sequential algorithms over fragmented graphs, and triggers cross-fragment communication only when necessary; moreover, IncEval ships only *changes* to status variable, which are updated monotonically (Theorem 1). In contrast, Giraph, GraphLab and Blogel pass vertex-vertex (vertex-block) messages as required by recasted programs.

(2) CC. Figures 6(d)-6(e) report the performance for CC detection, and tell us the following. (a) Both GRAPE and Blogel substantially outperform Giraph and GraphLab. For instance, when $n = 24$, GRAPE is on average 4.4 and 4.0 times faster than Giraph and GraphLab, respectively. (b) Blogel is faster than GRAPE, *e.g.,* 0.05s vs. 1.6s over liveJournal when $n = 24$. This is because Blogel embeds the computation of CCs in its graph partition phase as precomputation, while the partitioning cost (on average 16.2 seconds) is *not* included in the response time of Blogel. In other words, without precomputation, the performance of GRAPE is already comparable to the near "optimal" case reported by Blogel that is run over graphs already partitioned into connected components. (c) GRAPE incurs only 4.8% of communication cost of both Giraph and GraphLab on average, and is comparable to that of the near "optimal" case of Blogel (see Figures 8(d)-8(f), Appendix B).

(3) Sim. Fixing $|Q| = (8, 15)$, *i.e.,* patterns Q with 8 nodes and 15 edges, we evaluated matching via graph simulation over liveJournal and DBpedia. As shown in Figures 6(g)-6(h), (a) GRAPE consistently outperforms Giraph, GraphLab and Blogel over all queries. It is 2.5, 2.7 and 1.3 times faster over liveJournal, and 3.2, 2.8 and 1.7 times faster over DBpedia on average, respectively, when $n = 24$. (b) GRAPE scales better with the number n of processors than Giraph and GraphLab, and is comparable to Blogel in parallel scalability. (c) GRAPE ships 2.2%, 2.2% and 2.3% (liveJournal), and 0.45%, 0.45% and 0.9% (DBpedia) of the data shipped by Giraph, GraphLab and Blogel on average, respectively, when $n = 24$ (Figures 8(g)-8(h), Appendix B). In partic-

ular, GRAPE takes at most 6 supersteps to terminate, while Giraph, GraphLab and Blogel take 7, 8 and 10 supersteps, respectively. This again empirically validates Theorem 1, which allows us to monotonically update status variables.

(4) SubIso. Fixing $|Q|=(6,10)$, we evaluated subgraph isomorphism. As shown in Figures 6(i)-6(j) over liveJournal and DBpedia, respectively, (a) GRAPE is on average 1.86, 1.49 and 1.98 times faster than Giraph, GraphLab and Blogel, respectively, when $n = 24$. (b) GRAPE does well over all queries tested. It takes 2 supersteps and 38.9 seconds on average, while Giraph, GraphLab and Blogel take 62.4, 54.3 and 64.5 seconds and 4, 4 and 6 supersteps, respectively. (c) GRAPE scales well with the number n of processors. (d) GRAPE incurs on average 5.9%, 5.9% and 8.4% of the communication cost of Giraph, GraphLab and Blogel, respectively, when $n = 24$ (Figures 8(i)-8(j), Appendix B).

*(5) Collaborative filtering (*CF*).* For CF, we used movieLens [5] with two training sets, compared with the built-in SGD-based CF in Giraph and GraphLab, and CF implemented for Blogel. We calibrated the termination condition of all the systems as the convergence point when the root-mean-square error of predicted ratings is less than a threshold.

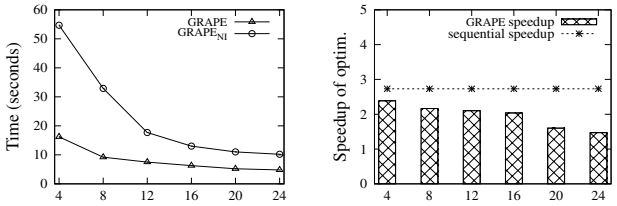
We first tested training set $|E_T| = 90\%\ |E|$. Note that CF favors "vertex-centric" programming since each user or product node only needs to exchanges data with their neighbors, as indicated by that GraphLab and Giraph outperform Blogel. Nonetheless, as shown in Fig. 6(k), GRAPE is on average 1.6, 1.1 and 3.4 times faster than Giraph, GraphLab and Blogel, respectively, when the number n of processors varies from 4 to 24. It scales well with n. In addition, it ships on average 9.2%, 9.2% and 10.2% of data shipped by Giraph, GraphLab and Blogel, respectively (Fig. 8(k), Appendix B).

We also tested smaller training set ($|E_T| = 50\%\ |E|$). Figure 6(l) shows that GRAPE outperforms Blogel and Giraph, and is comparable with GraphLab. It ships at most 11.6% of data shipped by Giraph, GraphLab and Blogel (Fig. 8(l)).

More results on larger synthetic graphs are reported in Fig. 9 (Appendix B), which are consistent with their counterparts on real-life graphs reported in Fig. 6.

Exp-2: Incremental computation. We evaluated the effectiveness of incremental IncEval. We implemented a batch version of GRAPE for Sim queries, denoted as GRAPE_{NI}, which uses PEval to perform iterative computations and handle the messages, instead of IncEval. It mimics the case when no incremental computation is used. As shown in Fig. 7(a) over liveJournal, (1) GRAPE outperforms GRAPE_{NI} by 2.1 times with 24 processors; and (2) the gap is larger when less workers are employed, *e.g.,* 3.4 times when 4 processors are used. This is because the less workers are used, the larger fragments reside at each worker, and as a consequence, heavier computation costs are incurred at each superstep. This verifies that incremental steps effectively reduces redundant local computations in iterative graph computations. The results on DBpedia are consistent and are not shown.

Exp-3. Compatibility. We also evaluated the compatibility of optimization strategies developed for sequential graph algorithms with GRAPE parallelization. For a query class $\mathcal{Q}$, a sequential algorithm $\mathcal{A}$ and its optimized version $\mathcal{A}^*$ for $\mathcal{Q}$, denote the speedup of the optimization as $\frac{T(\mathcal{A})}{T(\mathcal{A}^*)}$. Denote the running time of GRAPE parallelization of $\mathcal{A}$ (resp. $\mathcal{A}^*$) as



(a) The impact of IncEval (b) Optimization on GRAPE

Figure 7: Incremental steps and optimization

$T_p(\mathcal{A})$ (resp. $T_p(\mathcal{A}^*)$) for a given number n of workers. Ideally, $\frac{T(\mathcal{A})}{T(\mathcal{A}^*)}$ should be close to $\frac{T_p(\mathcal{A})}{T_p(\mathcal{A}^*)}$, *i.e.,* GRAPE *preserves* the speedup from the optimization. That is, the impact of the optimization is not "dampened out" by parallelization overhead such as synchronization and message passing.

We make a case for graph simulation. We evaluated two sequential algorithms, one from [27], and the other is an optimized version that employs indices to reduce candidates [19]. Using Sim queries over liveJournal, we found that the average speedup of sequential algorithms is 2.7. Varying n from 4 to 24, we report the speedup of the parallelized algorithms of GRAPE in Fig. 7(b). The result on DBpedia are consistent (not shown). The results suggest that the speedup is close to its sequential counterpart. Such optimization cannot be easily encoded in vertex programs of Giraph and GraphLab and the V-mode and B-mode programs of Blogel.

Summary. We find the following. (1) By plugging in sequential algorithms, GRAPE performs comparably to state-of-the-art systems. Over real-life graphs and using from 4 to 24 processors, GRAPE is on average 323, 274 and 7.9 times faster than Giraph, GraphLab and Blogel for SSSP, 2.7, 2.6 and 1.7 for Sim, 1.7, 1.4 and 1.7 for SubIso, and 1.9, 1.4 and 3.8 for CF, respectively. For CC, it is 3.9 and 3.8 times faster than Giraph and GraphLab, respectively, and is comparable to the "optimal" case of Blogel. The results on synthetic graphs are consistent (Appendix B). (2) Better still, GRAPE ships on average 5.6%, 5.6% and 10% of the data shipped by Giraph, GraphLab and Blogel for SSSP, 1.3%, 1.3% and 1.6% for Sim, 4.7%, 4.7% and 6.5% for SubIso, and 8.1%, 8.1% and 8.7% for CF, respectively, in the same setting. For CC, it incurs 7.3% and 7.3% of data shipment of Giraph and GraphLab, and is comparable with "optimized" Blogel (Appendix B). (3) Incremental steps effectively reduce iterative recomputation. For Sim, it improves the response time by 2.6 times on average. (4) GRAPE inherits the benefit of optimized sequential algorithms. For Sim, it is on average 2 times faster by using the algorithm of [19] instead of [27].

8. CONCLUSION

We have proposed an approach to parallelizing sequential graph algorithms. For a class of graph queries, users can plug in existing sequential algorithms with minor changes. GRAPE parallelization guarantees to terminate with correct answers under a monotonic condition if the sequential algorithms are correct. Moreover, graph algorithms for existing parallel graph systems can be migrated to GRAPE, without incurring extra cost. We have verified that GRAPE achieves comparable performance to the state-of-the-art graph systems for various query classes, and that (bounded) IncEval reduces the cost of iterative graph computations.

An open-source GRAPE will be available at [4]. An asynchronous version of GRAPE is also under development.

Acknowledgments. Fan, Xu, Yu, Cao and Tian are supported in part by ERC 652976, NSFC 61421003, 973 Program 2014CB340302, EPSRC EP/M025268/1, Shenzhen Peacock Program 1105100030834361, Guangdong Innovative Research Team Program 2011D005, the Foundation for Innovative Research Groups of NSFC, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Wu is supported by NSF BIGDATA 1633629 and Google Faculty Research Award. Cao is also supported by NSFC 61602023. Jiang is supported by HKRGC GRF HKBU12232716.

9. REFERENCES

[1] Aliyun. *https://intl.aliyun.com*.
[2] DBpedia. *http://wiki.dbpedia.org/Datasets*.
[3] Giraph. *http://giraph.apache.org/*.
[4] GRAPE. *http://grapedb.io/*.
[5] Movielens. *http://grouplens.org/datasets/movielens/*.
[6] MPICH. *https://www.mpich.org/*.
[7] Snap. *http://snap.stanford.edu/data/index.html*.
[8] Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml*.
[9] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, CMU, 2005.
[10] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2008.
[11] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221.
[12] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *SC*, 2013.
[13] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *SIGKDD*, pages 1456–1465, 2014.
[14] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *VLDB*, 2006.
[15] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SICOMP*, 32(5):1338–1355, 2003.
[16] 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.
[17] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.
[18] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.
[19] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractability to polynomial time. In *PVLDB*, 2010.
[20] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, 2012.
[21] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.
[22] W. Fan, X. Wang, and Y. Wu. Distributed graph simulation: Impossibility and possibility. *PVLDB*, 7(12), 2014.
[23] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.
[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX*, 2012.
[25] 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.
[26] T. J. Harris. A survey of PRAM simulation techniques. *ACM Comput. Surv.*, 26(2):187–206, 1994.
[27] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[28] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3), 1996.
[29] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *SODA*, 2010.
[30] G. Karypis and V. Kumar. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.
[31] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: Fast graph search with label similarity. *PVLDB*, 6(3), 2013.
[32] M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.
[33] Y. Koren, R. Bell, C. Volinsky, et al. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.
[34] 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.
[35] 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.
[36] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *ASPLOS*, 2014.
[37] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The tao of parallelism in algorithms. In *ACM Sigplan Notices*, volume 46, pages 12–25, 2011.
[38] C. Radoi, S. J. Fink, R. M. Rabbah, and M. Sridharan. Translating imperative code to MapReduce. In *OOPSLA*, 2014.
[39] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
[40] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.
[41] V. Raychev, M. Musuvathi, and T. Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *SOSP*, 2015.
[42] S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, 2013.
[43] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230, 2012.
[44] 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.
[45] P. Trinder. *A Functional Database*. PhD thesis, University of Oxford, 1989.
[46] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
[47] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Vol A*. 1990.
[48] J. Vinagre, A. M. Jorge, and J. Gama. Fast incremental matrix factorization for recommendation with positive-only feedback. In *International Conference on User Modeling, Adaptation, and Personalization*, 2014.
[49] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.
[50] 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.
[51] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 7(14):1821–1832, 2014.
[52] Y. Zhou, L. Liu, K. Lee, C. Pu, and Q. Zhang. Fast iterative graph computation with resource aware graph parallel abstractions. In *HPDC*, 2015.

Appendix A: Proofs

Proof of Theorem 1

By the correctness of Assemble, PEval and IncEval, we only need to show the following: for any query Q and graph G,

(1) there exists a natural number $r_{(Q,G)}$ for Q and G such that GRAPE terminates at superstep $r_{(Q,G)}$, with deterministic values $\bar{x}_i^{r_{(Q,G)}}$ for all update parameters in all fragments F_i of G (for $i \in [1,m]$); and

(2) IncEval computes partial answers $Q(F_i[\bar{x}_i^{r_{(Q,G)}}])$ on all fragments $F_i (i \in [1,m])$ of G.

Intuitively, (1) ensures that given Q and G, GRAPE always terminates in the same state, and (2) guarantees that partial answers $Q(F_i[\bar{x}_i^{r_{(Q,G)}}])$ are correctly computed for all fragments F_i $(i \in [1,m])$ of G. If these hold, GRAPE is guaranteed to return $Q(G)$ by the correctness of Assemble.

(1) We first show that GRAPE terminates. Assume by contradiction that there exist Q and G such that GRAPE does not terminate. Consider the values of update parameters in the fragments of G during the run. Since at least one update parameter has to be updated in a superstep of incremental computation (except the last step), and the total number of distinct values to update parameters is bounded by Q and G by the monotonic condition (a) given in Section 4.1. Hence there must exist supersteps p and q such that for each $i \in [1,m]$, $\bar{x}_i^p = \bar{x}_i^q$, *i.e.,* the values to all the parameters changed at supersteps p and q are the same. This contradicts the monotonic condition (b) that requires IncEval to update parameters following a partial order on their values. Thus for all Q and G, GRAPE must terminate.

To verify that the values to $C_i.\bar{x}$ when GRAPE terminates are deterministic for Q and G, we show the following: the values to $C_i.\bar{x}$ are updated deterministically at each superstep r in the run of GRAPE, by induction on r. (a) When $r = 1$, *i.e.,* in the first superstep by PEval, the parameters are initialized deterministically by the definition of PEval. (b) Assume that when $r \leq k$, the parameters in fragments of G for Q are changed deterministically at step r. Consider step $r = k+1$. Since $\bar{x}_i^k$'s $(i \in [1,m])$ are deterministic, IncEval generates M_i to each F_i deterministically, *i.e.,* $\bar{x}_i^{k+1} = \bar{x}_i^k \oplus M_i$ are updated deterministically. That is, values to $\bar{x}_i^{k+1}$ are also deterministic, independent of the order of fragments on which IncEval terminates at each superstep. Therefore, GRAPE always terminates for Q and G with the same final state for the update parameters.

(2) We prove that for any Q and G, at any superstep r of the run of GRAPE for Q and G with PEval, IncEval and Assemble, partial answers $Q(F_i[\bar{x}_i^r])$ $(i \in [1,m])$ are computed on all fragments F_i of G. We show this by induction on r.

(a) When $r = 1$. By the correctness of PEval, partial answers $Q(F_i[\bar{x}_i^1])$ are computed by PEval on fragments F_i of G.

(b) Assume that when $r = k$, GRAPE computes partial answers $Q(F_i[\bar{x}_i^k])$ on fragments F_i of G. Consider $r = k+1$. By the correctness of IncEval, GRAPE also correctly computes $Q(F_i[\bar{x}_i^k \oplus M]) = Q(F_i[\bar{x}_i^{k+1}])$ on each fragment F_i of G. Therefore, GRAPE computes partial answers on fragments of G at each superstep in the run for Q and G. □

Proof of Theorem 2

(1) Since BSP and GRAPE have the same amount of physical workers, each worker of BSP is simulated by a worker in GRAPE. Initially the graph is distributed in the same way as that in BSP algorithm $\mathcal{A}$. PEval is defined to do the same as the local computation during the first superstep of $\mathcal{A}$, and it generates messages that are identical to the ones in $\mathcal{A}$. From the second superstep, IncEval conducts the actions of each worker when executing BSP algorithm. Message routing and synchronization control adopt the same strategy as in $\mathcal{A}$. Obviously the computation on each worker in GRAPE is the same as its counterpart in BSP, and all messages sent or received by each pair of workers within each superstep are also identical, which lead to an optimal simulation.

(2) We use two supersteps in GRAPE to simulate one map-shuffle-reduce round of a MapReduce algorithm, including a map phase and a reduce phase, in the key-value message mode (see Section 3.5) for messages. More specifically, for a MapReduce algorithm $\mathcal{A}$ that has R rounds, we implement each round $r \in [1,R]$ of $\mathcal{A}$ in GRAPE as follows.

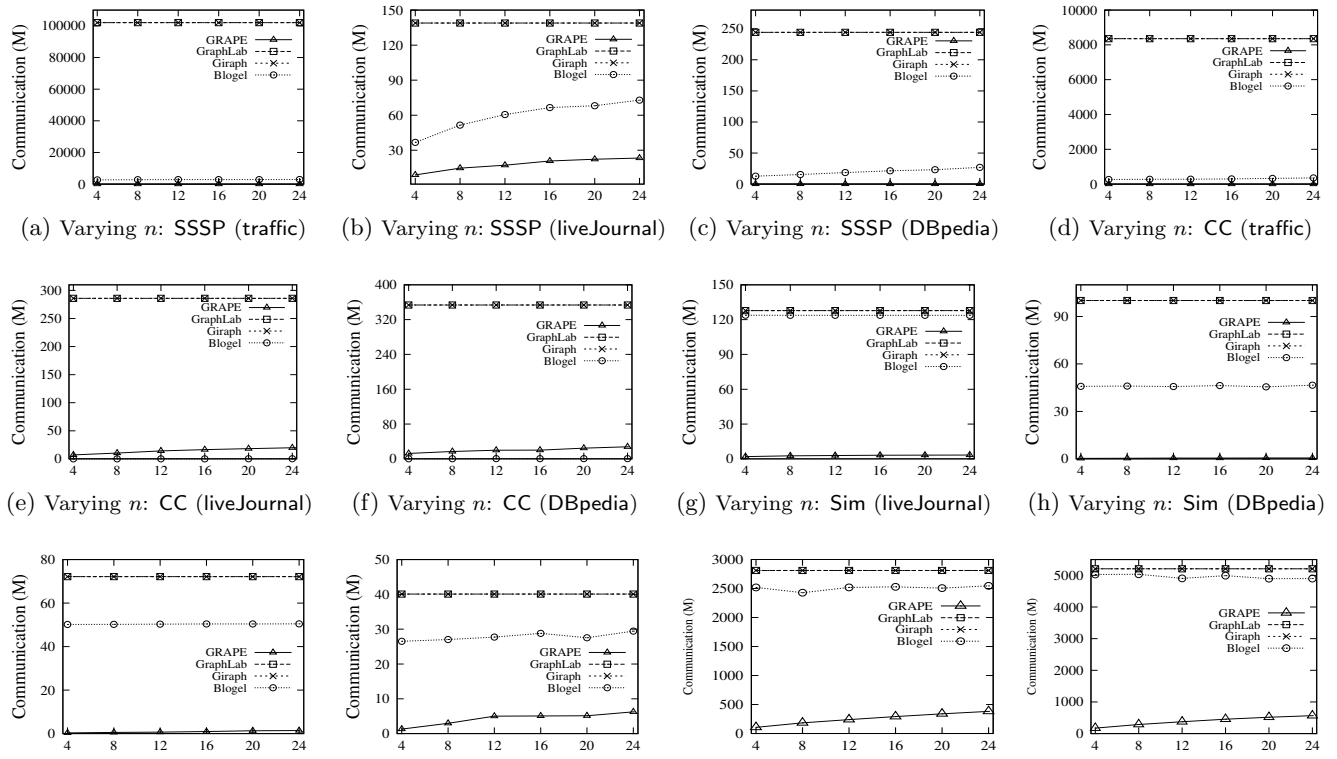
(a) Round $r = 1$: Initially input data is distributed among the worker by using the same strategy as in $\mathcal{A}$, such that each worker is assigned the same data as that of the mapper it simulates. We define PEval to be the same as the mapping function μ in round 1, *i.e.,* it performs the same computation as specified in μ and generates an intermediate multiset of key-value pairs. Moreover, the key-value pairs are treated as messages and sent to the coordinator P_0. Then P_0 groups all the messages ($\langle key; value \rangle$ pairs) with the same key and sends them to a worker that simulates the corresponding reducer dealing with key in $\mathcal{A}$. This process simulates one shuffle step of $\mathcal{A}$. After that, each worker that receives a message (list) $L_k = \langle k; v_{i_j} \ldots \rangle$ simulates a reducer, *i.e.,* we let function IncEval in this superstep do the same as the reducer function ρ in round 1. Note that IncEval uses the messages received only, ignoring the local data. The outputs of IncEval are also treated as messages and delivered to P_0. Upon receiving these, P_0 routes them based on the distribution of key-value pairs to mappers in the next round of $\mathcal{A}$, so that each worker gets the same key-value pairs as that of the mapper it will simulate in the next round.

(b) Round $r > 1$: The simulations for latter rounds are similar to those of the first round, except that PEval is no longer used. More specifically, the action of mapping function μ in round r is simulated by IncEval instead of PEval as in case (a). Hence, function IncEval is carefully designed to model the computation of the functions μ and ρ in different rounds of $\mathcal{A}$. IncEval operates on newly received messages alone, to simulate MapReduce. When $\mathcal{A}$ terminates, P_0 stops routing the messages produced in the last superstep and returns result in the same way as $\mathcal{A}$, possibly using Assemble.

It is easy to verify that the computational and communication cost of the GRAPE algorithm is the same as $\mathcal{A}$. Indeed, every worker simulates a mapper/reducer and conducts the same computation, and all the messages generated are identical to the key-value pairs transmitted in the shuffle network of $\mathcal{A}$. Thus, this makes an optimal simulation.

(3) A proof has been given in Section 4.2.

Taken together, GRAPE can easily switch to different modes, and does not imply degradation of computational power. We remark that Theorem 1 still holds in these settings as long as their messages can be such organized to satisfy the monotonic condition described in Section 3.1. □



(a) Varying n: SSSP (traffic) (b) Varying n: SSSP (liveJournal) (c) Varying n: SSSP (DBpedia) (d) Varying n: CC (traffic)

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

(i) Varying n: SubIso (liveJournal) (j) Varying n: SubIso (DBpedia) (k) Varying n: CF (E_T 90%) (l) Varying n: CF (E_T 50%)

Figure 8: Communication costs

Appendix B: More on Experimental Study

We report more experimental results, continuing Section 7.

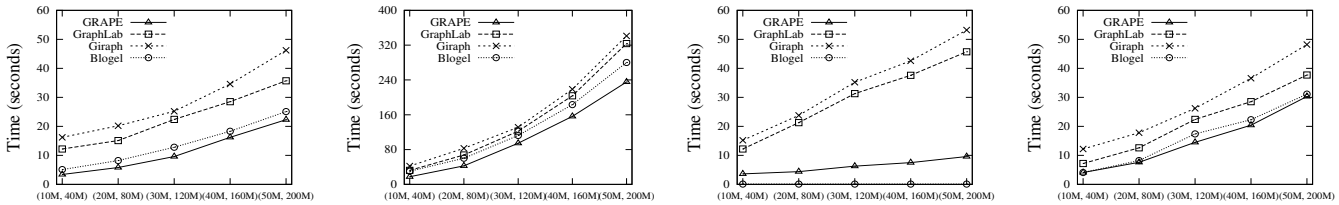
Exp-4: Communication cost. In the same setting of Exp-1, Figure 8 reports the communication costs of the systems. We observe that Giraph and GraphLab ship roughly the same amount of data. GRAPE incurs much less communication cost than Giraph and GraphLab. It ships on average 2.5%, 7.4%, 15%, $5 * 10^{-5}$% and 11.8% of the data shipped by Giraph and GraphLab for Sim, CC, SubIso, SSSP and CF, respectively, with 24 processors. While it ships more data than Blogel for CC for reasons to be given shortly, it only ships 2.6%, 21%, $1.7 * 10^{-3}$% and 12.7% of the data shipped by Blogel for Sim, SubIso, SSSP and CF, respectively. Moreover, the communication cost of GRAPE is insensitive to the increase of n, *i.e.,* when more processors are used, the communication cost does not change substantially.

(1) SSSP. Figures 8(a)-8(c) show that both GRAPE and Blogel incurs communication costs that are orders of magnitudes smaller than those of GraphLab and Giraph (whose curves coincide). For instance, GRAPE ships 0.07% of the data shipped by GraphLab (same for Giraph) on DBpedia. 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 30% and 0.9% of the data shipped by Blogel over liveJournal and DBpedia, respectively. This is because GRAPE ships only updated values. The improvement over Blogel on traffic is not substantial because the road network has a small average node degree, and hence imposes a smaller bound (worst-case data shipment) on the improvement of GRAPE over Blogel.

(2) CC. Figures 8(d)-8(f) demonstrate similar improvement of GRAPE over GraphLab and Giraph for CC, *e.g.,* on average GRAPE ships 5.4% of the data shipped by Giraph and GraphLab. Blogel is slightly better than GRAPE. As remarked in Section 7 for Exp-1(2), this is because Blogel precomputes CCs of graphs when partitioning and loading the graphs, and thus already recognizes connected components by using an internal partition strategy. While a fair comparison should include the time for precomputing CCs in the evaluation time of CC by Blogel, we cannot identify the communication cost saved by its preprocessing. Thus, the reported communication cost of Blogel is almost 0 in all cases. Nonetheless, GRAPE incurs communication cost comparable to the near "optimal" case reported by Blogel, when Blogel operates on a graph that is already partitioned as CCs.

(3) Sim. Figures 8(g) and 8(h) report the communication cost for graph simulation over liveJournal and DBpedia, respectively. One can see that GRAPE ships substantially less data, *e.g.,* on average 1.3%, 1.3% and 1.6% of the data shipped by Giraph, GraphLab and Blogel, respectively. Observe that here the communication cost of Blogel is much higher than that of GRAPE, even though Blogel adopts inter-block communication. This shows that the extension of vertex-centric to block-centric by Blogel does not help much on more complex queries. GRAPE works better than vertex-centric and block-centric systems on complex queries, by employing incremental IncEval to reduce excessive messages.

(4) SubIso. Figures 8(i) and 8(j) report the results for SubIso over liveJournal and DBpedia, respectively. The results are consistent with Sim queries. On average, GRAPE ships 4.7%, 4.7%, and 6.5% of the data shipped by Giraph, GraphLab and Blogel, respectively. Due to the locality of



(a) Varying $|G|$: Sim (b) Varying $|G|$: SubIso (c) Varying $|G|$: CC (d) Varying $|G|$: SSSP

Figure 9: Scalability on synthetic graphs

```
class ShortestPathVertex (Vertex<int, int, int>) {
  void Compute (MessageIterator* msgs) {
  int mindist = IsSource(vertex_id())? 0: INF;
  for (; !msgs->Done();msgs->Next())
    mindist = min(mindist, msgs->Value());
  if (mindist < GetValue()) {
    *MutableValue() = mindist;
    OutEdgeIterator iter = GetOutEdgeIterator();
    for (; !iter.Done(); iter.Next())
      SendMessageTo(iter.Target(), mindist + iter.GetValue()); }
  VoteToHalt(); }}
```

Figure 10: Giraph vertex program SSSP

subgraph isomorphism, matches to a pattern are confined in connected blocks. Hence Blogel takes advantage of its CC preserving graph partition, and does better than the case for Sim. Nevertheless, GRAPE only ships 6.5% of the data shipped by Blogel on average, and outperforms Blogel.

(5) CF. Figures 8(k) and 8(l) report the results for CF over movieLens, with 90% and 50% training set E_T, respectively. On average, GRAPE ships 9.2%, 9.2%, and 10.3% of the data shipped by Giraph, GraphLab and Blogel, respectively, for $|E_T| = 90\%\ |E|$; and 7.6%, 7.6%, and 8.0% for $|E_T| = 50\%\ |E|$. This verifies that GRAPE is effective in reducing the communication cost of CF even for algorithms that favor vertex-centric programming. It also shows that GRAPE remains effective when the amount of training data varies.

Exp-5: Scalability. We also evaluated the scalability of GRAPE over larger synthetic graphs. We developed a generator to produce graphs $G = (V, E, L)$ with L drawn from an alphabet $\mathcal{L}$ of 50 labels. It is controlled by the numbers of nodes $|V|$ and edges $|E|$, up to $50M$ and $200M$, respectively.

Fixing n = 24, we varied $|G|$ from $(10M, 40M)$ to $(50M, 200M)$. As reported in Fig. 9, we tested SSSP, CC, Sim and SubIso; as the "true" behavior of CF is better characterized by real-world data, we omit the performance of CF over the synthetic data. The results are consistent with Fig. 6 over real-life graphs. (a) All systems take longer when G gets larger, as expected. (b) GRAPE scales reasonably well with the increase of $|G|$. With $|G|$ increased by 5 times, the running time of GRAPE increases by 7 times, 2.7 times, 6 times and 12 times, for SSSP (with linear time sequential algorithm), CC (linear time), Sim (quadratic time) and SubIso (exponential time), respectively. (c) GRAPE consistently outperforms Giraph and GraphLab for all queries, by 2.1 and 1.5 times for SSSP, 5.3 and 4.6 times for CC, 3 and 2.4 times for Sim, and 1.7 and 1.4 times for SubIso. The gap for SSSP is smaller than it on traffic, due to the special features of traffic mentioned earlier. GRAPE is 1.1 times faster than Blogel for SSSP, 1.3 for Sim, and 1.3 for SubIso. Blogel does better than GRAPE on CC for the reasons given above.

Exp-6: Ease of programming. We also inspected the usability of GRAPE. Taking SSSP as an example, we examined (a) vertex-centric programs for Giraph (similarly for GraphLab), and (b) block-centric programs for Blogel. Parts of the Giraph and Blogel algorithms are shown in Figures 10 and 11, respectively. We adopt the Giraph code taken from [35], and use the Blogel code from its developers.

```
 void VCompute(Messages) { /*V-mode computing*/
1. if ( step == 1) {
2.    ··· /* initialize source distance, vote to halt otherwise */ }
3. else { for (msg : Messages) {
4.    ··· /* update local distance with minimum one in Messages*/ }}
 void BCompute(Messages, Container) {/*B-mode computing*/
1. for (vertex : Container) {
2.    if (vertex.isactive()) { heap.add(vertex); }}
3. while (heap.size > 0) { /*recasted Dijkstra's algorithm*/
4.    u = heap.peek(); edges = u.value().edges; split = u.value().split;
5.    for (edge : edges[0...split]) {
6.       ··· /* invokes V-mode computing for each in-block node*/}
7.    for (edge : edges[split...edge.size()]){
8.       ··· /* out-block msg passing */ }
9.    voteToHalt(); } }
```

Figure 11: Blogel block program for SSSP

Comparing these programs with their GRAPE counterpart (Figures 3 and 4), we find the following.

(1) The vertex program for Giraph requires substantial changes to its corresponding sequential algorithm. As shown in Fig. 10, the logic flow of a Giraph program for SSSP is quite different from that of a sequential SSSP algorithm. Writing such programs requires users to have prior knowledge about the query classes and the design principle of the vertex-centric model. Moreover, it is challenging to integrate graph-level optimization, *e.g.,* incremental evaluation, into the vertex programming model. In contrast, the logic flow of PIE algorithms (GRAPE) remains the same as those sequential algorithms adapted for PEval and IncEval.

Similar to Giraph, GraphLab code for SSSP (not shown) requires users to recast the sequential SSSP algorithm into vertex programs. For example, a sequential operation in an SSSP algorithm that "collects the distances from the neighbors of a node and updates the distances" is broken down to two core functions as follows: (a) the "Apply" function updates the local distance at each vertex; and (b) the "Scatter" function propagates the updated value to the neighbors of a node. In contrast, a GRAPE program keeps the integrity of this operation for all the nodes within a fragment.

(2) While Blogel supports block-centric computation, it also requires recasting of sequential algorithms, as shown in Fig. 11. Indeed, Blogel programming extends vertex-centric algorithms (*e.g.,* Giraph) by treating each block as a "virtual vertex", while still retaining the same message passing strategies for blocks as in the vertex-centric algorithms. Hence, its logic flow is along the same lines as Giraph algorithms, and requires recasting of sequential algorithms.