Parallelizing Sequential Graph Computations

WENFEI FAN*, University of Edinburgh, Beihang University, and Shenzhen Institute of Computing Sciences
WENYUAN YU, JINGBO XU, JINGREN ZHOU, XIAOJIAN LUO, QIANG YIN, Alibaba Group
PING LU, BDBC, Beihang University
YANG CAO, RUIQI XU, University of Edinburgh

This paper presents GRAPE, a parallel GRAPh Engine for graph computations. GRAPE differs from prior systems in its ability to parallelize existing sequential graph algorithms as a whole, without the need for recasting the entire algorithms into a new model. Underlying GRAPE are a simple programming model, and a principled approach based on fixpoint computation that starts with partial evaluation and uses an incremental function as the intermediate consequence operator. We show that users can devise existing sequential graph algorithms with minor additions, and GRAPE parallelizes the computation. Under a monotonic condition, the GRAPE parallelization guarantees to converge at correct answers as long as the sequential algorithms are correct. 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.

CCS Concepts: • **Information systems → Database management system engines**; **Parallel and distributed DBMSs**;

Additional Key Words and Phrases: Graph computations; parallel graph query engines; parallelizing sequential algorithms; convergence; simulation

ACM Reference Format:
Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, and Yang Cao, Ruiqi Xu. 2000. Parallelizing Sequential Graph Computations. *ACM Trans. Datab. Syst.* 0, 0, Article 38 (2000), 41 pages. https://doi.org/0000000.0000000

1 INTRODUCTION

Several parallel systems have been developed for graph computations, *e.g.*, Pregel [50], GraphLab [49], Giraph++ [63] and Blogel [71]. These systems, however, require users to recast graph algorithms into their models. While graph computations have been studied for decades and a number of sequential (single-machine) graph 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

*Corresponding author

Authors' addresses: Wenfei Fan, University of Edinburgh, 10 Crichton Street, Edinburgh, UK, EH8 9AB, Beihang University, 37 Xue Yuan Road, Haidian District, Beijing, China, 100191, Shenzhen Institute of Computing Sciences, Shenzhen, China, wenfei@inf.ed.ac.uk; Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Alibaba Group, 969 West Wen Yi Road, Yu Hang District. Hangzhou, China, 311121, {wenyuan.ywy,xujingbo.xjb,jingren.zhou,lxj193371,qiang.yq}@alibaba-inc.com; Ping Lu, BDBC, Beihang University, 37 Xue Yuan Road, Haidian District, Beijing, China, 100191, luping@buaa.edu.cn; Yang Cao, Ruiqi Xu, University of Edinburgh, 10 Crichton Street, Edinburgh, UK, EH8 9AB, {yang.cao,ruiqi.xu}@ed.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2000 Copyright held by the owner/author(s). Publication rights licensed to ACM.

0362-5915/2000/0-ART38 $15.00
https://doi.org/0000000.0000000

System	Category	Time(second)	Communication(MB)
Giraph	vertex-centric	434.0	113411.1
GraphLab	vertex-centric	41.7	106756.3
Blogel	block-centric	112.3	123377.0
GRAPE	semi-auto parallelization	24.3	14744.8

Table 1. Graph traversal on parallel systems

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 provide sequential (single-machine) graph algorithms as a whole (subject to minor changes), and it parallelizes the computation across multiple processors, without drastic degradation in either performance or functionality of the existing systems?

GRAPE. To answer this question, we develop GRAPE, a parallel GRAPh Engine for graph computations such as graph traversal, graph pattern matching, graph connectivity and collaborative filtering. Using our familiar terms, we refer to a graph computation problem $\mathcal{Q}$ as a class of queries, and an instance Q of $\mathcal{Q}$ as a *query* of $\mathcal{Q}$. GRAPE 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 (existing) 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 parallel". This makes parallel graph computations accessible to a large group of users who know conventional graph algorithms covered in undergraduate textbooks.

(2) Termination and correctness. GRAPE parallelizes the sequential algorithms based on a combination of partial evaluation and incremental computation. It guarantees to converge at correct answers under a monotonic condition, as long as the sequential algorithms provided are correct.

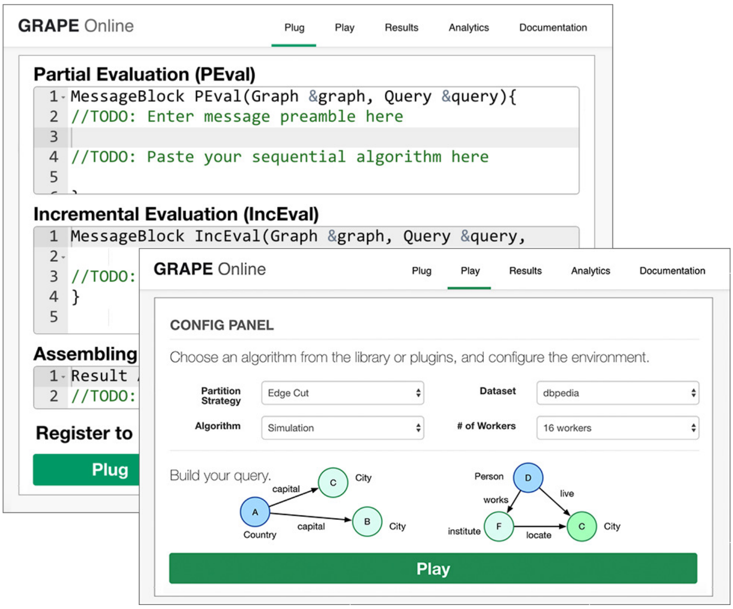
(3) Graph-level optimization. GRAPE naturally inherits all optimization strategies available for sequential algorithms and graphs, *e.g.,* indexing, compression and partitioning. In contrast, these strategies are hard to implement for vertex-centric programs.

(4) Scalability. The ease of programming does not imply performance degradation compared with the state-of-the-art systems, *e.g.,* vertex-centric Giraph [6] (Pregel) and GraphLab, and block-centric Blogel. For instance, Table 1 shows the performance of the systems for single source shortest-path queries (SSSP) over Friendster [4], a social network with 65 million users and 1.8 billion relationships (edges), using 192 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 $(P_1, \ldots, P_n)$, where F_i resides at P_i for $i \in [1, n]$, respectively. Given a query $Q \in \mathcal{Q}$ and a fragmented graph 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 $f(s, d)$ with respect to the known input s [42]. 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, its local fragment F_i is its known input s, while the data residing at other processors accounts for the yet unavailable input d. GRAPE computes $Q(F_i)$ at all processors P_i's *in parallel as partial evaluation.*

Incremental computation. Graph computations are often *iterative*. If $Q(G)$ cannot be obtained in one step by combining partial results $Q(F_i)$, 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



Partial Evaluation (PEval)

```
MessageBlock PEval(Graph &graph, Query &query){
//TODO: Enter message preamble here

//TODO: Paste your sequential algorithm here

```

Incremental Evaluation (IncEval)

```
MessageBlock IncEval(Graph &graph, Query &query,

//TODO:
}

```

Assembling

```
Result 
//TODO:
```

Register to

CONFIG PANEL

Choose an algorithm from the library or plugins, and configure the environment.

Partition Strategy: Edge Cut
Dataset: dbpedia
Algorithm: Simulation
# of Workers: 16 workers

Build your query.

Fig. 1. Programming Interface of GRAPE

to $Q(F_i)$ such that $Q(F_i \oplus M_i) = Q(F_i) \oplus \Delta O_i$, making maximum reuse of previous results $Q(F_i)$. Here (1) M_i is a message designated to worker P_i, where fragment F_i resides; (2) $F_i \oplus M_i$ is the abbreviation of the following steps: (a) deduce the change ΔF_i to F_i from M_i (we will show how to deduce the change in Section 3.2); and then (b) apply the change ΔF_i to fragment F_i; and (3) $Q(F_i) \oplus \Delta O_i$ is to apply the change ΔO_i to the old result $Q(F_i)$. It is to minimize the use of notations that we reload the notation $\oplus$; it will be clear from the context what operation $\oplus$ means. Incremental computation 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, it may be *bounded*: its cost depends only on the 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 fragment F_i [29, 57], minimizing unnecessary recomputation.

Workflow. Based on partial evaluation and incremental computation, GRAPE works as follows.

(1) Plug. GRAPE offers a simple programming interface as shown in Figure 1. For a class $\mathcal{Q}$ of graph queries, developers need to specify three functions: PEval, IncEval and Assemble in the algorithm panel. PEval and IncEval are (often existing) *sequential (single-machine) algorithms* for $\mathcal{Q}$, for partial evaluation and incremental computation, respectively; and Assemble is typically straightforward (see examples shortly). These 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 such a specification (PEval, IncEval and Assemble) registered for $\mathcal{Q}$, a graph G, a graph partition strategy and a number n of processors to work with (Figure 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 Figure 2.

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

(b) Then each processor P_i may exchange partial results with other processors via synchronous message passing under BSP [65]. Upon receiving message M_i, processor P_i *incrementally* computes local answer $Q(F_i \oplus M_i)$ by IncEval, operating on its local fragment F_i "updated" by M_i.

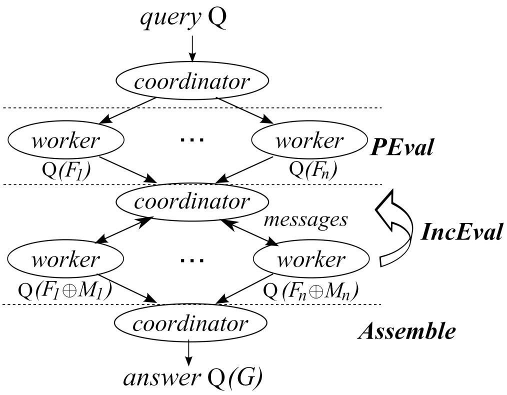


Fig. 2. Workflow of GRAPE

(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 $Q(F_i \oplus M_i)$ from P_i for $i \in [1, n]$ and assembles $Q(G)$.

That is, GRAPE parallelizes sequential algorithms as a whole, and computes a simultaneous fixpoint by taking IncEval as the intermediate consequence operator. 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 using IncEval, and supports graph-level optimization on F_i.

Example 1.1. Consider Single Source Shortest Path (SSSP), a routine graph computation problem. Given a directed graph G with edges labeled with positive weights, and a source node s in G (as a query Q), the goal 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 [32] as PEval, and a bounded sequential incremental algorithm of [56, 57] as IncEval. The algorithm of [56] essentially propagates changes to vertices or edges to other vertices in the graph following an order defined on some "keys" of the affected vertices. Similarly, the algorithm by [57] handles unit changes to graphs. The only addition to GRAPE is that for each fragment F_i, a variable dist(s, v) of positive numbers is declared for each node v, initially ∞ (except dist$(s, s) = 0$). As shown in Figure 2, PEval first computes $Q(F_i)$; it then repeats incremental steps 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 algorithms into a new model. □

Contributions. We propose GRAPE, which suggests a new approach to parallelizing (existing) sequential graph algorithms, from foundation to implementation.

(1) We introduce the parallel model of GRAPE, based on a fixpoint of partial evaluation and (bounded) incremental computation (Section 3). We also present the programming model of GRAPE. We show how GRAPE takes *existing* sequential graph algorithms as input, and parallelizes the entire algorithms, in contrast to parallelization of instructions or operators [53, 58].

(2) We prove two fundamental results (Section 4): (a) Assurance Theorem guarantees that for all queries $Q \in \mathcal{Q}$ and graphs G, GRAPE converges at correct answers $Q(G)$ under a monotonic condition as long as its input sequential algorithms are correct, and (b) Simulation Theorem shows

that MapReduce [22], BSP (Bulk Synchronous Parallel) [65] and PRAM (Parallel Random Access Machine) [66] can be optimally simulated by GRAPE. As a consequence, algorithms developed for existing graph systems can be migrated to GRAPE without increasing complexity bounds.

(3) As examples, we show that a variety of graph computations can be readily parallelized by GRAPE (Section 5). These include graph traversal (single-source shortest path queries SSSP), graph pattern matching (via graph simulation Sim and subgraph isomorphism SubIso), connected components (CC), and collaborative filtering (CF, as an example of 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 graph-level optimization strategies such as indexing, compression and dynamic grouping, since the sequential algorithms directly operate on fragments of a graph, which are graphs themselves. These 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, an open-source version of Pregel, (b) GraphLab, another 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) on a US road network traffic [2], GRAPE is on average 14842, 3992 and 756 times faster than Giraph, GraphLab and Blogel for SSSP, respectively, with 192 processors, due to the large diameter of the graph. (b) On other real-life graphs excluding traffic, GRAPE is on average 484, 36 and 15 times faster than Giraph, GraphLab and Blogel for SSSP, 151, 6.8 and 16 times for Sim, 149.3, 34.2 and 9.6 times for SubIso, and 4.6, 2.6 and 12.4 times for CF, respectively, when the number of processors ranges from 64 to 192. (b) In the same setting (excluding traffic), GRAPE ships on average 0.07%, 0.12% and 1.7% of the data shipped across machines by Giraph, GraphLab and Blogel for SSSP, 0.89%, 0.14% and 4.9% for Sim, 0.18%, 0.23% and 0.11% for SubIso, 5.6%, 43.3% and 3.2% for CF, respectively. When traffic is also included, GRAPE outperforms these systems by up to 6 orders of magnitude in communication cost for SSSP. (c) Incremental steps effectively reduce the cost and improve the performance of Sim by 9.6 times on average. (d) Optimization strategies for sequential algorithms remain effective for GRAPE and improve Sim by 20% on average.

Related work. This paper extends its conference version [31] as follows. (1) We provide proofs of Assurance Theorem and Simulation Theorem (Section 4). (2) We develop algorithms for graph simulation Sim, subgraph isomorphism SubIso, connected components CC, and collaborative filtering CF (Section 5). The details of the algorithms were not included in [31]. (3) We re-conduct all experiments of [31] by using larger graphs and more processors, to evaluate the scalability of GRAPE with the size of datasets and the parallel scalability with the number of processors, as well as the communication costs. We also add a COST [51] analysis of GRAPE (Section 7).

The other related work is categorized as follows.

Parallel graph systems. Several parallel models have been studied for graph computations, *e.g.,* PRAM [66], BSP [65] and MapReduce [22]. PRAM abstracts parallel RAM access over shared memory. A large collection of parallel graph algorithms are in place for PRAM. These algorithms may need to be optimized for the shared-nothing architecture that is widely used today. MapReduce makes parallel computation accessible to a large number of users, but may not be very efficient for iterative graph computations due to its blocking nature and I/O costs. BSP has proven more appropriate for graph computations. It models parallel computations in supersteps (including local computation, communication and a synchronization barrier) to synchronize communication among workers, such that messages from one superstep are accessible in the next one. Alternatively, AP

(Asynchronous Parallel) model allows a worker to have immediate access to incoming messages. Vertex-centric models [6, 49, 50] execute local computations defined at each vertex in parallel.

Several parallel graph systems have been developed under these models. Pregel [50] (Giraph [6]) implements BSP with vertex-centric programming, where a superstep executes a user-defined function at each vertex in parallel. GraphLab [49] supports both BSP and AP with vertex-centric programming. Block-centric systems [63, 71] extend vertex-centric programming to blocks, to exchange messages among blocks and reduce communication costs. Giraph++ [63] supports graph-centric programming to open up subgraphs to be programmed against. Blogel [71] allows blocks to have their status as a "vertex" and supports block-level communication. Nonetheless, both Giraph++ and Blogel still adopt the vertex-centric style programming paradigm.

Popular parallel graph systems also include GraphX [34], GRACE [67], GPS [59], Mizan [46], Pregel+ [72] (see [70] for a recent survey). GraphX [34] recasts graph computation into its distributed dataflow framework as a sequence of join and group-by stages punctuated by map operations, on Spark platform. GRACE [67] provides an operator-level, iterative programming model to enhance BSP with asynchronous execution. It provides a scheduler to control the order of vertex computation in a block. The system works in single-machine environment, with a focus on improving main memory utilization. GPS [59] implements Pregel with extended APIs and partition strategies. It advocates an optimization strategy such that algorithms can "post" messages, for a vertex to send the same message to all of its neighbors, by partitioning the neighbors of high-degree nodes across different processors (via their mirrored nodes). Mizan [46] optimizes Pregel with dynamic load balancing based on run-time monitoring of vertex computation. Pregel+ [72] introduces optimized message reduction similar to GPS but with an additional cost model to trade off mirroring and message combining costs. Both Mizan and Pregel+ are based on Pregel's model.

All these systems require recasting of existing sequential algorithms into a new model. For example, synchronous vertex programs are required by Pregel-like systems [6, 46, 50, 59, 63, 72]. Sequential algorithms need to be recast to Gather-Aggregate-Scatter (GAS) vertex programs in GraphLab [49]. When it comes to block-centric models [63, 71], sequential algorithms have to be recast to block-level programs that treat each subgraph as a single vertex.

This work aims to show that it is possible to parallelize existing sequential graph algorithms as a whole, without recasting the algorithms into a new model. GRAPE can simplify parallel programming and make parallel graph computations accessible to a large group of users, without drastic degradation in performance or functionality. To this end, GRAPE adopts the synchronization mechanism of BSP for its simplicity. As opposed to the prior systems, (a) GRAPE parallelizes sequential algorithms based on fixpoint computation with partial evaluation and incremental computation. (b) Following data-partitioned parallelism, given a partitioned graph, GRAPE allows workers to operate on different fragments in parallel, and exchanges among workers only the updated values of the status variables associated with border nodes. In contrast, for iterative computations, MapReduce needs to repartition graph and ship its entire state in each round [50]. (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 "inter-processor" 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 optimization methods that are hard to implement in vertex-centric systems; similarly for GraphLab (asynchronized). (d) Closer to GRAPE are block-centric models [63, 71]. However, the programming interface of [63] is still vertex-centric, and [71] 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 takes sequential algorithms PEval and IncEval from GRAPE library, and applies them to blocks in parallel without recasting. (e) None of the prior systems uses (bounded) incremental steps to speed up iterative computations. (f) To the best of our knowledge, none of these systems provides assurance on termination and the correctness of parallel graph computations.

Partial evaluation has been studied for certain XML [19] and graph queries [29]. There has also been a host of work on incremental graph computation (*e.g.,* [24, 29, 57]). 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 algorithms have been developed in MapReduce, vertex-centric models and others [29, 73]. 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 [54, 58] by breaking dependencies via symbolic and automatic analyses. There has also been work at data partition level [75], to perform multi-level partition ("parallel abstraction") and adapt locality-optimized access to different parallel abstraction. In contrast, GRAPE aims to parallelize sequential algorithms as a whole, and make parallel computation accessible to end users, while [54, 58, 75] target experienced developers of parallel algorithms. There have also been tools for translating imperative code to MapReduce, *e.g.,* word count [55]. 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 [54, 55, 58, 75] 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 [38]. There has also been recent work on simulation of PRAM on MapReduce and BSP [43]. In particular, [43] defines a framework $\mathcal{MRC}$ for MapReduce computations and shows that a large class of PRAM algorithms can be efficiently simulated by $\mathcal{MRC}$ with certain restrictions. This work extends [31] by providing optimal deterministic simulation results of MapReduce, BSP and PRAM on GRAPE, adopting the notion of optimal simulations of [66].

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.

We use two notions of subgraphs. A 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. edge $e \in E'$), $L'(v) = L(v)$ (resp. $L'(e) = L(e)$). A 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 graph G and a number m, a graph partition strategy $\mathcal{P}$ partitions G into *fragments* $\mathcal{F} = (F_1, \ldots, F_m)$ such that each $F_i = (V_i, E_i, L_i)$ is a subgraph of G, $E = \bigcup_{i \in [1,m]} E_i$, $V = \bigcup_{i \in [1,m]} V_i$, and F_i resides at processor P_i for $i \in [1, m]$. In vertex partition (*a.k.a.* edge-cut) [12, 18]), a cut edge from fragment F_i to F_j has a copy in each of F_i and F_j. Denote by

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

symbols	notations
$\mathcal{Q}$	a class of graph queries
Q	a 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 partition strategy $\mathcal{P}$
F_i	the i-th fragment of graph G via partition strategy $\mathcal{P}$
V_i, E_i	vertex set and edge set of fragment F_i, respectively
$\mathcal{F}$	fragmentation $(F_1, \ldots, F_n)$ of graph G
M_i	messages designated to worker P_i
$F_i \oplus M_i$	fragment F_i updated with message M_i

Table 2. Notations

We refer to those nodes in $F_i.I \cup F_i.O'$ as the *border nodes* of fragment F_i *w.r.t.* partition strategy $\mathcal{P}$. Note that $\mathcal{F}.I = \mathcal{F}.O$ and $\mathcal{F}.I' = \mathcal{F}.O'$.

In edge partition (*a.k.a.* vertex-cut) [47], the cut vertices are called entry vertices and exit vertices for the partitions, which correspond to the sets $\mathcal{F}.O \ \cup \mathcal{F}.I'$ and $\mathcal{F}.I \ \cup \mathcal{F}.O'$, respectively. In general, a border node is a vertex that relates to vertices or edges in two different fragments.

The *fragmentation graph* $G_{\mathcal{P}}$ of G via $\mathcal{P}$ is an index such that given each border node v in $F_i.I$ (resp. $F_i.O'$) ($i \in [1, m]$), $G_{\mathcal{P}}(v)$ retrieves a set of $\{j \mid v \in F_j.O\}$ (resp. $\{j \mid v \in F_j.I'\}$). As will be seen shortly, we will make use of $G_{\mathcal{P}}$ to deduce the directions of messages.

The notations of this paper are summarized in Table 2.

3 PROGRAMMING WITH GRAPE

Below we first introduce the parallel model of GRAPE. We then show how to program with GRAPE. Following BSP [65], GRAPE works with a *coordinator* P_0 and a set of m *workers* $P_1, \ldots, P_m$.

3.1 The Parallel Model of GRAPE

Given a graph partition strategy $\mathcal{P}$ and sequential algorithms PEval, IncEval and Assemble for a class $\mathcal{Q}$ of graph queries, GRAPE parallelizes the computations as follows. It first partitions graph G into fragments $\mathcal{F} = (F_1, \ldots, F_m)$ with strategy $\mathcal{P}$, and distributes the fragments across m shared-nothing *virtual workers* $(P_1, \ldots, P_m)$. It maps m virtual workers to n physical workers such that fragment F_i resides at worker P_i for $i \in [1, m]$. When $n < m$, multiple virtual workers mapped to the same worker share memory. It also constructs fragmentation graph $G_{\mathcal{P}}$. Note that graph G is partitioned *once* for *all queries* $Q \in \mathcal{Q}$ posed on graph G.

Parallel model. Given a query $Q \in \mathcal{Q}$, GRAPE computes answer $Q(G)$ to Q in the partitioned graph G, as shown in Figure 2. Upon receiving Q at coordinator P_0, GRAPE posts the same query Q to all the workers. To simplify the discussion, here we adopt synchronous message passing following BSP [65]. We will show how GRAPE implements point-to-point communication in Section 6. Furthermore, GRAPE also works under asynchronous parallel models [27].

Its parallel computation consists of the following three phases.

(1) Partial evaluation (**PEval**). In the first superstep, upon receiving query Q, each worker P_i computes partial result $Q(F_i)$ locally at its fragment F_i using PEval, in parallel (for $i \in [1, m]$). It also identifies and initializes a set of update parameters for each F_i that records the status of certain nodes, *e.g.*, 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 for update parameters).

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

(2.a) *Coordinator*. Coordinator P_0 checks whether for all $i \in [1, m]$, worker P_i is inactive, *i.e.,* P_i is done with its local computation and there exists no pending message designated for P_i. If so, GRAPE invokes Assemble and terminates (see below). Otherwise, P_0 composes a message M_i by aggregating messages from the last superstep (see details shortly), sends M_i to worker P_i for $i \in [1, m]$, 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 $i \in [1, m]$. Here $F_i \oplus M_i$ denotes the fragment F_i that is updated with message M_i, *i.e.,* F_i after its update parameters are changed with the values in M_i. At the end of the process, IncEval automatically finds the changes to the update parameters in each F_i, and sends the changes as a message to coordinator P_0 (see Section 3.3 for details).

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, to minimize unnecessary recomputation.

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

We will show in Section 4 that the parallel process converges at correct answers under a monotonic condition as long as the sequential algorithms PEval, IncEval and Assemble are correct; moreover, the simple parallel model does not lose expressive power.

3.2 PEval: Partial Evaluation

We now introduce the programming model of GRAPE. GRAPE provides a programming interface for users to extend (existing) sequential algorithms with message declarations. GRAPE registers the algorithms as stored procedures in its API library (Figure 1), and maps them to a query class $\mathcal{Q}$.

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

Function 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 for $\mathcal{Q}$. One only needs to extend it with the following additions:

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

Communication among workers is conducted via message passing. Messages are defined in terms of *update parameters* of each fragment F_i as follows.

(1) Message preamble. Function PEval (a) declares *status variables* $\bar{x}$ associated with vertices and edges for each fragment F_i, and (b) specifies a set C_i of nodes and edges relative to $F_i.I$ or $F_i.O'$ *w.r.t.* each fragment F_i. The status variables associated with C_i are denoted by $C_i.\bar{x}$, and are referred to as the *update parameters* of F_i. 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.

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, in GRAPE, 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. In most cases $d = 0$ and C_i is $F_i.I$ or $F_i.O'$. However, in some applications one needs $d \geq 0$, *e.g.*, subgraph isomorphism (SubIso, see Section 5.1). In such cases C_i may include nodes and edges from other fragments F_j of G.

A message M_i is a set of key-value pairs $\langle x, \mathsf{val}\rangle$, where x is a status variable declared in $C_i.\bar{x}$ and val is its value. GRAPE supports arbitrarily typed status variables, *e.g.*, val of M_i can be a numeric value (*e.g.*, in the algorithm for SSSP in Example 3.1), a multi-set of tuples $\langle r, \mathit{key}, \mathit{value}\rangle$ (in the simulation of MapReduce in the proof of Theorem 4.2), or even a user defined structure (a class).

(2) Message segment. PEval specifies 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 $C_i.\bar{x}$ 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.

(b) *Composing M_i*. For messages from each P_i, GRAPE does the following:

(i) it identifies variables in $C_i.\bar{x}$ with *changed values*;

(ii) it deduces the designations P_j of the messages by referencing the fragmentation graph $G_\mathcal{P}$; if $\mathcal{P}$ is edge-cut, the variable tagged with a node v in $F_i.I$ will be sent to worker P_j if v is in $F_j.O$ (*i.e.*, if j is in $G_\mathcal{P}(v)$); similarly for v in $F_i.O'$; 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 variable 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. When a node v has copies $v_i \in F_i$ and $v_j \in F_j$ residing in different fragments, *e.g.*, when v is a border node in $F_i.O \cap F_j.I$ $(i \neq j)$, $v_i.x$ and $v_j.x$ are treated as the same status variable x and are assigned the same value.

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 (see Section 6).

Example 3.1. We show how GRAPE parallelizes SSSP (see Example 1.1). 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 nodes $v \in V$. It adopts edge-cut partition [18]. It deduces $F_i.O$ by referencing $G_\mathcal{P}$, and stores $F_i.O$ at each fragment F_i.

As shown in Figure 3, PEval (lines 1-14) is *verbally identical* to Dijkstra's sequential algorithm [32]. 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

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

Message preamble: /*candidate set $C_i$ is $F_i.O$*/
  for each node $v \in V_i$, a float variable dist$(s, v)$;

/*sequential algorithm for SSSP (pseudo-code)*/
initialize priority queue Que;
dist$(s, s)$ := 0;
for each $v$ in $V_i$ do
  if $v! = s$ then
     dist$(s, v)$ := $\infty$;
Que.addOrAdjust($s$, dist$(s, s)$);
while Que is not empty do
  $u$ := Que.pop(); /* pop vertex with minimal distance */
  for each child $v$ of $u$ do /* only $v$ that still  has not been visited */
    $alt$ := dist$(s, u)$ + $L_i(u, v)$;
    if $alt$ < dist$(s, v)$ then
      dist$(s, v)$ := $alt$;
      Que.addOrAdjust($v$, dist$(s, v)$);
$Q(F_i)$ := {dist$(s, v)$ | $v \in V_i$}

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

Fig. 3. PEval for SSSP

for each dist(s, v). It finds those variables with smaller values, deduces their destinations P_j by referencing fragmentation graph $G_{\mathcal{P}}$, groups them into messages M_j, and sends M_j to worker P_j. □

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. Here $F_i \oplus M_i$ denotes F_i updated with M_i. 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.

Function IncEval can be any existing sequential incremental algorithm for $\mathcal{Q}$. It shares the message preamble of PEval. At the end of the process, it identifies *changed values* to $C_i.\bar{x}$ at each fragment F_i, and sends the changes as messages to P_0. Upon receiving the messages at coordinator P_0, GRAPE composes these messages as described in 3(b) in Section 3.2.

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

Consider an incremental algorithm IncEval 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 $Q(G)$. It is said to be *bounded* if its cost can be expressed as a function in the size of $|\text{CHANGED}| = |M| + |\Delta O|$, *i.e.,* the size of changes in the input and output [28, 57]. Intuitively, $|\text{CHANGED}|$ represents the updating costs inherent to the incremental problem for $\mathcal{Q}$ itself. For a bounded IncEval, its cost is determined by $|\text{CHANGED}|$, not by the size $|F_i|$ of entire F_i, no matter how big $|F_i|$ is. That is, it reduces computation on possibly big F_i to smaller data bounded by $O(|\text{CHANGED}|)$.

Example 3.2. Continuing with Example 3.1, we provide IncEval in Figure 4. It is the sequential incremental algorithm for SSSP developed in [56, 57], in response to changed dist(s, v) for v in $F_i.I$ (here message M_i includes changes to dist(s, v) for $v \in F_i.I$ deduced from $G_{\mathcal{P}}$). Using a queue Que,

```
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 /* only $v$ that still  has not been visited  */
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$\}$;
```

Fig. 4. IncEval for SSSP

it starts with M_i, propagates the changes to affected area, and updates the distances (see [56, 57] for details). 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 changes to the algorithm of [56, 57] are underlined in Figure 4. Following [56, 57], 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). □

Note that IncEval only needs to deal with changes M_i, *e.g.*, changes to $\mathsf{dist}(s, v)$ for $v \in F_i.I$ in Example 3.2. That is, changes are restricted to the update parameters, rather than generic updates.

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 complete query answer $Q(G)$. It is triggered when no more changes can be made to update parameters $C_i.\bar{x}$ for any $i \in [1, m]$.

Example 3.3. Continuing with Example 3.2, function 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 $Q(G)$. Indeed, 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 of [32, 56, 57] (*i.e.,* 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 (PEval) and a sequential incremental algorithm (IncEval) for $\mathcal{Q}$. Moreover, Assemble is typically a straightforward algorithm. A large number of sequential (incremental) algorithms are already in place for various $\mathcal{Q}$, after decades of study of graph computations. Thus GRAPE is promising for making parallel graph computations accessible to a large group of users.

Remark. Observe the following about PEval, IncEval, and Assemble.

(1) There have been methods for incrementalizing algorithms, to get incremental algorithms from their batch counterparts [11, 24]. Moreover, incremental algorithm IncEval only needs to deal with changes to status variables (update parameters), not necessarily generic updates (although to focus on the main idea, we present IncEval using the familiar notion of incremental graph algorithms).

Such changes are aggregated by function f_{aggr} and depend on how f_{aggr} is defined. Hence it is often not hard to develop IncEval by revising a batch algorithm in response to changes to update parameters, as will be shown by the case of CC (connected components) in Section 5.2.

(2) Incremental IncEval speeds up iterative computations by minimizing unnecessary recomputation of $Q(F_i)$ at each worker P_i, *no matter whether* IncEval *is bounded or not*. Indeed, boundedness is not the only criterion for the effectiveness of incremental algorithms. Alternative performance guarantees for incremental graph algorithms have been developed, such as semi-boundedness [28], localizable incremental algorithms and relative boundedness [24].

(3) In contrast to existing graph systems, GRAPE parallelizes sequential algorithms PEval and IncEval as a whole, with additional declaration of message segment (PEval). As a result, users do not have to "think like a vertex" [49, 50, 63, 71] 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.

(4) GRAPE aims to help users develop parallel programs, especially those who are more familiar with conventional sequential programming. This said, users of GRAPE still need to know the domain knowledge to design update parameters and aggregate functions.

4 FOUNDATION OF GRAPE

We next present fundamental results underlying GRAPE. We first identify a condition under which a PIE program guarantees to converge at correct answers under GRAPE (Section 4.1). We then demonstrate the expressive power of GRAPE by simulating BSP, MapReduce and PARM (Section 4.2).

4.1 Correctness of Parallel Model

Consider a partition strategy $\mathcal{P}$ and a PIE program ρ for a class $\mathcal{Q}$ of graph queries, where ρ consists of functions PEval, IncEval and Assemble. Given a query $Q \in \mathcal{Q}$, a graph G and a natural number m, the GRAPE parallelization of ρ can be modeled as a simultaneous fixpoint operator defined on m fragments. More specifically, it starts with PEval for partial evaluation, and conducts incremental computation by taking IncEval as the intermediate consequence operator:

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

where for $i \in [1, m]$, r indicates a superstep; F_i^0 is the fragment F_i assigned to worker P_i by the partition of G via $\mathcal{P}$; $F_i^r[\bar{x}_i]$ is fragment F_i at the end of superstep r carrying update parameters $C_i.\bar{x}_i$; M_i indicates changes to $C_i.\bar{x}_i$ (via message); and R_i^r denotes partial results (including values of $C_i.\bar{x}_i$) computed at fragment F_i after the $(r+1)$-th superstep. The computation proceeds until it reaches r_0 such that $R_i^{r_0} = R_i^{r_0+1}$. At this point, Assemble$(R_1^{r_0}, \ldots, R_m^{r_0})$ is computed and returned.

Note that the computation does not reach a fixpoint as long as update parameters $C_i.\bar{x}_i$ keep changing. This is consistent with the parallel model of GRAPE (Section 3.1).

There has been a large body of work on fixpoint computation, to study (a) whether a fixpoint computation converges [20, 35, 52, 74]; and (b) how to accelerate fixpoint computation [35, 40, 60, 69]. In this paper, we mainly focus on (a). Issue (b) has been addressed in [27], which is based on GRAPE.

As an example, below we identify one convergence guarantee for the simple parallel model as a sufficient condition. We start with some notations.

(1) We say that PIE program ρ *terminates* under GRAPE with $\mathcal{P}$ if for all queries $Q \in \mathcal{Q}$ and all graphs G, there always 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) We say that a PIE program ρ with PEval, IncEval and Assemble is *correct for* $\mathcal{Q}$ w.r.t. $\mathcal{P}$ if for all

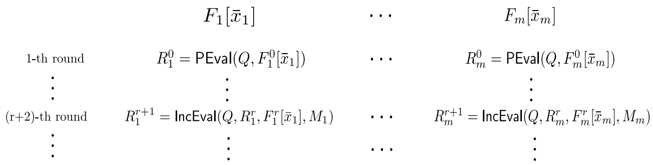

$F_1[\bar{x}_1]$
$\cdots$
$F_m[\bar{x}_m]$
1-th round
$R_1^0 = \mathsf{PEval}(Q, F_1^0[\bar{x}_1])$
$R_m^0 = \mathsf{PEval}(Q, F_m^0[\bar{x}_m])$
(r+2)-th round
$R_1^{r+1} = \mathsf{IncEval}(Q, R_1^r, F_1^r[\bar{x}_1], M_1)$
$R_m^{r+1} = \mathsf{IncEval}(Q, R_m^r, F_m^r[\bar{x}_m], M_m)$

Fig. 5. Computation of a PIE program

queries $Q \in \mathcal{Q}$ and all graphs G fragmented into $F_1, \ldots, F_m$ with $\mathcal{P}$, Assemble$(R_1^r, \ldots, R_m^r) = Q(G)$ at any superstep r when update parameters in $R_i^r (i \in [1, m])$ have the same values as in R_i^{r-1}, where $Q(G)$ is the answer to Q in G, and R_i^r is the partial result computed at F_i after the $(r+1)$-th superstep.

We say that GRAPE *correctly parallelizes* ρ with partition strategy $\mathcal{P}$ if for all queries $Q \in \mathcal{Q}$ and all graphs G, ρ always terminates under GRAPE with $\mathcal{P}$ and returns $Q(G)$.

(3) We say that PEval and IncEval satisfy the *monotonic condition w.r.t.* partition strategy $\mathcal{P}$ if for graphs G and every variable $x \in C_i.\bar{x}$ for $i \in [1, m]$, (a) the values of x are from a finite set computed from values in the active domain of G (*i.e.,* the constants in G); and (b) there exists a partial order $\leq_{p_x}$ on the values of x such that IncEval decreases x in the order of p_x.

Intuitively, condition (a) above says that x draws values from a finite domain, and condition (b) says that x is updated "monotonically" following p_x. These ensure that PIE programs with PEval and IncEval terminate under GRAPE with $\mathcal{P}$. For instance, dist(s, v) in Example 3.1 can only be changed in the decreasing order (*i.e.,* it is computed by function min for **aggregateMsg** of IncEval in the active domain of G), and hence PEval and IncEval for SSSP satisfy the monotonic condition.

We next provide a condition that warrants the correctness of GRAPE parallelization.

Theorem 4.1 [Assurance Theorem]: *Consider a* PIE *program* ρ *with* PEval, IncEval *and* Assemble *for a class* $\mathcal{Q}$ *of graph queries.* GRAPE *correctly parallelizes* ρ *with a graph partition strategy* $\mathcal{P}$ *if*

(a) PEval *and* IncEval *satisfy the monotonic condition* w.r.t. $\mathcal{P}$, *and*

(b) ρ *with* PEval, IncEval *and* Assemble *is correct for* $\mathcal{Q}$ w.r.t. $\mathcal{P}$. □

More specifically, (1) under the monotonic condition, the PIE program ρ guarantees to terminate under GRAPE and better yet, (2) it converges at correct answer $Q(G)$ for all queries $Q \in \mathcal{Q}$ and all graphs G as long as the sequential algorithms PEval, IncEval and Assemble of ρ are correct for $\mathcal{Q}$. In other words, condition (a) guarantees termination of a PIE program under GRAPE, and conditions (a) and (b) put together guarantee the correctness of a PIE program under GRAPE.

Proof. We show the correctness of Theorem 4.1 by analyzing the computations of a PIE program. Consider any run of a PIE algorithm depicted in Fig. 5. Observe the following.

(1) Termination. Under the monotonic condition of Theorem 4.1, we have that $\ldots \leq_{p_x} R_i^{r+1} \leq_{p_x} \ldots \leq_{p_x} R_i^1 \leq_{p_x} R_i^0$ for all $i \in [1, m]$. Since $R_i^0, R_i^1, \ldots, R_i^{r+1}, \ldots$ are from a finite domain, we know that there exists a number n such that $R_i^{t+1} = R_i^t$ for all $i \in [1, m]$ and $t \geq n$. That is, ρ terminates.

(2) Correctness. From the argument above it follows that ρ must terminate at some superstep r_0. Hence, by condition (b), Assemble$(R_1^{r_0}, \ldots, R_m^{r_0}) = Q(G)$, *i.e.,* ρ computes $Q(G)$. □

Remark. Observe the following.

(1) The fixpoint computation model does not reduce the expressive power of GRAPE. Indeed, (a) fixpoint computation has sufficient expressive power; many data mining and machine learning algorithms can be modeled as fixpoint computations [60, 69]. (b) We can conduct any computation in PEval and IncEval, and hence by GRAPE. We will see a formal characterization in Section 4.2.

(2) The monotonic condition is a sufficient condition for GRAPE computations to converge, but

it is not a necessary condition. Indeed, there has been a large body of work on convergence, *e.g.,* [35, 40, 60, 69], from which other characterizations can be deduced.

(3) It does not mean that only algorithms satisfying the monotonic condition can be parallelized in GRAPE. As will be shown by Theorem 4.2, any MapReduce algorithm can be migrated to GRAPE without extra complexity, and not all MapReduce algorithms are monotonic. The monotonicity is just a sufficient condition under which users do not have to worry about convergence.

4.2 The Expressivity of GRAPE

We next show that the simple parallel model of GRAPE does not imply degradation in the expressivity. As a result, GRAPE can readily switch to other parallel models without extra complexity.

Following [66], 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 costs. 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 [22], BSP [65] and PRAM [66]. Note that GRAPE parallelization is modeled as a simultaneous fixpoint computation. Moreover, GRAPE is a BSP system under the the following constraints: (1) in each round of computation, GRAPE runs the same function PEval or IncEval, while other parallel systems may run different user-defined functions in different rounds (*e.g.,* MapReduce); and (2) GRAPE only allows the status variables of the same vertex in different fragments to be exchanged, while there is no such restriction in some other parallel systems. We show that despite these restrictions, GRAPE does not degrade in expressive power, *i.e.,* it is as powerful as MapReduce, BSP and PRAM.

As a consequence of the result, all algorithms developed for graph systems based on these models can be migrated to GRAPE without increasing complexity bounds, including Pregel [50], GraphX [34], Giraph++ [63] and Blogel [71]. The result below is stronger than its counterpart in [31] in that it does not use key-value pairs (messages) in the simulation (see electronic appendix for proof).

Theorem 4.2 [Simulation Theorem]: *(1) All BSP algorithms with n workers in k supersteps can be optimally simulated on* GRAPE *with n workers in k supersteps;*

(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.* □

Remark. (1) Theorem 4.2 aims to show the expressive power of GRAPE, *e.g.,* all MapReduce algorithms can be migrated to GRAPE without increasing the complexity bounds. Nonetheless, it is possible that some simulated applications are not efficient in practice, due to a possible large constant in the simulation complexity $O(C)$ (see the proof of Theorem 4.2 in the electronic appendix).

(2) As indicated by Theorem 4.2, all parallel algorithms for MapReduce, BSP and PRAM are also supported by GRAPE. Moreover, those graph computations that have effective (*e.g.,* bounded) incremental algorithms can be accelerated by GRAPE.

(3) Compared with the vertex-centric model, the ability to run sequential algorithms over an entire fragment has several benefits. One of them is that it can reduce the number of supersteps, as demonstrated by SSSP. This is because within the fragment each worker can do some computation that would have required extra supersteps in a vertex-centric system like Pregel. This is analogous

to running multiple “local-supersteps”' in a worker before running a global superstep. Similarly it can reduce the communication since no message passing is needed within a fragment, and this happens also because of the reduction in supersteps. Finally, because the sequential algorithms have access to the entire fragment, existing sequential algorithms can be executed, and optimization techniques that are developed for the sequential algorithms are inherited in the parallel setting. Hence, GRAPE can speed up parallel computations and achieve better performance by conducting efficient fragment-level local computations, without incurring excessive communication costs.

(4) However, for algorithms that make only one or very few fragments “active” at a time, GRAPE may not speed up their parallel computations. These include “local” queries to find neighbors of a given node, or kNN (k nearest neighbors) queries with very small constant k. The evaluation of such queries is restricted to a small subgraph localized by the given node. Such a localized subgraph may be entirely contained in one fragment at one worker, and hence may not fully enjoy parallel processing unless we allow a fine-grained parallelization within the fragment by *e.g.,* using parallelized IncEval or partitioning the fragment into multiple small virtual fragments. Besides, GRAPE may not make P-complete problems such as Depth First Search (DFS) more efficient than other parallel platforms; these algorithms are inherently difficult to parallelize.

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 graph pattern matching (defined in terms of graph simulation and subgraph isomorphism), graph connectivity and collaborative filtering as examples (Sections 5.1–5.3, respectively).

We adopt edge-cut [12, 18] in this section unless stated otherwise. Under vertex-cut [47] and other graph partition strategies, PIE programs can be developed similarly.

5.1 Graph Pattern Matching

We start with graph pattern matching, which is commonly used in social media marketing [30], social network analysis [25] and knowledge base expansion [23], among other things.

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 $G = (V, E, L)$ *matches* a pattern $Q = (V_Q, E_Q, L_Q)$ via *graph simulation* if there exists a binary relation $R \subseteq V_Q \times V$ such that

(a) for each query node $u \in V_Q$, there exists a node $v \in V$ such that $(u, v) \in R$, and

(b) for each pair $(u, v) \in R$, (i) $L_Q(u) = L(v)$, and (ii) for each query edge (u, u') in E_Q, there exists an edge (v, v') in graph G such that $(u', v') \in R$.

For $(u, v) \in R$, we refer to v as a *match* of u. It is known that if G matches Q, then there exists a *unique maximum* relation [39], referred to as $Q(G)$. If G does not match Q, then $Q(G)$ is the empty set. Moreover, $Q(G)$ can be computed in $O((|V_Q| + |E_Q|)(|V| + |E|))$ time [25, 39].

Graph pattern matching via graph simulation is stated as follows.

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

We next show how GRAPE parallelizes graph simulation.

<u>*(1)* PEval</u>. GRAPE takes the sequential algorithm of [39] 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.

```
Input: $Q = (V_Q, E_Q, L_Q)$, $F_i = (V_i, E_i, L_i)$ (equally for $G$).
Output: Maximum match relation sim.

Message preamble: /*candidate set $C_i$ is $F_i.I$*/
  for each node $u$ in $V_Q$ and $v$ in $V_i$ a Boolean $x_{(u,v)}$ := true;

/*sequential algorithm for graph simulation (pseudo-code)*/
1.  $V_i := (V_i \setminus F_i.I') \cup F_i.O$;
2.  $E_i := E_i \setminus \{(u,v) \in E_i \mid u \in F_i.I' \wedge v \in F_i.I\}$;
3.  for each $u \in V_Q$ do  /* Initialization */
4.      if post($u$) = $\emptyset$ then
5.          sim($u$) := $\{v \in V_i \mid L_Q(u) = L_i(v)\}$;
6.          else sim($u$) := $\{v \in V_i \mid L_Q(u) = L_i(v) \wedge$ post($v$) $\neq \emptyset\}$;
7.      remove($u$) := pre($V_i$) \ pre(sim($u$));
8.      for each $v \in V_i$ \ sim($u$) do $x_{(u,v)}$ := false;
9.  while there exists $u \in V_Q$ such that remove($u$) $\neq \emptyset$ do
10.     for each $u' \in$ pre($u$) do  /* Refinement */
11.       for each $w \in$ remove($u$) do
12.         if $w \in$ sim($u'$) then
13.           sim($u'$) := sim($u'$) \ $\{w\}$; $x_{(u',w)}$ := false;
14.           for each $w' \in$ pre($w$) do
15.               if post($w'$) $\cap$ sim($u'$) = $\emptyset$ then
16.                 remove($u'$) := remove($u'$) $\cup$ $\{w'\}$;
17.     remove($u$) := $\emptyset$;
18. $Q(F_i)$ := sim;

Message segment: $M_i := \{x_{(u,v)} \mid u \in V_Q \wedge v \in F_i.I\}$;
                 aggregateMsg = min ($x_{(u,v)}$);
```

Fig. 6. PEval for graph simulation in GRAPE

Before giving the details of PEval, we first review the algorithm in [39]. The simulation algorithm [39] computes the match set sim(u) for each query node u via least fixpoint computation. The initial match set sim(u) contains all possible candidate matches of u. These match sets are then iteratively refined by removing non-matching nodes. The process stops when a fixpoint is reached.

As shown in Figure 6, the main body of PEval (lines 3-17) is almost identical to the simulation algorithm of [39], except the underlined parts to preprocess fragments. More specifically, PEval first preprocess each fragment F_i by removing incoming edges and their associated "foreign nodes", and by including nodes to which there exists an outgoing edge from F_i (lines 1-2). Such preprocessing is conducted to comply with the semantics of simulation relations. More specifically, the match status of a data node v, *i.e.,* whether v matches some query node u, is determined by the complete match status of *all* v's outgoing neighbors. This also implies that the match status of v is propagated and updated via the reverse direction of edges linked to v. The preprocessing yields fragment F_i such that (a) for each node v in $V_i \setminus F_i.O$, the outgoing edges from v are all included in E_i; and (b) for each node v' in $F_i.O$, there exists no outgoing edge from v' present in E_i. As a result, the match status of $V_i \setminus F_i.O$, *i.e.,* of the nodes *owned* by F_i, is computed and updated in F_i at worker P_i; and the match status of nodes in $F_i.O$ is computed and updated in other fragments.

For each node $u \in V_Q$, PEval starts with a set sim(u) of candidate matches v in F_i (lines 3-8), and iteratively removes from sim(u) those nodes that violate the simulation condition (lines 9-17). It uses post(v) and pre(v) to keep track of successors and predecessors of node v, respectively (see [39] for details). It refines sim(u) for all $u \in V_Q$. The partial result $Q(F_i)$ is designated (line 18).

```
Input: $Q = (V_Q, E_Q, L_Q)$, $F_i = (V_i, E_i, L_i)$, maximum match
relation sim for $Q$ and $F_i$; message $M_i$ (batch update).
Output: maximum match relation sim for $P$ and $F_i \oplus M_i$.

/* incremental graph simulation (pseudo-code)*/
1.  stack simset := $\emptyset$;
2.  for each $x^{\text{in}}_{(u,v)} \in M_i$ do simset.push($u, v$);  /*$x^{\text{in}}_{(u,v)}$ = false*/
3.  while simset is not empty do
4.    ($u$, $v$) := simset.pop(); sim($u$) := sim($u$) \ {$v$}; $x_{(u,v)}$ := false;
5.    for each $v' \in$ pre($v$) and $u' \in V_Q$ that $v' \in$ sim($u'$) do
6.      if $(u', u) \in E_Q$ and sim($u$) $\cap$ post($v'$) = $\emptyset$ then
7.        simset.push($u', v'$);
8.  $Q(F_i)$ := sim;

Message segment:
  $M_i := \{x_{(u,v)} \mid u \in V_Q, v \in F_i.I, x_{(u,v)}$ changed to false};
```

Fig. 7. IncEval for graph simulation

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. That is, the updated match status is propagated via the reverse direction of edges.

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 $G_{\mathcal{P}}$, groups them into messages M_j, and sends M_j to P_j.

(2) IncEval is the sequential incremental simulation algorithm of [28] in response to edge deletions. The changes to sim(u) are "equivalent to" removing some nodes from sim(u), which can be also seen as the results of removing some relevant edges. Thus propagating the changes of these nodes can be done by propagating the changes of deleted edges. Hence we can use the algorithm of [28] for edge deletions. Note that we just make use of the algorithm for edge deletions as IncEval to process changes to $x_{(u,v)}$, but IncEval does not have to handle generic edge deletions in the graph.

As shown in Figure 7, if status variable $x_{(u,v)}$ is changed to false by message M_i, it is treated as deleting "cross edges" to $v \in F_i.O$. Using a stack (line 1), it starts with changed status variables in M_i, propagates the changes to affected area, and removes from sim those matches that become invalid (lines 3-7; see [28] for more details). The partial result is now the revised sim relation (line 8). At the end of the process, IncEval sends to coordinator P_0 those values of the status variables in $C_i.\bar{x}$ that have been set false in the process, along the same lines as how PEval does it.

As shown in [28], IncEval is *semi-bounded*: 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 $|F_i|$. This reduces the cost of iterative computation of graph simulation (the **while** and **for** loops).

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

(4) The correctness of the GRAPE parallelization is warranted by Theorem 4.1, and the monotonic updates to $C_i.\bar{x}$. Indeed, $x_{(u,v)}$ is initially true for each border node v, and is changed at most once to false, taking the order false $\prec$ true. Furthermore, $x_{(u,v)}$ denotes whether v matches u, as warranted by the correctness of the sequential algorithms [28, 39] (PEval and IncEval).

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. More specifically, a *match* of Q in

```
Input: $Q = (V_Q, E_Q, L_Q)$, $F_i = (V_i, E_i, L_i)$.
Output: a local annotated version of $N_{d_Q}(s)$ in $F_i$ for each $s \in F_i.I \cup F_i.O'$.

Message preamble:
  an integer variable dist$(s, t)$ initialized as $\infty$ for each pair $s, t \in V_i$;

for each $s \in F_i.I \cup F_i.O'$ do
    dist$(s, s) := 0$; $V(s) := \{s\}$;
    queue Que := $\emptyset$; Enqueue(Que, s);
    while Que is not empty do
        $u :=$ Dequeue(Que);
        if dist$(s, u) < d_Q$ then
            for each $v$ linked with $u$ do
                if dist$(s, v) := \infty$ then
                    dist$(s, v) :=$ dist$(s, u) + 1$; $V(s) := V(s) \cup \{v\}$;
                    Enqueue(Que, $v$);
    $E(s) := \{(u, v) \mid (u, v) \in F_i, u, v \in V(s)\}$;
    $L(s) := \{$dist$(s, v) \mid v \in V(s)\}$;
$Q(F_i) :=$ $\{(s, V(s), E(s), L(s)) \mid s \in F_i.I \cup F_i.O')\}$;

Message segment: $M_i := \{(s, V(s), E(s), L(s)) \mid s \in F_i.I \cup F_i.O'\}$;
                 aggregateMsg = Expand$(s, V(s), E(s), L(s))$;

Procedure Expand
Input: a source $s \in F_i.I \cup F_i.O'$, $(u, V(u), E(u), L(u))$ for each $u \in \mathcal{F}.I \cup \mathcal{F}.O'$.
Output: the $d_Q$-neighbor $N_{d_Q}(s)$ of $s$ in $G$.

queue Que := $\emptyset$; $V_0 := V(s)$;
for each $v \in V(s)$ do
    if dist$(s, v) < d_Q$ and $v \in F_i.O \cup F_i.I'$ then
        Enqueue(Que, $(v, d_Q -$ dist$(s, v))$);
while Que is not empty do
    $(u, k) :=$ Dequeue(Que);
    use $G_{\mathcal{P}}$ to find $j$ such that $u \in F_j.I \cup F_j.O'$;
    for each $v \in V(u)$ do
        if dist$(u, v) \leq k$ then
            $V_0 := V_0 \cup \{v\}$;
            if $v \in F_j.O \cup F_j.I'$ and dist$(u, v) < k$ then
                Enqueue(Que, $(v, k -$ dist$(u, v))$);
$E_0 := \{(u, v) \mid u, v \in V_0, (u, v) \in E(w)$ for some $w \in \mathcal{F}.I \cup \mathcal{F}.O'\}$;
$N_{d_Q}(s) := (V_0, E_0)$;
```

Fig. 8. PEval for subgraph isomorphism

G is a subgraph $G' = (V', E', L', F'_A)$ of G such that there exists a *bijective function h* from V_Q to V', where (1) for each node $u \in V_Q$, $L_Q(u) = L'(h(u))$; and (2) $e = (u, u')$ is an edge in Q if and only if $e' = (h(u), h(u'))$ is an edge in G' and $L_Q(e) = L'(e')$.

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 $\text{Turbo}_{\text{ISO}}$, the sequential algorithm of [37] for subgraph isomorphism. It has *two* supersteps, one for PEval and the other for IncEval, outlined as follows.

(1) PEval identifies update parameters $C_i.\bar{x}$ at each fragment F_i. It declares an integer variable dist(s, t) as the status variable for each pair of nodes s and t in F_i, to record their distance in

```
Input: Pattern $Q = (V_Q, E_Q, L_Q)$, fragment $F_i = (V_i, E_i, L_i)$,
       message $M_i$ consisting of $N_{d_Q}(v)$ for each $v \in F_i.O \cup F_i.I'$.
Output: $Q(F_i \oplus M_i)$, i.e., the set of all matches of $Q$ in $F_i \oplus M_i$.

$F_i := F_i \oplus M_i$; /* extend $F_i$ with the nodes and edges in $M_i$ */
$Q(F_i) :=$ Turbo_ISO$(Q, F_i)$;
```

Fig. 9. IncEval for subgraph isomorphism

fragment F_i. It computes the d_Q-neighbor $N_{d_Q}(s)$ of each border node in $s \in F_i.I \cup F_i.O'$. Here d_Q is the diameter of pattern Q, *i.e.,* the length of the shortest path between any two nodes in Q when Q is treated as an undirected graph; and $N_d(v)$ is the subgraph of G induced by the nodes within d hops of v. At each F_i, $C_i.\bar{x}$ consists of $N_{d_Q}(s)$ for all $s \in F_i.I \cup F_i.O'$.

Intuitively, upon receiving update parameters $C_i.\bar{x}$ from all workers, coordinator P_0 completes the d_Q-neighbor of each border node s in the entire graph G, and sends the d_Q-neighbor to workers where s resides, to compensate information loss caused by fragmentation of graph G. After this step, one can directly apply Turbo$_{\text{ISO}}$ to each expanded fragment in parallel.

More specifically, PEval computes $C_i.\bar{x}$ at fragment F_i as shown in Figure 8. PEval performs a standard BFS traversal (Breath-First Search) from each border node s in $F_i.I \cup F_i.O'$ to identify (a) a set $V(s)$ of nodes that are reachable from s in d_Q hops in F_i (lines 2-10), and (b) a set $E(s)$ of edges that are associated with nodes in $V(s)$ (line 11). Here fragment F_i is treated as an undirected graph in the BFS traversal, ignoring the orientations of the edges (line 7). PEval annotates each node v in $V(s)$ with $\text{dist}(s, t)$ from s (lines 8-9). These compose a local (annotated) d_Q-neighbor $N_{d_Q}(s)$ in fragment F_i. To simplify the discussion, PEval sends these local $N_{d_Q}(s)$'s to coordinator P_0 (line 13). In practice, the union of all these d_Q-neighbors is sent to P_0 in a single message. The size of such a message is bounded by the size $|G|$ of graph G.

Upon receiving the local versions of $N_{d_Q}(s)$ for all s in $\mathcal{F}.I \cup \mathcal{F}.O'$, coordinator P_0 expands each of them to the d_Q-neighbor $N_{d_Q}(s)$ in the entire graph G. This is specified by procedure Expand as **aggregateMsg**, which performs a BFS-like traversal on the data received and combines necessary nodes and edges, by making use of fragmentation graph $G_{\mathcal{P}}$ (see Section 2) and the annotations associated with the nodes. A message M_i is composed and sent to worker P_i for $i \in [1, m]$, including all the nodes and edges in the d_Q-neighbor of s in the entire graph G, for each $s \in F_i.O \cup F_i.I'$.

(2) IncEval is the sequential algorithm Turbo$_{\text{ISO}}$ [37]. Given a pattern Q and a graph G, Turbo$_{\text{ISO}}$ finds all isomorphic matches of Q in G as follows. (1) It first picks a start vertex from query Q, and rewrites Q into a tree Q' by performing BFS search. Each node in the tree corresponds to a "neighborhood equivalence class" (NEC), by merging nodes with the same labels and neighborhoods. (2) Next, it explores "candidate regions" of Q', *i.e.,* subgraphs that subsume matches of Q. (3) For each candidate region, it computes an order on the nodes in Q', based on the number of their candidate matches in the region. (4) It then searches matches within the candidate region in this order. During the search, it only combines partial matches of the NECs, instead of inspecting all possible enumerations. (5) At last, it expands matches of the NECs to get exact matches of Q.

As shown in Figure 9, IncEval 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 $F_i.O \cup F_i.I'$ by applying Turbo$_{\text{ISO}}$. 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.

```
Input: $F_i = (V_i, E_i, L_i)$.
Output: $Q(F_i)$ consisting of $v$.cid for each $v \in V_i$.

Message preamble: /* candidate set $C_i$ is $F_i.I$ */
    for each $v \in V_i$, an integer variable $v$.cid initialized as $v$'s id;

1. $CC$ := DFS($F_i$); /* use DFS to find the set of local CCs */
2. for each local component C $\in CC$ do
3.     add a new single root node $v_r$;
4.     $v_r$.cid := min{$v$.cid | $v \in$ C};
5.     for each node $v \in$ C do
6.         link $v$ to $v_r$;  $v$.root := $v_r$;  $v$.cid := $v_r$.cid;
7. $Q(F_i)$ := {$v.cid$ | $v \in V_i$};

Message segment: $M_i$ := {$v$.cid | $v \in F_i.I$};
                 aggregateMsg = min($v$.cid);
```

Fig. 10. PEval for CC

(4) The correctness of the process is assured by Turbo$_{\text{ISO}}$ 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, 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.

The CC problem is stated as follows, and is known to be in $O(|G|)$ time [13].

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

GRAPE parallelizes CC as follows. 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. As shown in Figure 10, PEval first uses a standard sequential traversal DFS (Depth-First Search) to compute the local connected components of F_i (line 1). For each local component C, (a) PEval creates a "root" node v_r carrying the minimum node id in C as v_r.cid (lines 3-4), and (b) links all the nodes in C to v_r, and sets their cid as v_r.cid (lines 5-6). These can be completed in one pass of the edges of F_i via DFS. At the end of process, PEval sends $\{v.\text{cid} \mid v \in F_i.I\}$ to coordinator P_0. In other words, the set consists of the update parameters at fragment F_i.

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 each fragment F_i upon receiving M_i, in parallel, as shown in Figure 11. Observe that message M_i sent to P_i consists of v.cid with updated (smaller) values. For each v.cid in M_i, IncEval finds the root v_r of v (line 3), and updates v_r.cid

```
Input: $F_i = (V_i, E_i, L_i)$, partial result $Q(F_i)$, message $M_i$ (grouped cid).
Output: $Q(F_i \oplus M_i)$.

/*incremental connected component (pseudo-code) */
1.  $\Delta := \emptyset$;
2.  for each $v$.cid $\in M_i$ do   /* $v \in F_i.I$ */
3.      $v_r := v$.root;
4.      if $v$.cid $< v_r$.cid then
5.          $v_r$.cid := $v$.cid;  $\Delta := \Delta \cup \{v_r\}$;
6.  for each $v_r \in \Delta$ do   /*propagate the change*/
7.      for each $v' \in V_i$ linked to $v_r$ do
8.          $v'$.cid := $v_r$.cid;
9.  $Q(F_i) := \{v.\text{cid} \mid v \in V_i\}$;

Message segment: $M_i := \{v.\text{cid} \mid v \in F_i.I, v.\text{cid decreased}\}$;
```

Fig. 11. IncEval for CC

to the minimal one (lines 4-5). IncEval then propagates the changes from every updated root node v_r to all nodes linked to v_r by changing their cids to v_r.cid (lines 6-8). At the end of the process, IncEval sends to coordinator P_0 the updated cids of nodes in $F_i.I$ just like in PEval.

One can verify that the incremental algorithm IncEval is *bounded*: it takes $O(|M_i|)$ time to identify the root nodes, and $O(|\mathsf{AFF}|)$ time to update cids by following the direct links from the roots, where AFF consists of only those nodes with their cid *changed*. Hence, it avoids redundant local traversal.

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

(4) Correctness. The process terminates as the cids of the nodes are monotonically decreasing by the definition of **aggregateMsg**, 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) [48], a method commonly used for inferring user-product rates in social recommendation. It takes as input a bipartite graph G that includes two types of nodes, namely, 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 an (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 $\{e \in E \mid r(e) \neq \emptyset\}$, *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 min $\sum_{((u,p) \in E_T)} (r(u, p) - u.f^T * p.f)^2 + \lambda(\|u.f\|^2 + \|p.f\|^2)$.

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 [48], which iteratively (1) computes a prediction 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 $\epsilon(f, E_T)$.

The SGD algorithm [48] is inherently sequential. To parallelize it, a nice idea has been proposed by DSGD [33], based on a partition of the dataset such that at each round of computation, different workers can process disjoint datasets independently without conflicts. More specifically, it partitions

```
Input: $F_i = (V_i, E_i, L_i)$.
Output: $Q(F_i)$ consisting of $v.f$ for each $v \in V_i$.

Message preamble: /* candidate set $C_i$ is $F_i.I$ */
  a variable $x.v = (v.f, t)$ for each $v \in V_i$;

$B_i := \{e \in E(i,i) \mid r(e) \neq \emptyset\}$;
Run SGD on the training set $B_i$;
$t := t + 1$;
$Q(F_i) := \{v.f \mid v \in V_i\}$;

Message segment: $M_i := \{(v.f, t) \mid v \in F_i.I, v.f$ changed$\}$;

Procedure SGD
Input: A training set $B$.
Output: Revised training set.

/* SGD on the training set $B$ (pseudo-code)*/
for each $k \in \{1, 2, \ldots, |B|\}$ do
    Pick an edge $(u,p)$ from $B$ uniformly at random;
    $\epsilon(u,p) := r(u,p) - u.f^T * p.f$;
    $u.f' := u.f + \gamma(\epsilon(u,p) * p.f - \lambda * u.f)$;
    $p.f := p.f + \gamma(\epsilon(u,p) * u.f - \lambda * p.f)$;
    $u.f := u.f'$;
```

Fig. 12. PEval for CF

the user set U into m disjoint subsets $U(1)$, $U(2)$, $\ldots$, $U(m)$, and similarly, the product set P into disjoint $P(1)$, $P(2)$,$\ldots$, $P(m)$ such that $U = \bigcup_{i=1}^{m} U(i)$ and $P = \bigcup_{j=1}^{m} P(j)$, for a constant m. Correspondingly the training set E is divided into m^2 *blocks*, such that each $1 \leq i, j \leq m$, a block $E(i,j)$ identified by a pair (i,j) is the subset of E induced by $U(i)$ and $P(j)$. Clearly $E = \bigcup_{1 \leq i,j \leq m} E(i,j)$. Two blocks $E(i,j)$ and $E(i',j')$ are independent if $i \neq i'$ and $j \neq j'$. DSGD parallelizes SGD by utilizing the property that the factor vectors of independent blocks can be updated in parallel without conflicts.

Adopting the partition strategy of DSGD, GRAPE parallelizes the sequential SGD algorithm such that different workers can run SGD on different fragments of a graph G in parallel, without conflicts. GRAPE partitions G by a vertex-cut strategy and distributes the m^2 blocks into m different fragments. More specifically, it defines $F_i = (V_i, E_i, L_i)$, where $V_i = U(i) \cup \bigcup_{j=1}^{m} P(j)$ and $E_i = \bigcup_{j=1}^{m} E(i,j)$, *i.e.,* P is shared by all fragments while U is partitioned across different workers.

(1) PEval 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 is an integer (initially 0) that bookkeeps a timestamp at which $v.f$ is lastly updated. The candidate set C_i the border nodes set $F_i.I$.

As shown in Figure 12, PEval essentially runs the sequential SGD algorithm of [48] on the training block $E(i,i)$ as follows. Each time it picks an edge (u,p) from the training set uniformly at random and computes the prediction error $\epsilon(u,p)$. It updates local factor vectors by a magnitude proportional to γ in the opposite direction of the gradient as:

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

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

```
Input: $F_i = (V_i, E_i, L_i)$, partial result $Q(F_i)$,
       message $M_i$ including updated factor vectors and
       a pair $(i, p_i)$ indicates the training block.
Output: $Q(F_i \oplus M_i)$.

1.  for each $v^{\text{in}}.f \in M_i$ do
2.      $v.f := v^{\text{in}}.f$;
3.  $B_i := \{e \in E(i, p_i) \mid r(e) \neq \emptyset\}$;
4.  Run SGD on the training set $B_i$;
5.  $t := t + 1$;
6.  $Q(F_i) := \{v.f \mid v \in V_i\}$;

Message segment:
    $M_i := \{(v.f, t) \mid v \in F_i.I,\ v.f$ changed $\}$;
```

Fig. 13. IncEval for CF

By the partition strategy, the training blocks $E(1,1)$, $E(2,2)$, …, $E(d,d)$ on different fragments line on the diagonal of the rating matrix, and are independent of each other. At the end of its process, PEval sends message M_i that consists of updated $v.x$ for nodes $v \in C_i$ to coordinator P_0.

At coordinator P_0, GRAPE maintains $v.x = (v.f, t)$ for all border nodes $v \in \mathcal{F}.I = \mathcal{F}.O$. Upon receiving updated values $(v.f', t')$ with $t' > t$, it changes $v.f$ to $v.f'$, *i.e.,* it defines **aggregateMsg** as max on timestamps. This is well defined since different workers process independent blocks. GRAPE then groups the updated vectors into messages M_j, and sends M_j to P_j as usual. That is, GRAPE passes the latest updates to factor vectors to workers.

In addition, coordinator P_0 selects m independent blocks to be processed in the next round. To do this P_0 simply picks a permutation $p_1 p_2 \ldots p_m$ of $\{1, 2, \ldots, m\}$ following some fixed strategy, *e.g.,* simple cycle scheduling. It sends a pair (j, p_j) along with M_j to P_j. By the partition strategy, block $E(j, p_j)$ belongs to F_j and $E(j, p_j)$ is independent of $E(i, p_i)$ if $j \neq i$.

(2) IncEval iteratively updates the factor vectors of independent blocks. As shown in Figure 13, IncEval first updates the factor vectors with the latest changes (lines 1-2). It then extracts the training set B_i for the current round based on the block identifier assigned by P_0 (line 3), and runs the sequential SGD algorithm [48] on B_i just like PEval (line 4). Since the training sets B_1, B_2, …, B_d are extracted from the independent blocks $E(1, p_1)$, $E(2, p_2)$, …, $E(d, p_d)$, they can be processed in parallel without conflict. At the end of the process, it sends the updated vectors in C_i like PEval.

(3) Assemble simply takes the union of all the factor vectors of nodes from all the workers.

(4) Correctness. Observe that a permutation $p_1 p_2 \ldots p_m$ of $\{1, 2, \ldots, m\}$ corresponds to a m-monomial stratum of DSGD [33]. The permutation in each round is picked according to a stratum selection strategy. It is known that the strategy guarantees the convergence of DSGD (see [33] for more details). As a result, GRAPE converges and correctly infers CF models by the correctness of DSGD.

6 IMPLEMENTATION OF GRAPE

We next outline an implementation of parallel graph engine GRAPE.

Architecture overview. GRAPE adopts a four-tier architecture depicted in Figure 14.

(1) Its top layer is a user interface. As shown in Figure 1, GRAPE supports interactions with (a) developers who specify and register sequential PEval, IncEval and Assemble as a PIE program for

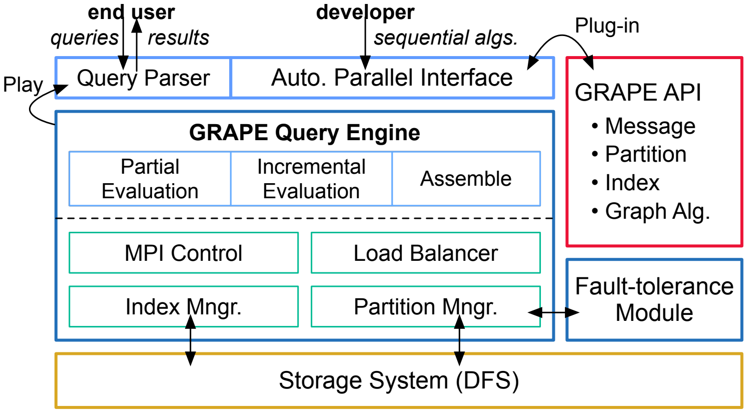


Fig. 14. GRAPE Architecture

a class $\mathcal{Q}$ of graph queries (the plug panel); and (b) end users who make use of 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).

(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 [5], which is also the basis of other parallel graph systems such as GraphLab [49] and Blogel [71]. It generates messages and coordinates messages in synchronization steps using standard MPI primitives.

We remark that at the conceptual level, to simplify the discussion, we adopt a coordinator to aggregate messages (Section 3). In practice, GRAPE implements point-to-point message passing instead: workers exchange messages directly without going through a coordinator, accumulate messages received in a buffer, and the aggregation function is invoked at each worker. It is easy to verify that this implementation and the centralized aggregation with coordinator produce the same results, since the aggregation function is invoked after all messages are received.

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 [44], (b) edge-cut partition [12, 18] and vertex-cut partition [47], (c) 1-D and 2-D partitions [17], 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 [62] that assigns edges to high degree nodes to reduce cross edges. New data partition strategies can also be deployed at GRAPE.

Multi-thread. GRAPE supports multi-threading. At each worker, there are multiple working threads, each acting as a virtual worker and handling one fragment. During computation, the threads at the same worker are maintained in a pool; a main thread at the worker takes the responsibility of assigning fragments and workload to idle threads in the pool. At the end of each round of the computation, the main thread generates messages and communicates with peer workers. The main thread is able to process messages in a buffer even when its working threads are still computing. This allows workers to overlap computation and communication, and reduce response time.

Graph-level optimization. In contrast to prior graph systems, GRAPE supports data-partitioned parallelism by parallelizing the runs of sequential algorithms. Since fragments of a graph are graphs themselves, all optimization strategies developed for sequential (batch and incremental) algorithms can be readily used by GRAPE, to improve the performance of PEval and IncEval over graph fragments. As examples, below we outline some of the graph-level optimization strategies.

(1) Indexing. Any indexing structure effective for sequential algorithm can be computed offline and directly used to optimize PEval, IncEval and Assemble. GRAPE can support indices including (1) 2-hop index [21] for reachability queries; and (2) neighborhood-index [45] for candidate filtering in graph pattern matching. Moreover, new indices can be incorporated into GRAPE API library.

(2) Compression. Another strategy is *query preserving compression* [26] 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 [26], this compression scheme is effective for graph pattern matching and graph traversal, among other things.

(3) Dynamic grouping. GRAPE dynamically groups 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 passing in each synchronization step.

To the best of our knowledge, many of these optimization strategies are not supported by the state-of-the-art vertex-centric and block-centric graph query systems. For instance, indexing and query-preserving compression for sequential algorithms do not carry over to vertex-centric programs, and block-centric programming essentially treats blocks as vertices rather than graphs.

Fault tolerance. GRAPE employs an arbitrator mechanism to recover from both worker failures and coordinator failures (*a.k.a.* single-point failures). More specifically, 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 parallel computation. It is also possible for GRAPE to adopt the optimistic recovery mechanism introduced in [60] for general fixpoint paradigm [15].

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 4.1 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.

7 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we next empirically evaluate GRAPE, for its (1) efficiency and scalability, (2) communication costs, (3) effectiveness of incremental steps, and (4) compatibility with optimization techniques developed for sequential graph algorithms. We used real-life graphs larger than those that [31] experimented with. We evaluated the performance of GRAPE compared with Giraph (a open-source version of Pregel), GraphLab, and Blogel (the fastest block-centric system we are aware of). We compared GRAPE with the prior graph systems by parallelizing existing sequential algorithms with a preliminary implementation of GRAPE [9].

Experimental setting. We used five real-life graphs of different types, including

(1) movieLens [3], a dense recommendation network (bipartite graph) that has 20 million movie ratings (as weighted edges) between 138000 users and 27000 movies;
(2) UKWeb [1], a Web graph with 133 million nodes and 5 billion edges;
(3) DBpedia [7], a knowledge base with 5 million entities and 54 million edges, with 411 labels;
(4) Friendster [4], a social network with 65 million users and 1.8 billion links; and
(5) traffic [2], a road network with 23 million nodes (locations) and 58 million edges.

To make use of unlabeled Friendster for Sim and SubIso, we assigned up to 100 random labels to nodes. We also randomly assigned weights to UKWeb, traffic and Friendster for testing SSSP.

Synthetic graphs. To evaluate the scalability of GRAPE (Exp-1 and Exp-2), we also developed a generator to produce synthetic graphs $G = (V, E, L)$ controlled by the numbers of nodes $|V|$ (up to 250 million) and edges $|E|$ (up to 2.5 billion), with L drawn from an alphabet $\mathcal{L}$ of 100 labels.

Partitioning and Loading. We used XtraPuLP [61] as the default graph partition strategy. In theory, GRAPE works regardless of what partitioning strategy is used, and guarantees to converge under the conditions given in Theorem 4.1. In practice, different strategies may yield partitions with various degrees of skewness and stragglers, which have an impact on the performance of GRAPE. Here we picked XtraPuLP, which is widely used in practice. On Friendster, for example, XtraPuLP took about 16 minutes, and our computations took at most 62 seconds. However, graph partitioning is performed once offline. Afterwards various queries are answered online on the same partition. The partitioning costs for traffic, UKWeb, DBpedia and movieLens are 6.0, 598.1, 32.2, 4.3 seconds, respectively.

GRAPE loads graph data from a distributed file system by each worker simultaneously. It takes about 20 minutes to import Friendster by 4 workers at the first time (16s, 24m, 44s, 10s for traffic, UKWeb, DBpedia, movieLens, respectively). After the first loading, the graph is "serialized" to the storage in a compact format, which largely reduces the loading time to 40s (2s, 86s, 4s, 2s for traffic, UKWeb, DBpedia, movieLens, respectively) for reloading afterwards when necessary.

It should be remarked that GRAPE is able to load a graph G once and process query workload (*i.e.,* a set of queries) posed on G, without reloading G. In contrast, GraphLab, Giraph and Blogel require the graph to be reloaded each time a single query is issued, and loading is costly over large graphs. In favor of these systems, we exclude the loading cost when reporting the experimental results.

Queries. We randomly generated the following queries for SSSP, Sim and SubIso. (a) We sampled 10 source nodes in each graph used, 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 experimented with.

Algorithms. We implemented the PIE programs (PEval, IncEval and Assemble) for the query classes given in Sections 3 and 5, namely, SSSP, Sim, SubIso, CC and CF, which are registered in the API library of GRAPE. 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 these query classes for Giraph, GraphLab and Blogel. We used the "default" code provided by the systems when available, and made our best efforts to develop "optimal" algorithms otherwise. We also used the "default" graph partition algorithms provided by these systems, *i.e.,* hash partitioning for GraphLab and Giraph, and Voronoi partitioning for Blogel. We implemented synchronized algorithms for both GraphLab and Giraph for the ease of comparison. As observed by [40, 41, 68], neither asynchronous model nor synchronous model outperform the other for different algorithms, input graphs and cluster scales. We expect the observed relative performance trends to hold on other similar graph systems.

We deployed the systems on a cluster of up to 12 machines, each with 16 processors (Intel Xeon 2.2GHz) and 128G memory (thus in total 192 processors). This is the best configuration we could afford. Each experiment was repeated 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluated the efficiency and scalability of GRAPE by varying the number n of processors used, from 64 to 192. For each algorithm, we chose datasets based on its applications in the real world, to demonstrate meaningful computations. For SSSP and CC, we experimented with real-life graphs UKWeb, traffic and Friendster. For Sim and SubIso, we used Friendster and DBpedia. We used movieLens for CF as its application in movie recommendation.

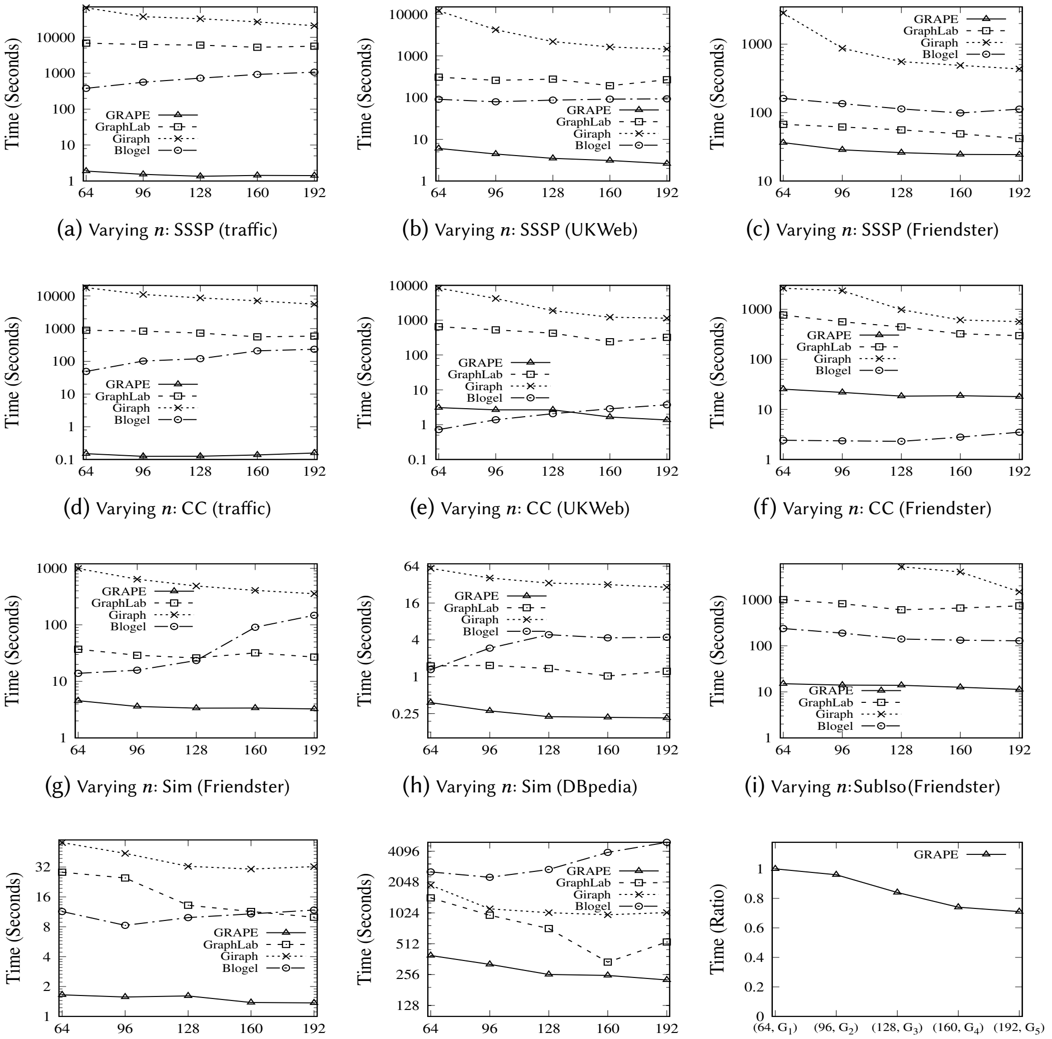
(1) SSSP. Figures 15a-15c report the performance of the systems for SSSP over Friendster, UKWeb and traffic, respectively. We report the average over 10 SSSP queries on each graph. The results on other graphs are consistent (not shown). From the results we can see the following.

(a) GRAPE outperforms Giraph, GraphLab and Blogel by 14842, 3992 and 756 times, respectively, over traffic with 192 processors (Figure 15a). In the same setting, it is 556, 102 and 36 times faster over UKWeb (Figure 15b), and 18, 1.7 and 4.6 times faster over Friendster (Figure 15c). These results demonstrate that by simply parallelizing sequential algorithms without further optimization, GRAPE already outperforms the state-of-the-art systems in response time for SSSP.

The improvement of GRAPE over all the systems on traffic is much larger than on Friendster and UKWeb since the traffic graph has a larger diameter. In addition, (i) for Giraph and GraphLab, this is because synchronous vertex-centric algorithms take more supersteps to converge on graphs with large diameters, such as traffic. Using 192 processors, Giraph take 10749 supersteps over traffic and 161 over UKWeb; similarly for GraphLab. In contrast, GRAPE is not vertex-centric and it takes 31 supersteps on traffic and 24 on UKWeb. (ii) Blogel also takes more (1690) supersteps over traffic than over UKWeb (42 supersteps) and Friendster (23 supersteps). It generates more blocks over traffic (with larger diameter) than UKWeb and Friendster. Since Blogel treats blocks as vertices, the benefit of parallelism is degraded with more blocks.

(b) In all cases, GRAPE take less time when n increases. On average, it is 1.4, 2.3 and 1.5 times faster for n from 64 to 192 over traffic, UKWeb and Friendster, respectively. (i) Compared with the results in [31] using less processors, this improvement degrades a bit. This is mainly because the larger number of fragments leads to more communication overhead. On the other hand, such impact is significantly mitigated by IncEval that only ships changed update parameters. (ii) In contrast, Blogel does not demonstrate such consistency in scalability. It takes more time on traffic when n is larger. When n varies from 160 to 192, it also takes longer over Friendster. Its communication cost dominates the parallel cost as n grows, "canceling out" the benefit of parallelism. (iii) GRAPE has scalability comparable to GraphLab over Friendster and scales better over UKWeb and traffic. Giraph has better improvement with larger n, but with constantly higher cost (see (a)) than GRAPE.

(c) GRAPE significantly reduces supersteps. It takes on average 22 supersteps, while Giraph, GraphLab and Blogel take 3647, 3647 and 585 supersteps, respectively. This is because GRAPE runs sequential algorithms over fragmented graphs with cross-fragment communication only



(a) Varying *n*: SSSP (traffic)

(b) Varying *n*: SSSP (UKWeb)

(c) Varying *n*: SSSP (Friendster)

(d) Varying *n*: CC (traffic)

(e) Varying *n*: CC (UKWeb)

(f) Varying *n*: CC (Friendster)

(g) Varying *n*: Sim (Friendster)

(h) Varying *n*: Sim (DBpedia)

(i) Varying *n*:SubIso(Friendster)

(j) Varying *n*: SubIso (DBpedia)

(k) Varying *n*: CF (movieLens)

(l) Scalability of GRAPE (Synthetic)

Fig. 15. Efficiency of GRAPE

when necessary, and moreover, IncEval ships only *changes* to status variables. In contrast, Giraph, GraphLab and Blogel pass vertex-vertex (vertex-block) messages.

(d) SSSP under Blogel runs in VB-model. In each superstep, it first runs V-compute over all vertices to identify "active" vertices, *i.e.,* vertices whose distance value is updated; it then runs B-compute on active vertices within blocks. Compared with pure vertex-centric models, running a sequential algorithm within blocks reduces communication cost. However, its V-compute incurs redundant computations since it runs over all vertices in each superstep. In contrast, GRAPE runs sequential algorithms within partitions and leverages incremental computation to reduce redundant computation and communication cost. In each round, IncEval only runs on affected vertices.

(2) CC. Figures 15d-15e report the performance for CC detection, and tell us the following. (a) Both GRAPE and Blogel substantially outperform Giraph and GraphLab. For instance, when $n = 192$, GRAPE is on average 12094 and 1329 times faster than Giraph and GraphLab, respectively. (b) Blogel is faster than GRAPE in some cases, *e.g.,* 3.5 seconds vs. 17.9 seconds over Friendster when $n = 192$. This is because Blogel embeds the computation of CC in its graph partition phase as precomputation, while this graph partition cost (on average 357 seconds using its built-in Voronoi partition) is *not* included in its response time. In other words, without precomputation, the performance of GRAPE is already comparable to the near "optimal" case reported by Blogel.

CC in GRAPE also works better than the one in Giraph++ [63]. This is because after exchanging messages between blocks, Giraph++ invokes computation on all internal vertices, and a large part of the computation is redundant. In contrast, IncEval of GRAPE processes only those affected vertices, by capitalizing on auxiliary indices that were inherited from sequential algorithms.

(3) Sim. Fixing $|Q| = (6, 10)$, *i.e.,* patterns Q with 6 nodes and 10 edges, we evaluated graph simulation over DBpedia and Friendster. As shown in Figures 15g-15h, (a) GRAPE consistently outperforms Giraph, GraphLab and Blogel over all queries. It is 109, 8.3 and 45.2 times faster over Friendster, and 136.7, 5.8 and 20.8 times faster over DBpedia for 20 queries on average, respectively, when $n = 192$. (b) GRAPE scales better with the number n of processors than the others. (c) GRAPE takes at most 21 supersteps, while Giraph, GraphLab and Blogel take 38, 38 and 40 supersteps, respectively. This empirically validates the convergence guarantee of GRAPE under monotonic status variable updates and its effect on reducing computation and communication costs.

(4) SubIso. Fixing $|Q| = (3, 5)$, we evaluated the performance of the systems for subgraph isomorphism. As shown in Figures 15i-15j over Friendster and DBpedia, respectively, (a) GRAPE is on average 76, 35 and 9 times faster than Giraph, GraphLab and Blogel when $n = 192$. (b) When n varies from 64 to 192, GRAPE is on average 1.3 and 1.2 times faster over Friendster and DBpedia, respectively. This is comparable with GraphLab that is 1.3 and 2.8 times faster, respectively.

(5) CF. For collaborative filtering, we used real-life movieLens [3] with a training set $|E_T| = 90\%|E|$. We compared GRAPE with the built-in SGD-based CF in GraphLab, and with CF implemented for Giraph and Blogel. It should be remarked that CF favors "vertex-centric" programming since each user or product node only needs to exchange data with its neighbors, as indicated by that GraphLab and Giraph outperform Blogel. Nonetheless, as shown in Figure 15k, GRAPE is on average 4.1, 2.6 and 12.4 times faster than Giraph, GraphLab and Blogel, respectively, when the number n of processors varies from 64 to 192. Moreover, GRAPE scales well with n.

(6) Scalability of GRAPE. As observed in [51], the speed-up of a system may degrade over more processors. We thus evaluated the scalability of GRAPE, which measures the ability to keep the same performance when both the size of graph G (denoted as $(|V|, |E|)$) and the number n of processors increase proportionally. We varied n from 64 to 192, and for each n, deployed GRAPE over a synthetic graph. The graph size varies from $(50M, 500M)$ (*i.e.,* 50 million nodes and 500 million edges; denoted as G_1) to $(250M, 2.5B)$ (denoted as G_5), with fixed ratio between edge number and node number and proportional to n . The scalability at *e.g.,* $(128, G_3)$ is the ratio of the time using 64 processors over G_1 to its counterpart using 128 processors over G_3. As shown in Fig. 15l, GRAPE preserves a reasonable scalability (close to linear scalability, the optimal scalability).

We further evaluated the COST [51] of GRAPE, which denotes the hardware configuration (the number of cores) required by GRAPE to outperform a competent single-threaded implementation. It measures the extra overhead (*e.g.,* communication) introduced by parallel systems relative to single-threaded implementations. The results are reported in Table 3. For CC, we adopted its

Algorithms	Dataset	GRAPE cores	GRAPE time (s)	Single-threaded time (s)
SSSP	traffic	4	6.08	6.14
	Friendster	4	96.3	101.1
	UKWeb	2	149.1	151.8*
CC	traffic	2	1.52	1.63
	Friendster	4	35.5	40.3
	UKWeb	2	21.6	27.9*
Sim	DBpedia	2	7.11	7.32
	Friendster	2	44.0	93.7*
SubIso	DBpedia	24	0.08	0.08
	Friendster	32	20.4	25.0*
CF	movieLens	2	460.0	733

Table 3. COST of GRAPE

original implementation[1] of [51] as the single-threaded version. For SSSP, Sim, SubIso and CF, no implementations are given in [51], and we adopted the best single-threaded implementations to our knowledge for comparison with GRAPE. Following [51], for large input graphs that are unable to fit into RAM, we used the external I/O on SSD as an extension to RAM (marked with * in Table 3).

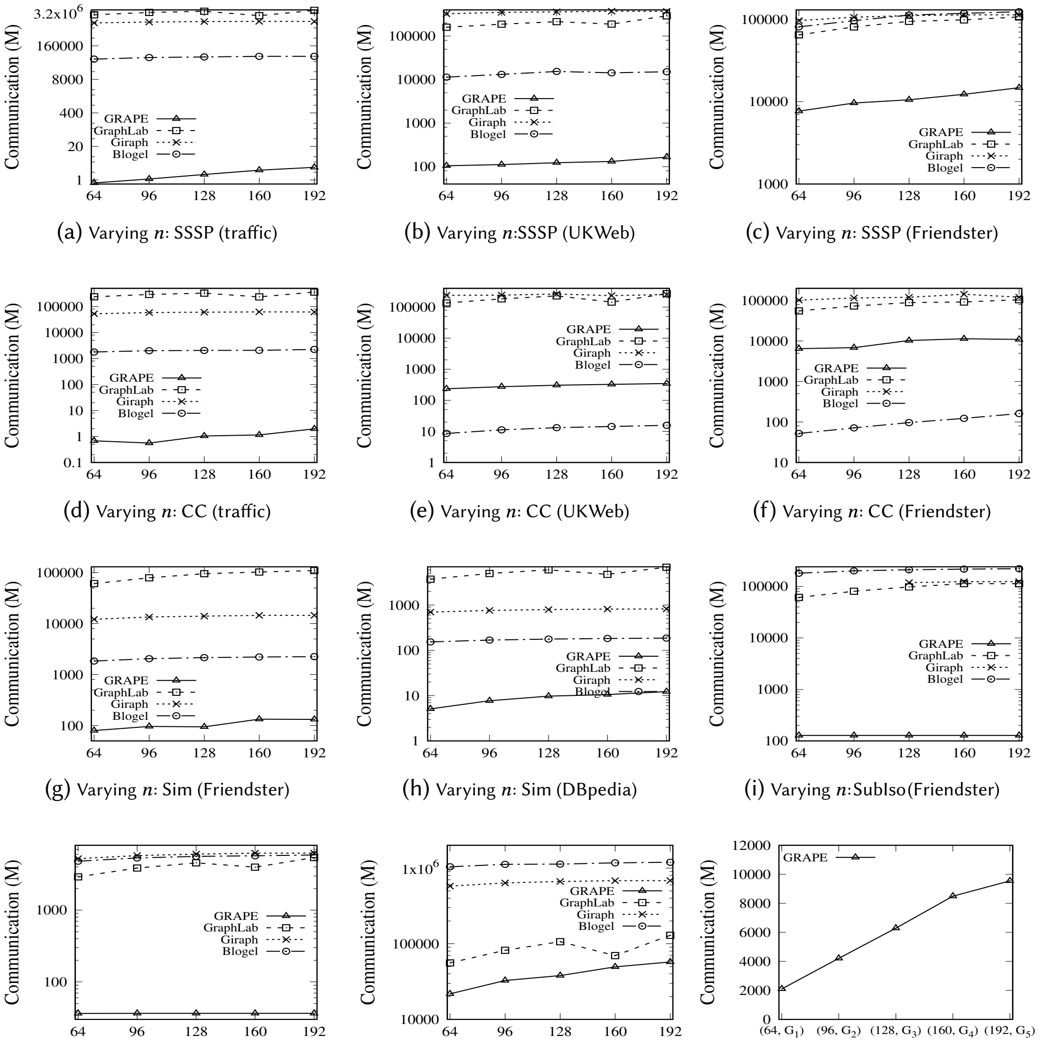
From Table 3 we can see the following. (1) For SSSP, CC, Sim and CF, GRAPE achieves speed-up over single-threaded implementations with just 2 or 4 cores over all tested input graphs. (2) For SubIso, even with its relatively heavy prefetching cost (Section 5.1), GRAPE still outperforms the single-threaded implementations with 24 or 32 cores (just 2 physical machines). (3) According to [51], GraphLab had a COST of 512 cores and Spark GraphX had unbounded COST (no configuration can outperform single-threaded). Therefore, GRAPE demonstrates better scalability than GraphLab and GraphX, with smaller extra overhead for parallel graph computations.

It should be remarked that parallelization overheads are inevitable for all distributed/parallel systems, including but not limited to GRAPE. Nonetheless, parallel processing often works better when dealing with large-scale graphs that are beyond the capacity of a single machine.

Exp-2: Communication cost. The communication cost (in bytes) reported by Giraph, GraphLab and Blogel depends on their own implementation of message blocks and protocols. As observed in [36], these built-in message or byte counters differ from each other: Blogel counts cross-process bytes, GraphLab reports cross-machine bytes, and Giraph tracks cross-partition bytes. For a fair comparison, we adopted a third-party tool Nethogs [10] following the practice [36]. It tracks the total bytes sent by each machine during the run, by monitoring the system file /proc/net/dev. This metric, better aligned to parallel models of the systems, reveals consistent results with better insights.

In the same setting as Exp-1, Figure 16 reports the communication costs of the systems. The results show that in all cases, GRAPE incurs much less communication cost than Giraph and GraphLab. On datasets excluding traffic, with 192 processors, it ships on average 0.08%, 1.1%, 0.3%, 0.18% and 8.4% of the data shipped for SSSP, Sim, CC, SubIso and CF by Giraph, and 0.11%, 0.14%, 0.3%, 0.19% and 44% by GraphLab, respectively; moreover, it reduces their cost by 6 and 5 orders of magnitude for SSSP and CC on traffic, respectively. While it ships more data than Blogel for CC due to the precomputation of Blogel remarked earlier, it only ships 6.2%, 0.1%, 1.9% and 4.8% of the data shipped by Blogel for Sim, SubIso, SSSP and CF, respectively. On traffic, GRAPE also reduces the communication cost of Blogel by 4 and 3 orders of magnitude for SSSP and CC, respectively.

[1] https://github.com/frankmcsherry/COST



(a) Varying n: SSSP (traffic)

(b) Varying n:SSSP (UKWeb)

(c) Varying n: SSSP (Friendster)

(d) Varying n: CC (traffic)

(e) Varying n: CC (UKWeb)

(f) Varying n: CC (Friendster)

(g) Varying n: Sim (Friendster)

(h) Varying n: Sim (DBpedia)

(i) Varying n:SubIso(Friendster)

(j) Varying n: SubIso (DBpedia)

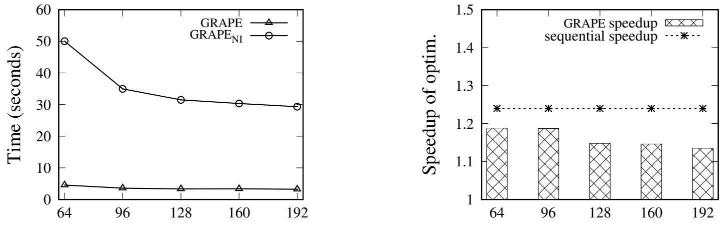
(k) Varying n: CF (movieLens)

(l) Data shipment (Synthetic)

Fig. 16. Communication costs

(1) SSSP. Figures 16a-16c show that both GRAPE and Blogel incur communication costs that are orders of magnitudes less than those of GraphLab and Giraph. This is because vertex-centric programming incurs a large amount of messages. Both block-centric programs (Blogel) and PIE programs (GRAPE) reduce unnecessary messages, and trigger inter-block communication only when necessary. We also observe that GRAPE ships 0.9% and 10% of the data shipped by Blogel over UKWeb and Friendster, respectively. Indeed, GRAPE ships only changed values of update parameters, and needs fewer supersteps. These significantly reduce the size and number of messages.

(2) CC. Figures 16d-16f demonstrate similar improvement of GRAPE over GraphLab and Giraph for CC. It ships on average 0.2% and 0.3% of the data shipped by Giraph and GraphLab on datasets excluding traffic, and 0.0015% and 0.0003% on traffic, respectively. Since Blogel precomputes CC



(a) The impact of IncEval

(b) Optimization on GRAPE

Fig. 17. Incremental steps and optimization

(see Exp-1(2)), it ships little data. Nonetheless, GRAPE is not far worse than the near "optimal" case of Blogel on Friendster and UKWeb, and ships only 0.05% of the data shipped by Blogel on traffic.

(3) Sim. Figures 16g and 16h report the communication cost for graph simulation over Friendster and DBpedia, respectively. One can see that GRAPE ships substantially less data, *e.g.,* on average 0.9%, 0.1%, 4.9% of the data shipped by Giraph, GraphLab and Blogel, respectively. Observe that the communication cost of Blogel is much higher than that of GRAPE, even though it adopts inter-block communication. This shows that the extension of vertex-centric to block-centric by Blogel may not suffice to reduce messages when it comes to complex queries. GRAPE works better than these systems by employing incremental IncEval to reduce redundant messages and computation.

(4) SubIso. Figures 16i and 16j report the results for SubIso over Friendster and DBpedia, respectively. The results are consistent with their counterparts for Sim. On average, GRAPE ships 0.18%, 0.24% and 0.11% of the data shipped by Giraph, GraphLab and Blogel, respectively.

(5) CF. Figure 16k reports the result for CF over movieLens. On average, GRAPE ships 5.6%, 43.3% and 3.2% of the data shipped by Giraph, GraphLab and Blogel, respectively.

(6) Synthetic. In the same setting as Figure 15l, Figure 16l reports the communication cost for SSSP using synthetic graphs. The results demonstrate that more communication cost is incurred over larger graphs and more processors, due to increased border nodes, as expected.

Exp-3: Incremental computation. We evaluated the effectiveness of incremental IncEval. We implemented a batch version of GRAPE for Sim queries, denoted as GRAPE$_{\text{NI}}$, 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 Figure 17a over Friendster, (1) GRAPE outperforms GRAPE$_{\text{NI}}$ by 9.0 times with 192 processors; and (2) the gap is larger when less processors are employed, *e.g.,* 11.0 times when 64 processors are used. This is because the less processors are used, the larger fragments reside at each processor, 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 (not shown).

Exp-4. Compatibility. We also evaluated the compatibility of optimization strategies developed for sequential graph algorithms with GRAPE parallelization. For a query class Q, a sequential algorithm $\mathcal{A}$ and its optimized version $\mathcal{A}^*$ for 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 $T_p(\mathcal{A})$ (resp. $T_p(\mathcal{A}^*)$) for a given number n of processors. 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 [39], and the other is an optimized version that employs indices to reduce candidates [25]. Using Sim queries over Friendster. we found that the average speedup of sequential algorithms is 1.24. Varying n from 64 to 192, we report the speedup of the parallelized algorithms of GRAPE in Figure 17b. The results 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. From the experimental results we find the following.

(1) By simply parallelizing sequential algorithms, the performance of GRAPE is already comparable to state-of-the-art systems. Using from 64 to 192 processors over real-life graphs excluding traffic, GRAPE is on average 484, 36 and 15 times faster than Giraph, GraphLab and Blogel for SSSP, 151, 6.8 and 16 times for Sim, 149.3, 34.2 and 9.6 times for SubIso, and 4.6, 2.6 and 12.4 for CF, respectively. For CC, it is 1377 and 212 times faster than Giraph and GraphLab, respectively, and is comparable to the "optimal" case of Blogel although Blogel embeds the computation of CC in its graph partition phase. On traffic, for SSSP and CC, GRAPE is on average 4, 3 and 2 orders of magnitude faster than Giraph, GraphLab and Blogel, respectively. The results on synthetic graphs are consistent.

(2) In the same setting, on datasets excluding traffic GRAPE ships on average 0.07%, 0.12% and 1.7% of the data shipped across machines by Giraph, GraphLab and Blogel for SSSP, 0.89%, 0.14% and 4.9% for Sim, 0.18%, 0.23% and 0.11% for SubIso, 5.6%, 43.3% and 3.2% for CF, respectively. For CC, it incurs 0.23% and 0.3% of data shipment of Giraph and GraphLab, and is comparable with "optimized" Blogel. On traffic, for SSSP and CC, it ships on average 5, 6 and 3 orders of magnitude less data shipment by Giraph, GraphLab and Blogel, respectively.

(3) GRAPE demonstrates good scalability when using more processors, since its incremental computation mitigates the impact of more border nodes and fragments.

(4) Incremental steps effectively reduce iterative recomputation. For Sim, it improves the response time by 9.6 times on average.

(5) GRAPE inherits the benefit of optimized sequential algorithms. For Sim, it is on average 20% faster by using the algorithm of [25] instead of the algorithm of [39].

8 CONCLUSION

We have proposed an approach to parallelizing sequential graph algorithms. Given a class Q of graph queries, users can devise existing sequential algorithms for Q with minor changes, without recasting the entire algorithms into a new model. GRAPE parallelizes the computation and guarantees to converge at correct answers under a monotonic condition, as long as the sequential algorithms are correct. Moreover, graph algorithms that are developed for existing parallel graph systems can be migrated to GRAPE, without incurring extra complexity. We have verified that GRAPE achieves comparable performance to the state-of-the-art graph systems for various query classes, and that (bounded) IncEval effectively reduces unnecessary recomputation and hence the cost of iterative graph computations. We hope that GRAPE will make parallel graph computations accessible to a large group of users who are more familiar with sequential algorithms.

A preliminary implementation of GRAPE is available at [9]. We are in the process of implementing asynchronous message passing, based on [27]. 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 update load is light, GRAPE adopts non-destructive updates that have proven useful in functional databases [64]. Otherwise, it switches to multi-version concurrency control [14] that keeps track of timestamps and versions, as adopted by existing distributed systems.

One topic for future work is to revise the asynchronous model of [27] to maximize the benefit of pipelined parallelism and data-partitioned parallelism. Another topic is to develop methods for incrementalizing graph algorithms with performance guarantees, extending [11, 16, 24].

ACKNOWLEDGMENTS

The authors are supported in part by 973 Program 2014CB340302, ERC 652976, NSFC 61421003, EPSRC EP/M025268/1, the Foundation for Innovative Research Groups of NSFC, and Beijing Advanced Innovation Center for Big Data and Brain Computing.

REFERENCES

[1] 2006. UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/*. (2006).

[2] 2010. Traffic. (2010). *http://www.dis.uniroma1.it/challenge9/download.shtml*.

[3] 2011. Movielens. *http://grouplens.org/datasets/movielens/*. (2011).

[4] 2012. Friendster. *https://snap.stanford.edu/data/com-Friendster.html*. (2012).

[5] 2012. MPICH. (2012). *https://www.mpich.org/*.

[6] 2014. Giraph. (2014). *http://giraph.apache.org/*.

[7] 2015. DBpedia. (2015). *http://wiki.dbpedia.org/Datasets*.

[8] 2017. Apache Hadoop. (2017). *http://hadoop.apache.org/*.

[9] 2017. GRAPE. *http://grapedb.io/*. (2017).

[10] 2017. Nethogs. (2017). `https://github.com/raboof/nethogs`.

[11] Umut A. Acar. 2005. *Self-Adjusting Computation*. Ph.D. Dissertation. CMU.

[12] Konstantin Andreev and Harald Racke. 2006. Balanced graph partitioning. *Theory of Computing Systems* 39, 6 (2006), 929–939.

[13] Jrgen Bang-Jensen and Gregory Z. Gutin. 2008. *Digraphs: Theory, Algorithms and Applications*. Springer.

[14] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.

[15] Dimitri P Bertsekas and John N Tsitsiklis. 1997. Parallel and Distributed Computation: Numerical Methods. (1997).

[16] Pramod Kumar Bhatotia. 2015. *Incremental Parallel and Distributed Systems*. Ph.D. Dissertation. Saarland University.

[17] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. 2013. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *SC*.

[18] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *SIGKDD*. 1456–1465.

[19] Peter Buneman, Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. 2006. Using Partial Evaluation in Distributed Query Evaluation. In *VLDB*.

[20] Badong Chen, Jianji Wang, Haiquan Zhao, Nanning Zheng, and José C. Príncipe. 2015. Convergence of a Fixed-Point Algorithm under Maximum Correntropy Criterion. *IEEE Signal Process. Lett.* 22, 10 (2015), 1723–1727.

[21] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SICOMP* 32, 5 (2003), 1338–1355.

[22] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008).

[23] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. 2015. Keys for Graphs. *PVLDB* 8, 12 (2015), 1590–1601.

[24] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental Graph Computations: Doable and Undoable. In *SIGMOD*.

[25] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph Pattern Matching: From Intractability to Polynomial Time. In *PVLDB*.

[26] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *SIGMOD*.

[27] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. 2018. Adaptive Asynchronous Parallelization of Graph Algorithms. In *SIGMOD*.

[28] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *TODS* 38, 3 (2013).

[29] Wenfei Fan, Xin Wang, and Yinghui Wu. 2014. Distributed graph simulation: Impossibility and possibility. *PVLDB* 7, 12 (2014), 1083–1094.

[30] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. 2015. Association Rules with Graph Patterns. *PVLDB* 8, 12 (2015), 1502–1513.

[31] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing Sequential Graph Computations. In *SIGMOD*. 495–510.

[32] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM* 34, 3 (1987), 596–615.

[33] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. 2011. Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. In *KDD*. 69–77.

[34] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*.

[35] Elaine T. Hale, Wotao Yin, and Yin Zhang. 2008. Fixed-Point Continuation for ℓ_1-Minimization: Methodology and Convergence. *SIAM Journal on Optimization* 19, 3 (2008), 1107–1130.

[36] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Ozsu, Xingfang Wang, and Tianqi Jin. 2014. An experimental comparison of Pregel-like graph processing systems. *VLDB* 7, 12 (2014).

[37] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *SIGMOD*.

[38] Tim J. Harris. 1994. A Survey of PRAM Simulation Techniques. *ACM Comput. Surv.* 26, 2 (1994), 187–206.

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

[40] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*. 1223–1231.

[41] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. In *SIGMOD*. 463–478.

[42] N. D. Jones. 1996. An Introduction to Partial Evaluation. *Comput. Surveys* 28, 3 (1996).

[43] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A Model of Computation for MapReduce. In *SODA*.

[44] George Karypis and Vipin Kumar. 1995. *METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0.* Technical Report.

[45] Arijit Khan, Yinghui Wu, Charu C Aggarwal, and Xifeng Yan. 2013. Nema: Fast graph search with label similarity. *PVLDB* 6, 3 (2013).

[46] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A system for dynamic load balancing in large-scale graph processing. In *EuroSys*. 169–182.

[47] Mijung Kim and K Selçuk Candan. 2012. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering* 72 (2012), 285–303.

[48] Yehuda Koren, Robert Bell, Chris Volinsky, et al. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.

[49] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB* 5, 8 (2012).

[50] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *SIGMOD*.

[51] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*.

[52] Joris M Mooij and Hilbert J Kappen. 2007. Sufficient conditions for convergence of the sum–product algorithm. *IEEE Transactions on Information Theory* 53, 12 (2007), 4422–4437.

[53] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. In *ASPLOS*.

[54] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *ACM Sigplan Notices*. 12–25.

[55] Cosmin Radoi, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. 2014. Translating imperative code to MapReduce. In *OOPSLA*.

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

[57] G. Ramalingam and Thomas Reps. 1996. On the computational complexity of dynamic graph problems. *Theoretical Computer Science* 158, 1-2 (1996).

[58] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *SOSP*.

[59] Semih Salihoglu and Jennifer Widom. 2013. GPS: A graph processing system. In *SSDBM*.

[60] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. 2013. "All roads lead to Rome": Optimistic recovery for distributed iterative data processing. In *CIKM*. 1919–1928.

[61] George M Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. 2017. Partitioning Trillion-edge Graphs in Minutes. In *IPDPS*.

[62] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *KDD*. 1222–1230.

[63] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, and John McPherson Shirish Tatikonda. 2013. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB* 7, 7 (2013), 193–204.

[64] Phil Trinder. 1989. *A Functional Database*. Ph.D. Dissertation. University of Oxford.

[65] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.

[66] Leslie G. Valiant. 1990. General Purpose Parallel Architectures. In *Handbook of Theoretical Computer Science, Vol A*.

[67] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. 2013. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*.

[68] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: Time to fuse for distributed graph-parallel computation. In *PPOPP*.

[69] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Trans. Big Data* 1, 2 (2015), 49–67.

[70] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Foundations and Trends in Databases* 7, 1-2 (2017), 1–195.

[71] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *PVLDB* 7, 14 (2014), 1981–1992.

[72] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*.

[73] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. *PVLDB* 7, 14 (2014), 1821–1832.

[74] Zhensheng Zhang and Christos Douligeris. 1991. Convergence of Synchronous and Asynchronous Algorithms in Multiclass Networks. In *INFOCOM*. 939–943.

[75] Yang Zhou, Ling Liu, Kisung Lee, Calton Pu, and Qi Zhang. 2015. Fast iterative graph computation with resource aware graph parallel abstractions. In *HPDC*.

SUPPLEMENTARY MATERIALS

Proof of Theorem 4.2

(1) Simulate BSP. We first show that all BSP programs can be optimally simulated by GRAPE. Recall that a BSP algorithm $\mathcal{A}$ consists of the following three components: (1) n workers that perform computation of some function f; (2) a router that ships data between workers; and (3) a mechanism for synchronizing the computations of the workers [65].

To simulate $\mathcal{A}$, we use PEval and IncEval to simulate the function f, and coordinator P_0 to play the role of the router. To simulate arbitrary data shipment of BSP, similar to the construction in [27], we construct a complete graph G_W of n nodes as additional input of the BSP program. That is, G_W is a clique of n nodes, which assigns one node w_i to each worker P_i for $i \in [1, n]$, and stores the data transferred to P_i in the status variables of w_i, as its update parameters. The reason for using G_W is because in GRAPE, only the update parameters at a worker can be exchanged between workers. By employing G_W, all the n nodes become border nodes, and we can simulate the arbitrary data shipment of $\mathcal{A}$ by storing the data in the update parameters of GRAPE workers, and encoding the shipments as messages. The coordinator P_0 collects messages from workers, and routes them to the corresponding workers. GRAPE adopts the same synchronization mechanism as BSP.

More specifically, we simulate a given BSP algorithm $\mathcal{A}$ with a PIE program $\mathcal{B}$ as follows.

(a) Data partition: Program $\mathcal{B}$ also uses n workers. It partitions and distributes the input of $\mathcal{A}$ in exactly the same way as $\mathcal{A}$. In addition, as mentioned above, $\mathcal{B}$ takes as input a complete graph G_W of n nodes, which is evenly distributed across the n works, one node designated for each worker.

(b) Update parameters. The update parameters of $\mathcal{B}$ are such defined that they encode the messages transferred by $\mathcal{A}$. That is, for each node w_i in G, a status variable x is declared such that $w_i.x$ maintains the data at P_i to be exchanged with other workers.

(c) PEval *and* IncEval: Both PEval and IncEval of $\mathcal{B}$ are simply the function f of $\mathcal{A}$.

(d) Message grouping. The GRAPE message scheme of $\mathcal{B}$ is used to (i) group the received messages; (ii) distribute the workload to workers. More specifically, the coordinator P_0 does the following.

- It first collects all the update parameters for each worker P_i ($i \in [1, n]$).
- It then sends (the changed values of) the update parameters designated to worker P_i ($i \in [1, n]$), in the same way as $\mathcal{A}$ does.

The complete graph G_W allows $\mathcal{B}$ to exchange messages between different workers, *i.e.,* to support point-to-point message passing. The data shipment strategies of BSP can be encoded by the aggregate function of $\mathcal{B}$, to dispatch messages just like $\mathcal{A}$ does.

(e) Assemble: The Assemble function simply collects and returns the partial results from workers.

Given these, one can verify that the PIE program correctly simulates $\mathcal{A}$ under GRAPE. That is, for any input I of algorithm $\mathcal{A}$, $\mathcal{B}$ takes I and G_W as input and returns the same result as $\mathcal{A}$.

Next, we verify that the simulation is optimal. Suppose that given an input I, $\mathcal{A}$ runs in k rounds using T time. Suppose that at superstep r ($r \in [1, k]$), $\mathcal{A}$ takes T_r time and sends C_r amount of data. We show that $\mathcal{B}$ also does the job in k supersteps and takes $O(T)$ time. To this end, it suffices to show that $\mathcal{B}$ simulates round r in $O(T_r)$ times, and moreover, it sends $O(C_r)$ amount of data as messages. To see this, observe that when simulating round r of $\mathcal{A}$, $\mathcal{B}$ runs the same function f on the same data, and sends the same data as messages to the coordinator P_0 as $\mathcal{A}$ does. The coordinator then routes the message to other workers. Thus $\mathcal{B}$ simulates round r of $\mathcal{A}$ in $O(T_r)$ times, with $O(C_r)$ as its communication cost. Formally, this can be verified by induction on r.

Putting these together, we can see that GRAPE can optimally simulate BSP.

(2) Simulate MapReduce. It has been shown in [27] that all MapReduce programs using n processors can be optimally simulated by GRAPE using n processors. For the completeness of this paper, we also present the construction in [27] here, and provide more details of the proof.

A MapReduce algorithm $\mathcal{A}$ is specified as follows. The input of $\mathcal{A}$ is a multi-set I_0 of $\langle$*key, value*$\rangle$ pairs. The operations of $\mathcal{A}$ are characterized as a sequence $(B_1, \ldots, B_k)$ of subroutines, where B_r ($r \in [1, k]$) consists of a mapper μ_r and a reducer ρ_r. Given input I_0, $\mathcal{A}$ iteratively runs the subroutines B_r ($r \in [1, k]$) as follows [22, 43].

(i) The mapper μ_r processes each pair $\langle$*key, value*$\rangle$ in I_{r-1} one by one, and produces a multi-set I'_r of $\langle$*key, value*$\rangle$ pairs as output.

(ii) Group pairs in I'_r by the *key* values. That is, two pairs are put in the same group if and only if they have the same *key* value. Group I'_r by distinct *key*s; this yields G_{k_1}, …, and G_{k_j}.

(iii) The reducer ρ_r processes the groups G_{k_l} ($l \in [1, j]$) one by one, and produces a multi-set I_r of $\langle$*key, value*$\rangle$ pairs as output.

(iv) If $r < k$, then $\mathcal{A}$ continues to run the next subroutine B_{r+1} on I_i in the same manner as steps (i)-(iii); otherwise, $\mathcal{A}$ outputs I_k and terminates.

Here I_0 is the input of $\mathcal{A}$, and I_r is the output of B_r ($r \in [1, k]$). There are two other parameters, namely, M_r and N_r ($r \in [1, k]$), which are the numbers of instances of mappers and reducers for each subroutine B_r, respectively. That is, there exist M_r processors and N_r processors running μ_r and ρ_r, respectively. As put in [43], each mapper (resp. reducer) processes only one $\langle$*key, value*$\rangle$ pair (resp. group) at a time. To simplify the discussion, we assume *w.l.o.g.* that each routine uses n mappers and n reducers, by reassigning the workload when needed. Generally speaking, there also exists a partition strategy for distributing the outputs of the mappers to the reducers in a MapReduce algorithm. To simplify the discussion, we only consider the hash partition used as default methods in Hadoop [8]. Other partition methods can be simulated similarly.

To simulate the process of the MapReduce algorithm $\mathcal{A}$ above, we use PEval to simulate the mapper μ_1 of B_1, and IncEval to play the roles of reducer ρ_i and mapper μ_{i+1} in each superstep, for $i \in [1, k-1]$, and the final reducer ρ_k. Complications arise from the following mismatches: (a) $\mathcal{A}$ has a sequence $(B_1, \ldots, B_k)$ of subroutines, while algorithms of GRAPE have only three functions, namely, PEval, IncEval, and Assemble; and (b) $\mathcal{A}$ groups data by keys and distributes $\langle$*key, value*$\rangle$ pairs across processors, while workers of GRAPE can only exchange the values of update parameters. In a nutshell, to solve problem (a), we define IncEval that takes the subroutines B_1, …, B_k of $\mathcal{A}$ as program branches, and makes use of an index $r(r \in [1, k])$ to select correct branches via update parameters. We use the status variables of update parameters to store tuples of form $\langle r$, *key, value*$\rangle$ (for $r \in [1, k]$), where r is the index for selecting subroutines in IncEval. GRAPE picks the correct subroutines in IncEval by properly maintaining and citing index r. We solve problem (b) along the same lines as what we do for simulating BSP above, by employing a complete graph G_W as additional input. We use boarder nodes of G_W to denote processors, and employ update parameters $C_i.\bar{x}$ to store associated data.

More specifically, given a MapReduce algorithm $\mathcal{A}$ with n processors, we simulate $\mathcal{A}$ with a PIE program $\mathcal{B}$ under GRAPE with n workers. Suppose that the input of $\mathcal{A}$ is a multi-set I_0 of $\langle$*key, value*$\rangle$.

(a) Data partition: Similar to the simulation of BSP, we construct a complete graph G_W of n nodes as additional input of the GRAPE algorithm $\mathcal{B}$, such that each worker P_i is represented by a node w_i for $i \in [1, n]$. Each node w_i has a status variable x to store a multi-set of $\langle r$,*key, value*$\rangle$ tuples. As mentioned above, r keeps track of different rounds of the computation of $\mathcal{B}$, for IncEval to select

right subroutines. We distribute the ⟨*key, value*⟩ pairs in I_0 in exactly the same way as $\mathcal{A}$ does before running subroutine B_1; the status variable of each w_i in G_W contains the pairs assigned to worker P_i.

(b) Update parameters. As remarked earlier, for each node w_i in G_W, its status variable x takes a multi-set of ⟨*r*, *key, value*⟩ tuples as its value, and the update parameter at worker P_i is $w_i.x$.

(c) PEval: PEval simulates mapper μ_1 of subroutine B_1 and organizes its output, as follows.

- Each worker first runs the mapper μ_1 of subroutine B_1 on its local data.
- Next, it prepares the output of μ_1 and stores it in the update parameters for later supersteps. Suppose that the output of μ_1 is $(I_1)'$. For each pair ⟨*key, value*⟩ in $(I_1)'$, it includes a tuple $\langle 1, key, value\rangle$ in an update parameter. Here the index 1 in the tuples indicates that these tuples are the results of the mapper μ_1.

This is possible since graph G_W is complete, all the nodes in G_W are border nodes, and each node in G denotes a worker. Hence each worker can modify the update parameters of all the other workers.

(d) Message grouping. The coordinator groups and routes messages as follows.

- It first takes a union of the update parameters of all node w_i ($i \in [1, n]$), and then groups the update parameters by the *key* values.
- Next, to balance the workload, it uses hash partition to assign each tuple ⟨*r*, *key, value*⟩ to a reducer (*i.e.,* a worker), following Hadoop. More specifically, suppose that h is the hash function used. Then all tuples ⟨*r*, *key, value*⟩ with the same *key* will be assigned to worker P_j with $j = h(key) \bmod n + 1$, where n is the total number of workers, and $r \in [1, k]$ indicates the round of computation. After this, PEval stores all tuples ⟨*r*, *key, value*⟩ with the specific *key* in the update parameter of worker P_j.

That is, the **aggregateMsg** function groups the ⟨*r*, *key, value*⟩ tuples by the *key* value, and assigns each group to a worker based on hash function h; the group of tuples is sent to the worker as messages and stored in its update parameter.

(e) IncEval. Upon receiving messages from the coordinator P_0, IncEval first extracts the index r from the ⟨*r*, *key, value*⟩ tuples in the messages, and uses the index r to pick the right subroutine in IncEval. Note that although we run different subroutines by IncEval, GRAPE uses the same function IncEval through the course of the execution. This is done by treating subroutines as branch programs in IncEval, controlled by index r. More specifically, IncEval does the following.

- Extract a multi-set of ⟨*key, value*⟩ pairs from the ⟨*r*, *key, value*⟩ tuples received. Denote by $(I_r)'$ the result of the process.
- On $(I_i)'$, run the reducer ρ_r, which can be seen as a branch program of IncEval. Suppose that I_r is the output of the reducer ρ_r.
- If $r = k$, *i.e.,* if the reducer ρ_k has finished the job, then IncEval sets the update parameter of each worker to be empty, which will terminate the algorithm $\mathcal{B}$; otherwise, IncEval runs mapper μ_{r+1} on I_r, and generates tuples of the form ⟨$r + 1$, *key, value* ⟩ from the output of μ_{r+1}. The output is processed in the same way as described in the message grouping part above.

(f) Assemble: The Assemble function simply takes a union of the partial results from all workers.

Having these functions, the PIE program $\mathcal{B}$ simulates $\mathcal{A}$ under GRAPE as follows. Suppose that $\mathcal{A}$ runs in k rounds. Then $\mathcal{B}$ runs in $k + 1$ rounds, since given a subroutine (μ_r, ρ_r) ($r \in [1, k]$), $\mathcal{B}$ simulates the mapper μ_r in round r, and the reducer ρ_r in round $r + 1$. The extra round is also needed to set parameters empty and terminate the process. By induction on k, one can easily verify that given any input I_0 of ⟨*key, value*⟩ pairs, $\mathcal{A}$ and $\mathcal{B}$ produce the same result.

We next show that under GRAPE, the PIE program $\mathcal{B}$ optimally simulates the MapReduce algorithm $\mathcal{A}$. Suppose that given input I_0, $\mathcal{A}$ runs in T time, and has communication cost C. More specifically, a subroutine B_r ($r \in [1, k]$) takes T_r time, and sends C_r amount of data. To show that $\mathcal{B}$ runs in $O(T)$ time and sends $O(C)$ amount of data, it suffices to show that $\mathcal{B}$ simulates B_r in $O(T_r)$ time, and sends messages of size $O(C_r)$ in the round. To verify this, observe that when simulating B_r, $\mathcal{B}$ runs the same functions μ_r and ρ_r on the same data, except that $\mathcal{B}$ encapsulates the produced $\langle key, value\rangle$ pairs as $\langle r, key, value\rangle$ tuples. The update parameters $\langle r, key, value\rangle$ are generated by the workers in parallel, also in $O(T_r)$ time. In addition, since we only add a number (index r) to each key-value pair in the update parameters, $\mathcal{B}$ sends messages of size $O(C_r)$ in the round. Putting these together, we have that $\mathcal{B}$ simulates B_r in $O(T_r)$ time, and incurs communication cost $O(C_r)$ via messages. Hence, $\mathcal{B}$ runs in $O(T)$ time, and sends $O(C)$ amount of data in total.

From the argument above it follows that GRAPE can optimally simulate MapReduce.

(3) Simulate PRAM. Finally, we show that all PRAM programs can be simulated by GRAPE.

PRAM employs a number of processors that share memory, and each processor can access any memory cell in unit time. The computation is typically 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 by various access policies of shared memory, *e.g.*, CREW PRAM indicates concurrent read and exclusive write (see [66] 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 [43]. By Theorem 4.2(2) above, each MapReduce algorithm in r rounds can be simulated by GRAPE in $r + 1$ supersteps. From these Theorem 4.2(3) follows. □

Received February 2007; revised March 2009; accepted June 2009