Think Sequential, Run Parallel

Wenfei Fan[1,2], Muyang Liu[2], Ruiqi Xu[1], Lei Hou[2], Dongze Li[2], Zizhong Meng[2]

[1] University of Edinburgh, UK
[2] Beihang University, China

Abstract. Parallel computation is often a must when processing large-scale graphs. However, it is nontrivial to write parallel graph algorithms with correctness guarantees. This paper presents the programming model of GRAPE, a parallel GRAPh Engine [19]. GRAPE allows users to "plug in" sequential (single-machine) graph algorithms as a whole, and it parallelizes the algorithms across a cluster of processors. In other words, it simplifies parallel programming for graph computations, from think parallel to think sequential. Under a monotonic condition, it guarantees to converge at correct answers as long as the sequential algorithms are correct. We present the foundation underlying GRAPE, based on simultaneous fixpoint computation. As examples, we demonstrate how GRAPE parallelizes our familiar sequential graph algorithms. Furthermore, we show that in addition to its programming simplicity, GRAPE achieves performance comparable to the state-of-the-art graph systems.

1 Introduction

There has been increasing demand for graph computations, *e.g.,* graph traversal, connectivity, pattern matching, and collaborative filtering. Indeed, graph computations have found prevalent use in mobile network analysis, pattern recognition, knowledge discovery, transportation networks, social media marketing and fraud detection, among other things. In addition, real-life graphs are typically big, easily having billions of nodes and trillions of edges [24]. With these comes the need for parallel graph computations. In response to the need, several parallel graph systems have been developed, *e.g.,* Pregel [33], GraphLab [32, 22], Trinity [42], GRACE [47], Blogel [50], Giraph++ [44], and GraphX [23].

However, users often find it hard to write and debug parallel graph programs using these systems. The most popular programming model for parallel graph algorithms is the vertex-centric model, pioneered by Pregel and GraphLab. For instance, to program with Pregel, one needs to "think like a vertex", by writing a user-defined function *compute(msgs)* to be executed at a vertex v, where v communicates with other vertices by message passing (*msgs*). Although graph computations have been studied for decades and a large number of sequential (single-machine) graph algorithms are already in place, to use Pregel, one has to recast the existing algorithms into vertex-centric programs. Trinity and GRACE also support vertex-centric programming. While Blogel and Giraph++ allow blocks to have their status as a "vertex" and support block-level communication, they still adopt the vertex-centric programming paradigm. GraphX also recasts graph computation into its distributed dataflow framework as a sequence of

join and group-by stages punctuated by map operations, on the Spark platform. The recasting is nontrivial for users who are not very familiar with the parallel models. Moreover, none of the systems provides a guarantee on the correctness or even termination of parallel programs developed in their models. These make the existing systems a privilege for experienced users only.

Is it possible to simplify parallel programming for graph computations, from think parallel to think sequential? That is, can we have a system that allows users to plug in existing sequential graph algorithms for a computational problem, and it automatically parallelizes the computation across a cluster of processors? Better yet, is there a general condition under which the parallelization guarantees to converge at correct answers as long as the sequential algorithms plugged in are correct? After all, humans find it far easier to devise sequential processes that cope with the interference and synchronisation required in parallel algorithms.

It was to answer this question that we developed GRAPE [19], a parallel GRAPh Engine for graph computations. The main objective of GRAPE is to make parallel graph computations accessible to a large group of users. It allows users to think sequential and go parallel, by parallelizing sequential graph algorithms as a whole. Moreover, under a monotonic condition, it guarantees to converge at correct answers when provided with correct sequential graph algorithms. As proof of concept, GRAPE has been developed [18] and evaluated in industry. In addition to its programming simplicity, it outperforms the state-of-the-art parallel graph systems in scalability and efficiency.

The remainder of the paper is organized as follows. We present the parallel model underlying GRAPE (Section 2), based on simultaneous fixpoint computation with partial evaluation and incremental computation. We then demonstrate how our familiar sequential graph algorithms are parallelized by GRAPE (Section 3), including single-source shortest path (SSSP), graph simulation (Sim), connected components (CC) and minimum spanning tree (MST). In addition, we provide an empirical study to demonstrate the scalability and efficiency of GRAPE, compared to the state-of-the-art graph systems (Section 4). Finally, we discuss related work and identify topics for future research (Section 5).

2 From Think Parallel to Think Sequential

We next present the programming model and parallel model of GRAPE [19].

2.1 Graphs and Graph Partition

We start with basic notations, in particular graph partitions.

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; and (3) each node v in V (resp. edge $e \in E$) carries $L(v)$ (resp. $L(e)$), indicating its content, as found in social networks, knowledge bases and property graphs.

We will 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. $L'(e) = L(e)$). 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. GRAPE supports data-partitioned parallelism: it partitions a graph G and distributes fragments of G across m processors, such that computations on G can be conducted in parallel on the fragments. More specifically, given a graph G and a number m, a graph partition strategy $\mathcal{P}$ partitions G into *fragments* $(F_1, \ldots, F_m)$ 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 $i \in [1, m]$.

GRAPE allows users to pick a strategy $\mathcal{P}$ to partition G, *e.g.,* vertex-cut [30] or edge-cut [7]. When $\mathcal{P}$ is vertex-cut, denote by

- $F_i.O$ the set of *border nodes* $v \in V_i$ such that there exists a copy of v in another fragment F_j $(i \neq j)$; and
- $\mathcal{F}.O = \bigcup_{i\in[1,m]} F_i.O$.

Similarly, border nodes are defined under edge-cut, which have an edge to (or from) nodes in another fragment (see [19] for details).

We adopt vertex-cut in the sequel unless stated otherwise; the results of the paper still hold under other partition strategies.

2.2 Programming Model

Consider a graph computation problem $\mathcal{Q}$. Informally, a parallel program for $\mathcal{Q}$ is a program that operates on a graph G, where G is partitioned and distributed across a cluster of processors. Here we assume that the cluster adopts the shared nothing architecture in which processors do not share memory or disk storage, and the processors exchange information among themselves by message passing, as commonly adopted nowadays. In our familiar terms, we refer to an instance of the problem (excluding graph G) as a query Q of $\mathcal{Q}$. Given a query $Q \in \mathcal{Q}$, the program computes the set $Q(G)$ of answers to Q in graph G by operating on the fragments of G in parallel with the processors.

To develop a parallel algorithm for a class $\mathcal{Q}$ of queries with GRAPE, one only needs to specify the following three functions.

(1) PEval: a sequential (single-machine) algorithm for $\mathcal{Q}$ that given a query $Q \in \mathcal{Q}$ and a graph G, computes the answer $Q(G)$ to Q in G.
(2) IncEval: a sequential incremental algorithm for $\mathcal{Q}$ that given Q, G, $Q(G)$ and updates ΔG to G, computes updates ΔO to the old output $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$, where $G \oplus \Delta G$ denotes G updated by ΔG [40].
(3) Assemble: a function that collects partial answers computed locally at each processor by PEval and IncEval, and assembles the partial results into complete answer $Q(G)$. This function is often straightforward.

The three functions are referred to as *a* PIE *program for* $\mathcal{Q}$ (PEval, IncEval and Assemble). Note that PEval and IncEval can be any *existing sequential* (incremental) algorithms for $\mathcal{Q}$ that operate on a fragment F_i of graph G partitioned via a strategy $\mathcal{P}$. Note that fragment F_i is a graph itself.

The only additions are the following declarations in PEval.

(a) Update parameters. PEval declares *status variables* $\bar{x}$ for a set C_i of nodes in a fragment F_i, which store contents of F_i or intermediate results of a computation. Here C_i is a set of nodes and edges within d-hops of the nodes in $F_i.O$, for an integer d that is determined by Q. When $d = 0$, C_i is $F_i.O$.

We denote by $C_i.\bar{x}$ the set of *update parameters* of F_i, including status variables associated with the nodes and edges in C_i. As will be seen shortly, the variables in $C_i.\bar{x}$ are candidates to be updated by incremental steps IncEval.

(b) Aggregate functions. PEval also specifies an aggregate function f_{aggr}, *e.g.,* min and max, for conflict resolution, *i.e.,* to resolve conflicts when multiple processors attempt to assign different values to the same update parameter.

Update parameters and aggregate function are specified in PEval and are shared by IncEval. We will provide examples in Section 3.

2.3 Parallel Computation Model

We next show how GRAPE parallelizes a PIE program ρ (PEval, IncEval, Assemble) for $\mathcal{Q}$. Given a partition strategy $\mathcal{P}$ and PIE program ρ, GRAPE first partitions G into $(F_1, \ldots, F_m)$ with $\mathcal{P}$, and distributes the fragments across m shared-nothing *virtual workers* (*i.e.,* processors) $(P_1, \ldots, P_m)$, respectively. It maps m virtual workers to n physical workers. When $n < m$, multiple virtual workers that are mapped to the same worker share memory.

Note that graph G is partitioned *once* for *all queries* $Q \in \mathcal{Q}$ on G.

We start with basic ideas behind GRAPE parallelization.

(1) Given a function $f(s, d)$ and the s part of its input, *partial evaluation* is to specialize $f(s, d)$ *w.r.t.* the known input s [29]. That is, it performs the part of f's computation that depends only on s, and generates a partial answer, *i.e.,* a residual function f' that depends on the as yet unavailable input d. For each worker P_i in GRAPE, its local fragment F_i is its known input s, while the data residing at other workers is the yet unavailable input d. As will be seen shortly, given a query $Q \in \mathcal{Q}$, GRAPE computes $Q(F_i)$ in parallel as partial evaluation.

(2) Workers exchange *changed values* of their local update parameters with each other. Upon receiving message M_i that consists of changes to the update parameters at fragment F_i, worker P_i treats M_i as *updates* to F_i, and *incrementally* computes changes ΔO_i to $Q(F_i)$ such that $Q(F_i \oplus M_i) = Q(F_i) \oplus \Delta O_i$, making maximum reuse of previous results $Q(F_i)$. This is often more efficient than recomputing $Q(F_i \oplus M_i)$ starting from scratch, since in practice M_i is typically small, and so is O_i. Better still, the incremental computation may be *bounded*: its cost can be expressed as a function in $|M_i| + |\Delta O_i|$, *i.e.,* the size of changes in the input and output, instead of $|F_i|$, no matter how big F_i is [40, 16].

Model. Given a query $Q \in \mathcal{Q}$ at the master (processor) P_0, GRAPE answers Q in the partitioned graph G following BSP [45]. It posts the same query Q to all the workers, and computes $Q(G)$ in three phases as follows, as shown in Fig. 1.

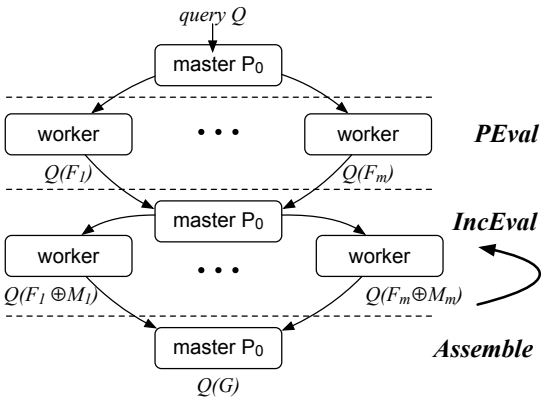


Fig. 1. Workflow of GRAPE

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

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

(2) Incremental computation (**IncEval**). GRAPE iterates the following supersteps until it terminates. Following BSP, each superstep starts after the master P_0 receives messages (possibly empty) from *all workers* P_i for $i \in [1, m]$. A superstep has two steps itself, one at P_0 and the other at the workers.

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

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

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

(3) Termination (**Assemble**). At each superstep, master P_0 checks whether for all $i \in [1, m]$, P_i is inactive, *i.e.,* P_i is done with its local computation, and there exist no more changes to any update parameter of F_i. If so, GRAPE invokes Assemble at P_0, which pulls partial results from all workers, groups together the partial results and gets the final result at P_0, denoted by $\rho(Q, G)$. It returns $\rho(Q, G)$ and terminates. Otherwise, it proceeds to the next superstep (step (2)).

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

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

$$R_i^{r+1} = \mathsf{IncEval}(Q, R_i^r, F_i^r[\bar{x}_i], M_i), \tag{2}$$

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

2.4 Features of GRAPE

As outlined above, GRAPE has the following unique features.

(1) Parallel programming simplicity. GRAPE allows users to plug in sequential graph algorithms as a whole (subject to declarations of update parameters and aggregate function in PEval), and executes these algorithms on fragmented and distributed graphs. That is, users can "think sequential" when programming with GRAPE, instead of think parallel. Moreover, a large number of sequential graph algorithms are already in place after decades of study, and are well optimized. Moreover, there have been methods for incrementalizing graph algorithms, to get incremental algorithms from their batch counterparts [6, 14]. Furthermore, as will be shown in Sections 3.2 and 3.4, it is quite straightforward to develop IncEval by revising a batch sequential algorithm. These make parallel graph computations accessible to college students who know conventional graph algorithms covered in undergraduate textbooks.

This said, GRAPE cannot be used without some insight by simply plugging in sequential algorithms without making any change. Programming with GRAPE still requires to declare update parameters and an aggregate function.

(2) Correctness guarantees. Under a general condition, GRAPE parallelization is guaranteed to converge at correct answers. To see this, we use the following notations. (a) A sequential algorithm PEval for $\mathcal{Q}$ is *correct* if given all $Q \in \mathcal{Q}$ and graphs G, it terminates and returns $Q(G)$. (b) A sequential incremental algorithm IncEval for $\mathcal{Q}$ is *correct* if given all $Q \in \mathcal{Q}$, graphs G, old output $Q(G)$ and updates ΔG to G, it computes changes ΔO to $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$. (c) We say that Assemble is *correct for* $\mathcal{Q}$ *w.r.t.* $\mathcal{P}$ if when GRAPE with PEval, IncEval and $\mathcal{P}$ terminates at superstep r_0, $\mathsf{Assemble}(Q(F_1[\bar{x}_1^{r_0}]), \ldots, Q(F_m[\bar{x}_m^{r_0}])) = Q(G)$, where $\bar{x}_i^{r_0}$ denotes the values of parameters $C_i.\bar{x}_i$ at round r_0. (d) We say that GRAPE *correctly parallelizes* a

PIE program ρ with a partition strategy $\mathcal{P}$ if for all queries $Q \in \mathcal{Q}$ and graphs G, GRAPE guarantees to reach a fixed point such that $\rho(Q, G) = Q(G)$.

As shown in [19], GRAPE correctly parallelizes a PIE program ρ for a graph computation problem $\mathcal{Q}$ if (a) its PEval and IncEval are correct sequential algorithms for $\mathcal{Q}$, and (b) Assemble correctly combines partial results, and (c) PEval and IncEval satisfy the following monotone condition: for all variables $x \in C_i.\bar{x}$, $i \in [1, m]$, (a) the values of x are computed from the active domain of G, and (b) there exists a partial order p_x on the values of x such that IncEval updates x in the order of p_x. That is, x draws values from a finite domain (condition (a) above), and x is updated "monotonically" following p_x (condition (b)).

It should be remarked that the monotonicity above is just a sufficient condition for GRAPE computations to converge, but it is not a necessary condition. Indeed, a variety of contracting conditions have been developed for fixpoint computation, *e.g.,* [10–12]. These conditions can be adapted to GRAPE convergence as well, in addition to the monotonic condition given above.

Moreover, it does not mean that only algorithms satisfying the monotonic condition can be parallelized in GRAPE. As will be seen shortly, any MapReduce algorithm can be migrated to GRAPE without extra complexity. Obviously not all MapReduce algorithms have the monotonicity. In other words, the monotonicity is just a condition under which one does not have to worry about convergence.

(3) Expressive power. The programming simplicity does not imply degradation in functionality of the existing systems. Following [46], we say that a parallel model $\mathcal{M}_1$ can *optimally simulate* model $\mathcal{M}_2$ if there exists a compilation algorithm that transforms any program with cost C on $\mathcal{M}_2$ to a program with cost $O(C)$ on $\mathcal{M}_1$. The cost includes computational cost and communication cost.

As shown in [19], GRAPE can optimally simulate MapReduce [13], BSP [45] and PRAM (Parallel Random Access Machine) [46]. That is, all algorithms in MapRedue, BSP or PRAM with n workers can be simulated by GRAPE using n workers with the same number of supersteps and memory cost. As a consequence, these algorithms can be migrated to GRAPE without increasing the complexity.

The result above aims to show the expressive power of GRAPE. In particular, graph computations that have effective (*e.g.,* bounded) incremental algorithms may be substantially accelerated by GRAPE. Nonetheless, for algorithms that make only one fragment active at a time, we do not expect that GRAPE speeds up their parallel computations. A particular example is Depth First Search (DFS), which is known to be hard to parallelize. While DFS can be parallelized by GRAPE, GRAPE may not make it more efficient than other platforms.

(4) Graph-level optimization. GRAPE naturally inherits all optimization strategies available for sequential graph algorithms, *e.g.,* indexing, compression and partitioning. Indeed, PEval and IncEval work on fragments, which are graphs themselves. Hence prior optimization strategies developed for sequential graph algorithms remain effective for PEval and IncEval. In contrast, these strategies are hard to implement for, *e.g.,* vertex-centric programs.

(5) Reducing redundant computations. GRAPE reduces the costs of iterative graph computations by using IncEval, to minimize unnecessary recomputations. We should remark that IncEval speeds up iterative computations by making use of prior partial results $Q(F_i)$ 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 [16], localizable incremental algorithms and relative boundedness [14].

(6) Compatibility. To simplify the discussion, we have focused on synchronous model BSP, when iterative computation is separated into supersteps, and messages from one superstep are only accessible in the next one. Our recent results have shown that the programming model of GRAPE remains intact under asynchronous parallel model (AP), when a worker has immediate access to incoming messages, and when fast workers can move ahead, without waiting for stragglers. Moreover, the convergence condition given above can be adapted to the asynchronous model. In other words, with GRAPE, it is no longer hard to write, debug and analyze parallel algorithms, no matter whether under BSP or AP.

3 Programming with GRAPE

We next show how GRAPE parallelizes familiar graph algorithms, by taking single-source shortest distance (SSSP), graph simulation (Sim), connected components (CC) and minimum spanning tree (MST) as examples. We parallelize these algorithms in Sections 3.1–3.4 under a vertex-cut partition. Taken together with the parallelization of [19] under edge-cut, these show that GRAPE programming works equally well under vertex-cut and edge-cut partition.

3.1 Graph Traversal

Consider $\mathcal{Q}$ denoting *the single source shortest path problem* (SSSP). It targets 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. SSSP is stated as follows.

- Input: A directed graph G as above, and a node s in G.
- Output: Distance $\mathsf{dist}(s, v)$ for all nodes v in G.

It is known that SSSP is in $O(|E| + |V|\mathsf{log}|V|)$ time [20].

For SSSP under vertex cut, GRAPE takes existing sequential (incremental) algorithms for SSSP as PEval and IncEval, just like GRAPE under edge-cut [19].

(1) PEval. As shown in Fig. 2, PEval (lines 1-11 of Fig. 2) is verbally identical to Dijsktra's algorithm [20], except that it declares the following (underlined):

(a) for each node $v \in V_i$, an integer variable $\mathsf{dist}(s, v)$, initially ∞ (except $\mathsf{dist}(s, s) = 0$); the candidate set C_i is the set $F_i.O$ of border nodes and the set of updated parameters is $C_i.\bar{x} = \{\mathsf{dist}(s, v) \mid v \in F_i.O\}$; and

Input: Fragment $F_i(V_i, E_i, L_i)$, source vertex s.
Output: A set $Q(F_i)$ consisting of current $\mathsf{dist}(s, v)$ for all $v \in V_i$.
Message preamble: /* candidate set C_i is $F_i.O$*/
    for each node $v \in V_i$, an integer variable $\mathsf{dist}(s, v)$;
1. initialize priority queue Que; $\mathsf{dist}(s, s) := 0$;
2. **for each** v in V_i **do**
3. **if** $v! = s$ **then** $\mathsf{dist}(s, v) := \infty$;
4. Que.addOrAdjust(s, $\mathsf{dist}(s, s)$);
5. **while** Que is not empty **do**
6. u := Que.pop() /* pop vertex with minimal distance */
7. **for each** child v of u **do** /* only if v has not been visited */
8. $alt := \mathsf{dist}(s, u) + L_i(u, v)$;
9. **if** $alt < \mathsf{dist}(s, v)$ **then**
10. $\mathsf{dist}(s, v) := alt$;
11. Que.addOrAdjust(v, $\mathsf{dist}(s, v)$);
12. $Q(F_i) := \{(v, \mathsf{dist}(s, v)) \mid v \in V_i\}$;
Message segment: $M_i := \{(v, \mathsf{dist}(s, v)) \mid v \in F_i.O\}$;
$f_{\mathsf{aggr}} = \mathsf{min}(\{\mathsf{dist}(s, v)\})$;

Fig. 2. PEval for SSSP

(b) an aggregate function f_{aggr} defined as min to resolve the conflicts: if multiple values are assigned to the same $\mathsf{dist}(s, v)$ by different workers, the smallest value is taken by the linear order on integers.

At the end of its process, PEval sends $C_i.\bar{x}$ to master P_0. At P_0, GRAPE maintains $\mathsf{dist}(s, v)$ for all nodes $v \in F_i.O$ ($i \in [1, m]$). Upon receiving messages from all workers, it takes the smallest value for $\mathsf{dist}(s, v)$ of each border node $v \in C_i.\bar{x}$. For each $i \in [1, m]$, it finds those variables with smaller $\mathsf{dist}(s, v)$ for $v \in F_i.O$, groups them into message M_i, and sends M_i to P_i.

(2) IncEval. As shown in Fig. 3, IncEval is the sequential incremental algorithm for SSSP developed in [39], in response to changed $\mathsf{dist}(s, v)$ for v in $F_i.O$ (here M_i includes changes to $\mathsf{dist}(s, v)$ for $v \in F_i.O$). Using a queue Que, it starts with M_i, propagates the changes to affected area, and updates the distances (see [40]). The partial result is now the set of revised distances (line 11). At the end of the process, the updated values of $C_i.\bar{x}$ are sent to the master as messages, where the aggregate function min is applied to resolve conflicts as in PEval.

Here IncEval is bounded. Following [39], it can be verified that its cost is determined by the size of "updates" $|M_i|$ and the changes to the output. This reduces the cost of iterative computation of SSSP (the While and For loops).

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

(4) Correctness. Termination is guaranteed since the values of update parameters are from a finite domain and are monotonically decreasing in the process. The correctness is assured since (a) the algorithms for PEval [20] and IncEval [40] are correct and (b) IncEval are monotonic by taking min as f_{aggr}.

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

```
1.  initialize priority queue Que;
2.  for each dist(s, v) in M_i do
3.    Que.addOrAdjust(v, 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 if v has not been visited*/
7.      alt := dist(s, u) + L_i(u, v);
8.      if alt < dist(s, v) then
9.        dist(s, v) := alt;
10.       Que.addOrAdjust(v, dist(s, v));
11. Q(F_i) := {(v, dist(s, v)) | v ∈ V_i};
```

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

Fig. 3. IncEval for SSSP

3.2 Graph Simulation

We next study graph simulation, which is commonly used in social media marketing [17] and social network analysis [15], among other things.

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

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 is 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 (u, u') in E_Q, there exists (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 [27], referred to as $Q(G)$. If G does not match Q, $Q(G)$ is the empty set. Moreover, it is known that $Q(G)$ can be computed in $O((|V_Q| + |E_Q|)(|V| + |E|))$ time [27, 15].

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 $Q(G)$.

GRAPE parallelizes Sim by adopting the sequential algorithm gsim for Sim developed in [27]. It uses the initialization of gsim as PEval to generate candidate matches $\mathsf{sim}(u)$ for each query node $u \in V_Q$; it then uses the main loop of gsim as IncEval, to refine $\mathsf{sim}(u)$ by recursively filtering out false positives in $\mathsf{sim}(u)$.

(1) PEval. As shown in Fig. 4, PEval adopts the initialization step of gsim. It sets C_i to $F_i.O$ and declares, for each query node $u \in V_q$ and data node v in fragment F_i, a status variable $\mathsf{cnt}_{(v,u)}$. Here $\mathsf{cnt}_{(v,u)}$ denotes the number of successors of v that are candidate matches of u in G, defined as $|\{w \mid w \in \mathsf{post}(v) \wedge w \in \mathsf{sim}(u)\}|$, where $\mathsf{post}(v)$ denotes the set of successors of v in G. It will be used by IncEval to filter out invalid candidate matches of u from $\mathsf{sim}(u)$. PEval initializes $\mathsf{sim}(u)$

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

Message preamble: /* candidate set C_i is $F_i.O$ */
  for each node u in V_Q and v in V_i, an integer variable $\mathsf{cnt}_{(v,u)} := 0$;

/* Initialize variable $\gamma(v) = \mathsf{true}$ if v has a successor in G, at loading time */
1. **for each** $u \in V_Q$ **do**
2. **if** $\mathsf{post}(u) = \emptyset$ **then**
3. $\mathsf{sim}(u) := \{v \in V_i \mid L_Q(u) = L_i(v)\}$;
4. **else** $\mathsf{sim}(u) := \{v \in V_i \mid L_Q(u) = L_i(v) \wedge \gamma(v) = true\}$;
5. **for each** $v \in V_i$ **do**
6. $\mathrm{cnt}_{(v,u)} = |\{w \in V_i \mid w \in \mathsf{post}'(v) \wedge w \in \mathsf{sim}(u)\}|$;
7. $Q(F_i) := \mathsf{sim}$;

Message segment: $M_i := \{\Delta\mathsf{cnt}_{(v,u)} \mid u \in V_Q \wedge v \in F_i.O\}$;
$f_{\mathsf{aggr}} = \mathsf{sum}\ (\Delta\mathsf{cnt}_{(v,u)})$;

Fig. 4. PEval for graph simulation

in the same way as sequential algorithm gsim (lines 1-4), except that it uses a Boolean variable $\gamma(v)$ to check candidate matches in C_i for $\mathsf{sim}(u)$, where $\gamma(v)$ is set to *true* if v has any successor in G and is initialized when loading the data graph G. It also initializes $\mathsf{cnt}_{(v,u)}$ as the number of "local" successors of v that are in fragment F_i and are candidate matches of u in F_i (lines 5-6). As will be seen shortly, we use counters $\mathsf{cnt}_{(v,u)}$ to determine invalid match candidates.

We take $F_i.O$ as C_i, and treat $C_i.\mathsf{cnt}$ as update parameters. After $\mathsf{cnt}_{(v,u)}$ is locally initialized in PEval, the set $C_i.\mathsf{cnt}$ is sent to master P_0. At P_0, upon receiving messages from all workers, the changes to $\mathsf{cnt}_{(v,u)}$ are aggregated using $f_{\mathsf{aggr}} = \mathsf{sum}$ to generate the global value of $\mathsf{cnt}_{(v,u)}$. GRAPE then groups these variables into message M_i and sends M_i to P_i.

(2) IncEval. As shown in Fig. 5, IncEval is a minor revision of gsim; it refines candidate matches (lines 1-14). In particular, it uses $\mathsf{cnt}_{(v,u)}$ to speedup the refinement: if $\mathsf{cnt}_{(v,u)} = 0$, then no children of v can match u, and hence v cannot match any query node u' that is a parent of u in Q. The counter cnt on v's parents is then updated, which is used to identify more false matches. More specifically, IncEval first updates $\mathsf{cnt}_{(v,u)}$ on border nodes, by applying changes to $\mathsf{cnt}_{(v,u)}$ in the message. For each (u, v), if $\mathsf{cnt}_{(v,u)} = 0$, then there is no match of u in $\mathsf{post}(v)$. Hence v cannot match any vertex in $\mathsf{pre}(u)$ in V_Q. After false match (u', v) is spotted, $\mathsf{cnt}_{(w,u')}$ is reduced by 1 for all w in $\mathsf{pre}(\mathsf{v})$. This is propagated through incoming edges iteratively, to identify more false matches.

Similar to the edge-cut version of IncEval in [16], one can verify that IncEval is *semi-bounded*: its cost is decided by the size of updates M_i and changes to the affected area necessarily checked by all incremental algorithms for Sim, rather than by F_i. This guarantees the efficiency of IncEval for graph simulation.

(3) Assemble 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.

Input: pattern Q, fragment F_i, partial result sim and message M_i.
Output: maximum match relation sim for Q and $F_i \oplus M_i$.

1. queue $\mathsf{remove} := \emptyset$;
2. **for each** $\Delta\mathsf{cnt}_{(v,u)}$ in M_i **do**
3. $\quad \mathsf{cnt}_{(v,u)} := \mathsf{cnt}_{(v,u)} + \Delta\mathsf{cnt}_{(v,u)}$;
4. **for each** $\mathsf{cnt}_{(v,u)}$ that is updated to 0 **do**
5. $\quad \mathsf{remove}(u) := \mathsf{remove}(u) \cup \{v\}$;
6. **while** there exists $u \in V_Q$ such that $\mathsf{remove}(u) \neq \emptyset$ **do**
7. $\quad$ **for each** $u' \in \mathsf{pre}(u)$ **do**
8. $\quad\quad$ **for each** $w \in \mathsf{remove}(u)$ **do**
9. $\quad\quad$ **if** $w \in \mathsf{sim}(u')$ **then**
10. $\quad\quad\quad \mathsf{sim}(u') := \mathsf{sim}(u') \setminus \{w\}$;
11. $\quad\quad\quad$ **for each** $w' \in \mathsf{pre}'(w)$ **do**
12. $\quad\quad\quad\quad$ decrease $\mathsf{cnt}_{(w',u')}$ by 1;
13. $\quad\quad\quad\quad$ **if** $\mathsf{cnt}_{(w',u')} = 0$ **then** $\mathsf{remove}(u') := \mathsf{remove}(u') \cup \{w'\}$;
14. $\quad\quad \mathsf{remove}(u') := \emptyset$;
15. $Q(F_i) := \mathsf{sim}$;

Message segment:
$M_i := \{\Delta\mathsf{cnt}_{(v,u)} \mid u \in V_Q, v \in F_i.O,\ \mathsf{cnt}_{(v,u)}\text{changed}\}$;

Fig. 5. IncEval for graph simulation

(4) Correctness of the GRAPE parallelization above is warranted by monotonic updates to $C_i.\mathsf{cnt}$ and by the correctness of sequential algorithm gsim [27]. More specifically, $\mathsf{cnt}_{(v,u)}$ is initially the maximum count of possible matches in $\mathsf{post}(v)$ with u after the process of PEval; it is monotonically reduced in the IncEval process, until it reaches the number of true matches in $\mathsf{post}(v)$ with u.

3.3 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 as follows.

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

The problem is known to be in $O(|G|)$ time [9].

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 $v.\mathsf{cid}$ for each node v in fragment F_i, initialized as its node id. As shown in Figure 6, PEval first uses a standard

Input: $F_i = (V_i, E_i, L_i)$.
Output: $Q(F_i)$ consisting of $v.\mathsf{cid}$ 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 $v.\mathsf{cid}$ initialized as v's id;
1. $\mathcal{CC} := \mathsf{DFS}(F_i)$; /* use DFS to find the set of local CCs */
2. **for each** local component $\mathsf{C} \in \mathcal{CC}$ **do**
3. add a new single root node v_r;
4. $v_r.\mathsf{cid} := \min\{v.\mathsf{cid} \mid v \in \mathsf{C}\}$;
5. **for each** node $v \in \mathsf{C}$ **do**
6. link v to v_r; $v.\mathsf{root} := v_r$; $v.\mathsf{cid} := v_r.\mathsf{cid}$;
7. $Q(F_i) := \{v.cid \mid v \in V_i\}$;
Message segment: $M_i := \{v.\mathsf{cid} \mid v \in F_i.O\}$;
$f_{\mathsf{aggr}} = \mathsf{min}(v.\mathsf{cid})$;

Fig. 6. PEval for CC

sequential traversal DFS (Depth-First Search) to compute the local connected components of F_i. For each local component C, (a) PEval creates a "root" node v_r carrying the minimum node id in C as $v_r.\mathsf{cid}$, and (b) links all the nodes in C to v_r, and sets their cid as $v_r.\mathsf{cid}$. These can be completed in one pass of the edges of F_i via DFS. At the end of process, PEval sends $\{v.\mathsf{cid} \mid v \in F_i.O\}$ to master P_0. The set consists of the update parameters at fragment F_i.

At master P_0, GRAPE maintains $v.\mathsf{cid}$ for each all $v \in F_i.O$ ($i \in [1, m]$). It updates $v.\mathsf{cid}$ by taking the smallest cid if multiple cids are received, by taking min as f_{aggr} in the message segment of PEval. It groups the nodes with updated cids into messages M_i, and sends M_i to worker P_i.

(2) IncEval incrementally updates the cids of the nodes in each fragment F_i upon receiving M_i, in parallel, as shown in Figure 7. Observe that message M_i sent to P_i consists of $v.\mathsf{cid}$ with updated (smaller) values. For each $v.\mathsf{cid}$ in M_i, IncEval finds the root v_r of v (line 3), and updates $v_r.\mathsf{cid}$ to the smaller $v.\mathsf{cid}$. IncEval then propagates the changes from every updated root node v_r to all nodes linked to v_r by changing their cids to $v_r.\mathsf{cid}$. At the end of the process, IncEval sends to master P_0 the updated cids of nodes in $F_i.O$ 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 that have the same cid in the same connected component, and returns all the connected components.

(4) Correctness. The process terminates as the cids of the nodes are monotonically decreasing by f_{aggr} until no changes can be made. Moreover, it correctly merges two local connected components by propagating the smaller 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** /* $v \in F_i.O$ */
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. $Q(F_i) := \{v.\mathsf{cid} \mid v \in V_i\}$;
Message segment: $M_i := \{v.\mathsf{cid} \mid v \in F_i.O,\ v.\mathsf{cid}$ decreased$\}$;

Fig. 7. IncEval for CC

3.4 Minimum Spanning Tree

Consider a connected undirected graph $G = (V, E, W)$, where for each edge $e = (v, v')$, $W(e)$ is a number specifying the cost to connect v and v'. A *spanning tree* T of G is a subgraph of G that is a tree (*i.e.,* an undirected graph in which any two nodes are connected by exactly one path), and includes all the vertices of V. A *minimum weighted spanning tree* T of G is a spanning tree of G such that the total weight $w(T) = \Sigma_{e \in T} W(e)$ is minimized. To simplify the discussion, we assume that each edge e in G has a distinct cost $W(e)$. It is known that a unique MST exists in such a graph G [21]. The MST problem is stated as follows.

- Input: A graph $G = (V, E, W)$ as described above.
- Output: The minimum spanning tree MST of G.

It is known that MST is in $O(|E| + |V| \log |V|)$ time.

GRAPE parallelizes MST as follows. It combines Prim's sequential MST algorithm [37] and Borůvka's sequential algorithm [35] as PEval: it adopts Prim's algorithm to generate initial partial MSTs (*i.e.,* sub-trees of the final MST), and uses Borůvka's algorithm to generate messages. For IncEval, it employs Borůvka's algorithm alone to iteratively connect those partial MSTs, forming the final MST. It should be remarked that marrying Prim's and Borůvka's MST algorithms is a common practice for efficiently computing MST in parallel (see *e.g.,* [8]).

(1) PEval. As shown in Fig. 8, PEval takes $F_i.O$ as C_i and declares, for each node u in C_i, a triple $u.\mathsf{m}(u, \mathsf{tid}, \mathsf{e})$ initialized as $(u, u.\mathsf{id}, nil)$, where u.id is the node id of u. It generates a set $\mathcal{T}$ of partial MSTs of F_i *excluding* border nodes in $F_i.O$, using Prim$'$, a minor revision of Prim's algorithm to ensure such partial MSTs are guaranteed to be sub-trees of the global MST of G (line 1). It then treats each border node in $F_i.O$ as a partial MST and includes them in $\mathcal{T}$ as well (line 2). For each partial MST T in $\mathcal{T}$, it maintains an index for T, denoted by T.tid, using the minimum node id of nodes in T (line 3). Such tids will be used to combine partial MSTs by IncEval. For convenience, we also write T_u as the unique partial MST that contains u and u.tid as T_u.tid.

Input: Fragment $F_i = (V_i, E_i, W_i)$.
Output: $Q(F_i)$ consisting of all edges in partial MSTs.
Message preamble: /* Candidate set C_i is $F_i.O$ */
for each node $u \in C_i$, a triple $u.\mathsf{m}(u, \mathsf{tid}, \mathsf{e})$ is initialized as $(u, u.\mathsf{id}, \mathit{nil})$.

1. $\mathcal{T} := \mathsf{Prim}'(F_i')$; /* F_i' denotes F_i excluding border nodes */
2. **for each** $v \in F_i.O$ **do** add $T_v := (\{v\}, \emptyset)$ as a partial MST to $\mathcal{T}$;
3. **for each** partial MST $T \in \mathcal{T}$ **do** $T.\mathsf{tid} := \min_{v \in T} v.\mathsf{id}$;
4. **for each** $T \in \mathcal{T}$ **do**
5. $e := \arg\min_{(u,v) \in E_i, u \in T, v \notin T, u.\mathsf{tid} \neq v.\mathsf{tid}} W(u, v)$; /* assume $e = (u, v)$ */
6. **if** $u \notin F_i.O$ **then** add (u, v) to T_u; update $T_u.\mathsf{tid}$ to $\min(T_u.\mathsf{tid}, v.\mathsf{tid})$;
7. **else** $u.\mathsf{m} := (u, u.\mathsf{tid}, (u, v))$; /* generate message for u */
8. **for** each u in $F_i.O$ whose message has not been generated **do**
9. $u.\mathsf{m} := (u, u.\mathsf{tid}, \mathit{nil})$; /* generate message for u */
10. $Q(F_i) := \mathcal{T}$;

Message segment: $M_i := \{u.\mathsf{m} \mid u \in C_i\}$;
$f_{\mathsf{aggr}} = (u, \min_{u.\mathsf{m}[\mathsf{tid}]} u.\mathsf{m}[\mathsf{tid}], \arg\min_{u.\mathsf{m}[e]} W(u.\mathsf{m}[\mathsf{e}]))$

Fig. 8. PEval for MST

It then generates messages for border nodes by using Borůvka's algorithm (lines 4-7). Following Borůvka's algorithm, it treats each MST in $\mathcal{T}$ as a "virtual" node and expands each "virtual" node, say, MST $T \in \mathcal{T}$, with the edge e adjacent to T that has the minimum weight among all edges connecting T and some other MSTs in $\mathcal{T}$ (line 5). It merges the new edge $e = (u, v)$ into T and updates the MST index of T accordingly if $u \in T$ and u is not a border node (line 6). When u is a border node, PEval cannot decide whether the local minimum weighted adjacent edge e is the global minimum edge in G for u, and hence PEval encodes it together with $u.\mathsf{tid}$ in the message for u (line 7). For those border nodes whose messages have not been generated in this way, a default message without the adjacent edge (*i.e., nil*) is generated (lines 8-9).

The message for each border node u on all fragments will be gathered at master P_0. The minimum tid and the global minimum adjacent edge for u will be deduced by f_{aggr} specified in the message segment of PEval.

(2) IncEval. Following Borůvka's algorithm, IncEval iteratively merges partial MSTs in $\mathcal{T}$. Since each message $(u_0, \mathsf{tid}_0, e_0)$ for border node u_0 tells us that (a) the minimum tid for partial MSTs containing u_0 on all fragments is tid_0, and (b) the global minimum weighted adjacent edge for expanding u_0 is e_0, IncEval connects the partial MSTs upon receiving messages in two steps. It first updates tids of all local MSTs with tid_0 so that all MSTs containing u_0 on all fragments are assigned with the same tid (line 1-2). It further merges partial MSTs on each fragment via the aggregated global minimum weighted adjacent edge e_0 in the message, and updates the tid accordingly (lines 3-5). It then expands each updated MST in $\mathcal{T}$ and generates messages for border nodes in the same way as PEval (lines 6-11). Note that by only connecting MSTs with distinct tids (line 4) and using tids to choose minimum weighted adjacent edges (line 8), IncEval ensures that no cycle is produced in the entire process of IncEval iterations.

Input: a set of partial MST $\mathcal{T}$, message M_i.
Output: $Q(F_i \oplus M_i)$.

1. **for each** $m = (u_0, \mathsf{tid}_0, e_0) \in M_i$ **do** /* update tid's */
2. $T_{u_0}.\mathsf{tid} := \min(T_{u_0}.\mathsf{tid}, \mathsf{tid}_0)$;
3. **for each** $m = (u_0, \mathsf{tid}_0, e_0 = (u, v)) \in M_i$ **do** /* merge local MSTs */
4. **if** e_0 is in F_i and $T_u.\mathsf{tid} \neq T_v.\mathsf{tid}$ **then**
5. merge T_u and T_v in $\mathcal{T}$ (denoted by T'); $T'.\mathsf{tid} := \min(T_u.\mathsf{tid}, T_v.\mathsf{tid})$;
6. **for each** $T \in \mathcal{T}$ **do** /* generate messages */
7. **if** there exists $u \in T$ such that $v \notin T$, $u.\mathsf{tid} \neq v.\mathsf{tid}$ and $(u, v) \in E_i$ **then**
8. $e := \arg\min_{(u,v)\in E_i, u\in T, v\notin T, u.\mathsf{tid}\neq v.\mathsf{tid}} W(u, v)$; /* assume $e = (u, v)$ */
9. **for each** border node u in T **do** $u.m := (u, u.\mathsf{tid}, (u, v))$;
10. **for** each u in $F_i.O$ whose message has not been generated **do**
11. **if** $u.\mathsf{tid}$ has been changed **then** $u.m := (u, u.\mathsf{tid}, nil)$;
12. $Q(F_i \oplus M_i) := \mathcal{T}$;

Message segment: $M_i := \{u.m \mid u \in C_i\}$;

Fig. 9. IncEval for MST

One can verify that the incremental IncEval is *bounded*: it takes $O(|M_i|)$ time to update the tids and merge partial MSTs, and $O(|\mathsf{AFF}|)$ time to generate messages, where AFF consists of border nodes in $F_i.O$ with changed tids, and hence, $|\mathsf{AFF}|$ is bounded by the changes of the output of IncEval.

(3) Assemble simply merges edges in the partial MSTs from all fragments and returns all the edges, as the final MST.

(4) Correctness. The process terminates as the tid's and the weights of selected adjacent edges of border nodes for connecting MSTs are monotonically decreasing by f_{aggr} until no changes can be made. Its correctness follows from the following: (a) by Prim's algorithm, PEval correctly computes MSTs of the subgraph consisting of inner edges in each fragment; and (b) by Borůvka's algorithm, IncEval correctly merges the MSTs computed by PEval into the final MST of G.

4 Experimental Study

Using real-life and synthetic graphs, we next evaluate the performance of GRAPE for its (1) efficiency, (2) communication cost, and (3) scale-up. We compared the performance of GRAPE with that of three state-of-the-art graph systems: (a) Giraph [3] and synchronized $\mathsf{GraphLab}_{\mathsf{sync}}$ (PowerGraph [22]) under the bulk synchronous parallel model (BSP), and (b) asynchronized $\mathsf{GraphLab}_{\mathsf{async}}$ under asynchronous model (AP) without global synchronization, when a worker has immediate access to messages, allowing fast workers to move ahead [3].

Experimental setting. We start with our settings.

Graphs. We used four real-life graphs of different types, including DBpedia [1], traffic [4], Friendster [2] and UKWeb [5], as shown in Table 1, such that each

[3] $\mathsf{GraphLab}_{\mathsf{sync}}$ and $\mathsf{GraphLab}_{\mathsf{async}}$ run different modes of GraphLab (PowerGraph).

	Graph Type	$\mid V \mid$	$\mid E \mid$	Algorithm
DBpedia	knowledge graph	28 million	33.4 million	for Sim, MST
traffic	road network	23 million	58 million	for SSSP, CC, MST
Friendster	social network	65 million	1.8 billion	for SSSP, CC, Sim
UKWeb	Web graph	133 million	5 billion	for SSSP, CC, Sim

Table 1. Real-life Graph Information

algorithm was evaluated with at least two real-life graphs. We randomly assigned weights to traffic, Friendster and UKWeb for testing SSSP and MST, and assigned up to 50 node labels to unlabeled Friendster for testing Sim.

To test the scalability of GRAPE, we developed a generator to produce graphs $G = (V, E, L)$ controlled by the number $|V|$ of nodes (up to 0.4 billion) and edges $|E|$ (up to 20 billion), with L drawn from an alphabet of 100 labels.

Queries. We randomly generated queries for SSSP and Sim. (a) For SSSP, we sampled 10 source nodes from each graph G used, such that each source node can reach 90% nodes in G. We constructed an SSSP query for each node. (b) We generated 20 pattern queries Q for Sim, controlled by $|Q| = (|V_Q|, |E_Q|, L_Q)$, where $|V_Q|$ and $|E_Q|$ denote the number of nodes and edges, respectively, using labels L_Q drawn from the graphs experimented with.

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, Giraph and GraphLab 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.

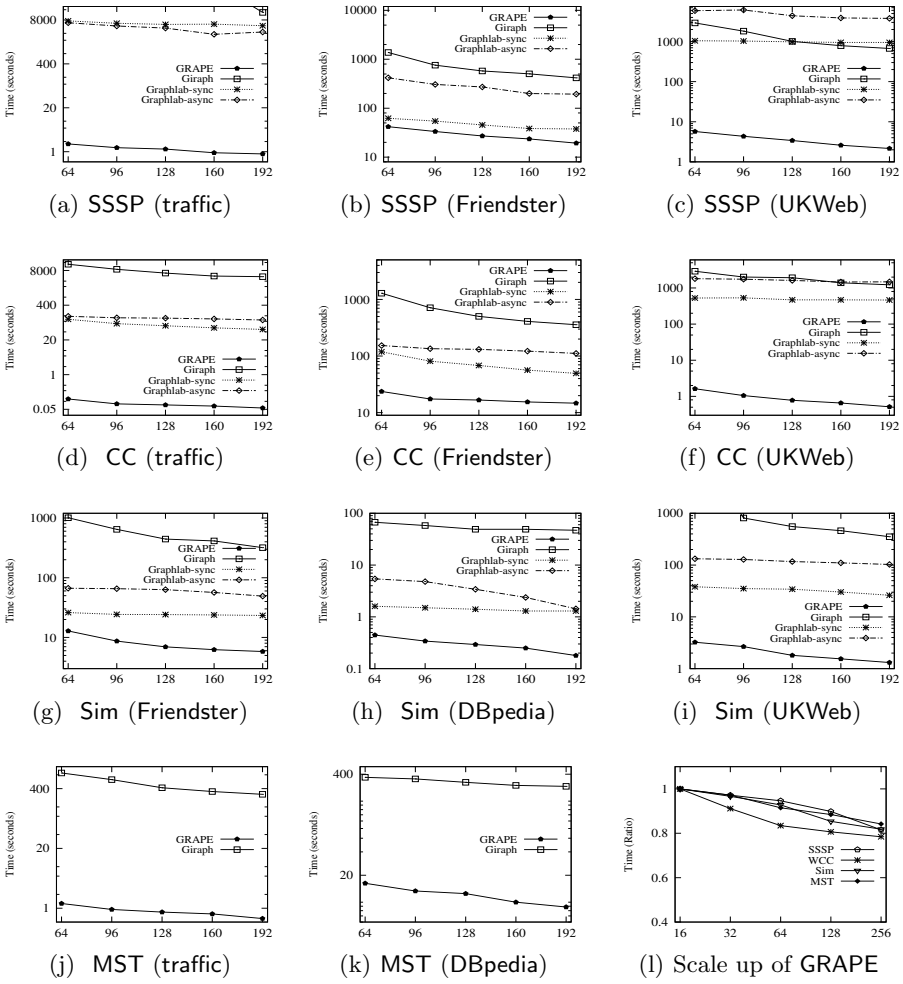
Algorithms. We evaluated the PIE programs developed in Section 3 for SSSP, CC, Sim and MST on GRAPE. We used "default" code provided by Giraph and GraphLab when it was available. Otherwise, we made our best efforts to develop and optimize the algorithms on the competitor systems if possible (see below).

We used the degree-based hashing (DBH) [48] algorithm to partition graphs as the default graph partition strategy. It was a state-of-the-art vertex-cut graph partition strategy. To improve the locality of partition, we first applied Xtra-PuLP [43] to graphs, and then took its output as the input of DBH.

We deployed these systems on an HPC cluster, and used servers with 16 cores of 2.40GHz, 128GB memory. Each core is treated as a worker. We ran each experiment 5 times, and the average of results is reported here.

Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluated the efficiency of GRAPE by varying the number n of workers used, from 64 to 192, compared with Giraph, $\text{GraphLab}_{\text{sync}}$ and $\text{GraphLab}_{\text{async}}$ when possible. For SSSP and CC, we experimented with real-life graphs traffic, Friendster and UKWeb; for Sim, we used Friendster, DBpedia and UKWeb; and for MST, we used traffic and DBpedia, based on applications of these algorithms in transportation networks, knowledge bases, Web and social graph analysis. We do not report times that exceeded 20000s.



(a) SSSP (traffic) (b) SSSP (Friendster) (c) SSSP (UKWeb)

(d) CC (traffic) (e) CC (Friendster) (f) CC (UKWeb)

(g) Sim (Friendster) (h) Sim (DBpedia) (i) Sim (UKWeb)

(j) MST (traffic) (k) MST (DBpedia) (l) Scale up of GRAPE

Fig. 10. Performance Evaluation

(1) SSSP. We compared the efficiency of GRAPE for SSSP with that of Giraph, $\overline{\text{GraphLab}_{\text{sync}}}$ and $\text{GraphLab}_{\text{async}}$ by using "default" code provided by these systems. The results are reported in Figures 10(a)-10(c), which tell us the following.

(a) GRAPE consistently outperforms these systems. Over traffic (resp. Friendster and UKWeb), it is on average 15449 (resp. 21.5 and 310.8), 6261 (resp. 2.0 and 438.5) and 4026.7 (resp. 10.0 and 1749) times faster than Giraph, $\text{GraphLab}_{\text{sync}}$ and $\text{GraphLab}_{\text{async}}$, respectively. Note that the improvement of GRAPE on traffic is far more significant than on Friendster and UKWeb. This is because Giraph, $\text{GraphLab}_{\text{sync}}$ and $\text{GraphLab}_{\text{async}}$ adopt vertex-centric programming, which takes more "rounds" to converge on graphs with larger diameters. For instance, on Friendster, Giraph takes 36 rounds to converge, similarly for $\text{GraphLab}_{\text{sync}}$, compared with 21 rounds by GRAPE. In contrast, on traffic, a graph with larger diameter, Giraph and $\text{GraphLab}_{\text{sync}}$ take 10789 and 10778 rounds, respectively,

while GRAPE takes 31 rounds. These verify the efficiency of GRAPE as a parallel engine for graph traversal algorithms such as SSSP.

(b) GRAPE is on average 2.0, 2.2 and 2.6 times faster on traffic, Friendster and UKWeb, respectively, when the number n of workers varies from 64 to 192. That is, the more workers are used, the faster SSSP runs on GRAPE.

(2) CC. We evaluated GRAPE versus Giraph, GraphLab$_{\text{sync}}$ and GraphLab$_{\text{async}}$ using their "default" code for CC. As shown in Figures 10(d)-10(f) over traffic, Friendster and UKWeb, respectively, (a) GRAPE substantially outperforms these systems. When $n = 192$, GRAPE is on average 28787, 10960 and 3957 times faster than Giraph, GraphLab$_{\text{sync}}$ and GraphLab$_{\text{async}}$ over these real-life graphs, respectively. (b) GRAPE scales well with the number of workers used: it is on average 2.3 times faster when n varies from 64 to 192.

(3) Sim. Fixing $|Q| = (6, 10)$, *i.e.,* patterns Q with 6 nodes and 10 edges, we evaluated GRAPE versus Giraph, GraphLab$_{\text{sync}}$ and GraphLab$_{\text{async}}$ for Sim. We developed Sim algorithms for the other platforms with our best efforts since neither Giraph nor GraphLab provides code for Sim. As shown in Figures 10(g)-10(h) over Friendster, DBpedia and UKWeb, respectively, (a) GRAPE outperforms other systems. When $n = 192$, GRAPE is on average 195, 10.5 and 36.0 times faster than Giraph, GraphLab$_{\text{sync}}$ and GraphLab$_{\text{async}}$ over the three graphs, respectively. (b) On average GRAPE is 2.4 times faster when n varies from 64 to 192.

(4) MST. We evaluated the efficiency of GRAPE for MST versus Giraph, with code for MST from [25]. We did not compare with GraphLab$_{\text{sync}}$ and GraphLab$_{\text{async}}$ since as observed in [26], MST "cannot be implemented efficiently on GraphLab because GraphLab does not fully support graph mutations", *e.g.,* deletions of edges and vertices; such mutations are needed for an efficient implementation of MST. As shown in Figures 10(j)-10(k) over traffic and DBpedia, respectively, (a) GRAPE is on average 502.7 (resp. 35.75) times faster than Giraph on traffic (resp. DBpedia), when $n = 192$. (b) GRAPE is on average 2.1 times faster when n is increased from 64 to 192, *i.e.,* GRAPE makes good use of parallelism.

Exp-2: Communication. We next report the communication costs of the systems. Different systems measure communication costs in different ways because each system makes use of its own implementation of message blocks and protocols [26]. For a fair comparison, we monitored the system file /proc/net/dev to report total bytes of message sent by each machine, following the practice of [26]. This metric reveals consistent results with better insights.

The communication costs over real-life graphs are reported in Table 2, when 192 workers were used. These results tell us the following. For all these algorithms, GRAPE incurs less communication costs than the other systems. On average GRAPE ships 3.9%, 3.5%, and 1.0% of data shipped by Giraph, GraphLab$_{\text{sync}}$ and GraphLab$_{\text{async}}$ for SSSP, 4.9%, 3.6% and 2.4% for CC, and 2.8%, 0.38% and 0.41% for Sim, respectively. For MST, the communication cost of GRAPE accounts for 20.5% of Giraph. In particular, GRAPE ships only 0.2%, 0.0003% and

	Giraph	GraphLab$_{\text{sync}}$	GraphLab$_{\text{async}}$	GRAPE
SSSP				
traffic	1426920	4016909	4548842	1.2
Friendster	101758	112673	377529	11840
UKWeb	89015	297179	1514413	152.6
CC				
traffic	61419	266594	579265	1.66
Friendster	74864	100087	227475	11000
UKWeb	227754	202706	810039	112.9
Sim				
Friendster	15901	114311	10149	1182
DBpedia	871	7213	12990	7.8
UKWeb	24158	222290	310658	4.3
MST				
traffic	9300	/	/	29.6
DBpedia	2701	/	/	1119

Table 2. Communication cost (MB)

0.00016% of data shipped by Giraph, GraphLab$_{\text{sync}}$ and GraphLab$_{\text{async}}$ on traffic. This is because GRAPE converge in far less rounds than vertex-centric systems. Among other things, GRAPE reduces communication costs by employing incremental IncEval, which ships only changed values of update parameters.

Exp-3 Scale-up of GRAPE. As observed in [34], the speed-up of a distributed system may degrade when using more workers. Thus we evaluated the scale-up of GRAPE, which measures the degradation of speed-up when both the size of graph $G = (|V|, |E|)$ and the number n of workers increase proportionally. We varied n from 16 to 256, and for each n, deployed GRAPE over a synthetic graph of size varied from $(25M, 1.25B)$ to $(0.4B, 20B)$, proportional to n.

As reported in Figure 10(l) for SSSP, CC, Sim and MST, respectively, GRAPE preserves a reasonable scale-up, all above 0.8. We did not test with single-thread since many of the graphs are too large to fit in a single machine.

Summary. From the experimental study we find the following. (1) GRAPE consistently outperforms the state-of-the-art systems. Over real-life graphs and with 192 workers, compared to Giraph, GraphLab$_{\text{sync}}$ and GraphLab$_{\text{async}}$, GRAPE is on average (a) 5260, 2233 and 1940 times faster for SSSP, (b) 28787, 10960 and 3957 times faster for CC, and (c) 195, 10.5 and 36.0 times faster for Sim, respectively. It is 7187 times faster than Giraph for MST (as remarked earlier, GraphLab does not efficiently support MST). (2) GRAPE speeds up SSSP, CC, Sim and MST on average 2.3, 2.3, 2.4 and 2.1 times, respectively, when the number of workers n varies from 64 to 192. (3) On average, its communication costs account for 2.8%, 3.7%, 1.2% and 20.5% of the other systems for SSSP, CC, Sim and MST, respectively. (4) GRAPE has a reasonable scale-up.

These results are consistent with their counterparts reported in [19] under edge-cut partition. Compared to GRAPE under edge-cut partition, GRAPE under vertex-cut is on average 0.91 times slower for SSSP but is 1.21 times faster for CC. It incurs 79% and 56% of the communication cost under edge-cut for SSSP and CC, respectively. That is, GRAPE has comparable performance under vertex-cut and edge-cut for SSSP and CC. The PIE program for Sim under vertex-cut (Section 3.2) is slightly different from its counterpart under edge-cut [19]. It employs a different version of IncEval, which has to synchronize the status of border nodes. As a result, the PIE program for SIM under vertex-cut is 0.75 times slower and incurs on average 4 times more communications cost than its edge-cut counterpart. As remarked earlier, MST was not studied in [19].

5 Concluding Remarks

The main objective of GRAPE is to simplify parallel programming for graph computations, from think parallel to think sequential. It allows users to devise existing sequential graph algorithms (with declarations of update parameters and an aggregate function; see Section 2.2), and parallelizes the computation across a cluster of machines. It reduces the total cost of ownership and makes parallel graph computations accessible to companies that cannot afford experienced developers who are able to write, debug and analyze parallel graph algorithms. Moreover, GRAPE guarantees to converge at correct answers under a general condition as long as it is provided with correct single-machine graph algorithms, and it inherits optimization strategies developed for sequential graph algorithms.

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

GRAPE also works well for other applications. We have also carried out PoC at a company that provides big data services, and at one of the largest telecommunication equipment and service companies in the world. The results are consistent and very promising: GRAPE is able to perform a number of tasks that are not supported by the state-of-the-art graph systems, and for jobs that can also be run at other systems, it substantially outperforms the existing systems.

To the best of our knowledge, GRAPE is the first system that is able to parallelize existing sequential graph algorithms as a whole, without recasting the algorithms into a new model. Prior work on automated parallelization has focused on the instruction or operator level [41, 36] by breaking dependencies via symbolic and automate analyses. There has also been work at a data partition level [51], to perform multi-level partition ("parallel abstraction") and adapt locality-optimized access to different parallel abstraction. In contrast, GRAPE does not require users to revise the logic of the existing algorithms. It makes parallel computation accessible to end users, while [41, 36, 51] target experienced developers of parallel algorithms. There have also been tools for translating imperative code to MapReduce, *e.g.,* word count [38]. GRAPE advocates a different approach, by parallelizing the runs of sequential graph algorithms to benefit from data-partitioned parallelism, without translating the algorithms.

This paper extends [19] in the following. (a) We develop PIE algorithms for SSSP, CC and Sim under vertex-cut, demonstrating the adaptability of GRAPE to vertex-cut from edge-cut [19]. (b) We provide a new PIE algorithm for MST. (c) We conduct experiments using larger graphs, and demonstrate the performance of GRAPE under vertex-cut compared to its counterpart under edge-cut [19].

As a topic for future work, we are developing a new parallel model that subsumes BSP, AP and SSP (Stale Synchronous Parallel model [28] for machine learning with parameter servers [31, 49]) as special cases. We are currently extending GRAPE to support the new model such that it is able to automatically switch among these models at different stages in a single execution, to optimize performance. Another topic is to support streaming updates when answering continuous queries for, *e.g.,* fraud detection, beyond static graphs assumed by existing graph systems. GRAPE is well positioned to accomplish this given that incremental computation is built in its parallel computation model.

Acknowledgments. The paper is a tribute to Professor Chaochen Zhou, who took Fan as an MSc student 30 years ago, despite pressure from a powerful person, whom Fan confronted to get justice done for his late former MSc adviser.

The authors are supported in part by 973 Program 2014CB340302, ERC 652976, EPSRC EP/M025268/1, NSFC 61421003, Beijing Advanced Innovation Center for Big Data and Brain Computing, Shenzhen Peacock Program 1105100030834361, and Joint Research Lab between Edinburgh and Huawei.

References

1. DBpedia. *http://wiki.dbpedia.org/Datasets*.
2. Friendster. *https://snap.stanford.edu/data/com-Friendster.html*.
3. Giraph. *http://giraph.apache.org/*.
4. Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml*.
5. UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/*, 2006.
6. U. A. Acar. *Self-Adjusting Computation*. PhD thesis, CMU, 2005.

7. K. Andreev and H. Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.
8. D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J. Parallel Distrib. Comput.*, 66(11):1366–1378, 2006.
9. J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2008.
10. G. M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25(2):226–244, 1978.
11. D. P. Bertsekas. Distributed asynchronous computation of fixed points. *Math. Program.*, 27(1):107–120, 1983.
12. D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, 1969.
13. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.
14. W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.
15. W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractability to polynomial time. In *PVLDB*, 2010.
16. W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.
17. W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns. *PVLDB*, 8(12):1502–1513, 2015.
18. W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang. GRAPE: Parallelizing sequential graph computations. *PVLDB*, 10(12):1889–1892, 2017.
19. W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, B. Zhang, Z. Zheng, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *SIGMOD*, 2017.
20. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.
21. R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *TOPLAS*, 5(1):66–77, 1983.
22. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USENIX*, 2012.
23. J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.
24. I. Grujic, S. Bogdanovic-Dinic, and L. Stoimenov. Collecting and analyzing data from E-Government Facebook pages. In *ICT Innovations*, 2014.
25. M. Han and K. Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.
26. M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *VLDB*, 7(12), 2014.
27. M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
28. Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.
29. N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3), 1996.
30. M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.

31. M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. Andersen, and A. Smola. Parameter server for distributed machine learning. In *NIPS workshop on Big Learning*, 2013.
32. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.
33. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.
34. F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what cost? In *HotOS*, 2015.
35. J. Nesetril, E. Milková, and H. Nesetrilová. Otakar boruvka on minimum spanning tree problem. *Discrete Mathematics*, 233(1-3):3–36, 2001.
36. K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, 2011.
37. R. C. Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6), 1957.
38. C. Radoi, S. J. Fink, R. M. Rabbah, and M. Sridharan. Translating imperative code to MapReduce. In *OOPSLA*, 2014.
39. G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
40. G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.
41. V. Raychev, M. Musuvathi, and T. Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *SOSP*, 2015.
42. B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, 2013.
43. G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, 2017.
44. Y. Tian, A. Balmin, S. A. Corsten, and J. M. Shirish Tatikonda. From "think like a vertex" to "think like a graph". *PVLDB*, 7(7):193–204, 2013.
45. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
46. L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Vol A*. 1990.
47. G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.
48. C. Xie, L. Yan, W.-J. Li, and Z. Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In *NIPS*, 2014.
49. E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, June 2015.
50. D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.
51. Y. Zhou, L. Liu, K. Lee, C. Pu, and Q. Zhang. Fast iterative graph computation with resource aware graph parallel abstractions. In *HPDC*, 2015.