SCIENCE CHINA
Information Sciences

• POSITION PAPER •

October 2020, Vol. 63 203101:1–203101:21
https://doi.org/10.1007/s11432-020-2952-7

Graph algorithms: parallelization and scalability

Wenfei FAN[1,2,3]*, Kun HE[2,4], Qian LI[2,4] & Yue WANG[2]

[1]*School of Informatics, University of Edinburgh, Edinburgh* EH8 9AB*, UK;*
[2]*Shenzhen Institute of Computing Sciences, Shenzhen University, Shenzhen* 518060*, China;*
[3]*Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing* 100191*, China;*
[4]*Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen University, Shenzhen* 518060*, China*

Received 1 April 2020/Revised 9 May 2020/Accepted 10 June 2020/Published online 21 September 2020

Abstract For computations on large-scale graphs, one often resorts to parallel algorithms. However, parallel algorithms are difficult to write, debug and analyze. Worse still, it is difficult to make algorithms parallelly scalable, such that the more machines are used, the faster the algorithms run. Indeed, it is not yet known whether any PTIME computational problems admit parallelly scalable algorithms on shared-nothing systems. Is it possible to parallelize sequential graph algorithms and guarantee convergence at the correct results as long as the sequential algorithms are correct? Moreover, does a PTIME parallelly scalable problem exist on shared-nothing systems? This position paper answers both questions in the affirmative.

Keywords parallelization, parallel scalability, PTIME problems, graph algorithms, shared-nothing systems

Citation Fan W F, He K, Li Q, et al. Graph algorithms: parallelization and scalability. Sci China Inf Sci, 2020, 63(10): 203101, https://doi.org/10.1007/s11432-020-2952-7

1 Introduction

When processing large-scale graphs, parallel computations are often a must. Several parallel systems have been developed for graph computations, e.g., Pregel [1], PowerGraph [2], and Giraph++ [3]. However, to make effective use of parallel graph computations, at least two issues have to be addressed.

The first issue concerns the programming models of parallel graph systems. Parallel algorithms are difficult to write, debug, and analyze [4]. On the one hand, a large number of sequential (single-machine) graph algorithms are already in place. On the other hand, to use Pregel, for instance, one has to "think like a vertex" and recast the existing algorithms into a vertex-centric model; similarly when programming with other parallel graph systems. The recasting is nontrivial for people who are not very familiar with parallel models. This makes these parallel systems a privilege for experienced users.

The second issue involves the effectiveness of parallel algorithms. When we turn to a parallel graph system, we assume that when provided with more processors (computing nodes), we can make effective use of the additional resources and speed up the execution of our algorithms. However, is this assumption always correct? Parallel systems typically adopt the shared-nothing architecture nowadays. On such a system, the use of more processors is often accompanied by heavier communication among these processors. As a consequence, while the use of more processors may distribute the computational cost, the parallel runtime of our algorithms may not necessarily be reduced. For instance, algorithms for single-source shortest path are "essentially not scalable with an increasing number of machines" [5].

These concerns highlight the need for studying the following problems.

* Corresponding author (email: wenfei@inf.ed.ac.uk)

Parallelization. Is it possible to parallelize existing sequential graph algorithms? That is, we want a system such that we can "plug" sequential graph algorithms into it as a whole (subject to minor changes), and it parallelizes the computation across a cluster of processors. Moreover, the parallelized computation (a) should converge at the correct results under a generic condition, as long as the sequential algorithms plugged in are correct; and (b) should not degrade the performance of the existing parallel systems.

Intuitively, parallelization allows us to reuse existing sequential algorithms for parallel computations without making substantial changes, such that we can "think sequential" instead of "think parallel". This would make parallel graph computations accessible to a large group of users who know conventional graph algorithms covered in undergraduate textbooks, and reduce the total cost of ownership (TCO).

Parallel scalability. How should we characterize the effectiveness of parallel algorithms? A natural criterion is parallel scalability [6], which measures speedup over sequential algorithms by parallelization, such that the more processors we use, the faster the parallel algorithms run. Intuitively, a parallelly scalable algorithm is able to scale with big data by adding processors when required. Of course, parallel scalability is not the only criterion for evaluating parallel algorithms. However, if an algorithm is not parallelly scalable, the chances are that it may not be able to cope with real-life graphs when they grow very large.

Parallel scalability is difficult to achieve, especially on shared-nothing systems. To the best of our knowledge, parallelly scalable algorithms on such systems are only available for intractable problems such as subgraph isomorphism [7–9]. When it comes to tractable (PTIME, polynomial-time computable) problems, no parallelly scalable algorithm is yet available on shared-nothing systems. Moreover, it is known that for graph simulation (a quadratic-time problem), parallelly scalable algorithms are beyond reach [10]. This suggests that parallel scalability requests a departure from the polynomial hierarchy in classical computational complexity theory [11]. An immediate question is, on shared-nothing systems, whether there exists any PTIME computational problem that admits parallelly scalable algorithms at all.

Contributions & organization. This position paper provides an informal overview of our latest efforts to the above problems. We present GRAPE [12], a recent graph system that supports parallelization. We also report new results on the parallel scalability of PTIME computational problems.

(1) *Parallelization* (Section 2). We present GRAPE, a parallel graph engine [12], based on a fixpoint model with partial evaluation and incremental computation. It adopts a shared-nothing architecture in which each processor runs a sequential algorithm on a fragment of a partitioned graph and communicates with other processors via message passing. It supports a simple programming model that takes existing sequential graph algorithms, and parallelizes their computation without the need for revising their logic. Better still, the computation guarantees to converge at the correct results under a monotone condition, as long as the sequential algorithms are correct. As shown in [12,13], besides its programming simplicity, GRAPE performs better than or is at least comparable to the state-of-the-art parallel graph systems.

(2) *Parallel scalability* (Section 3). We formalize the notion of parallel scalability following [6] by taking the sequential complexity of single-machine algorithms as a yardstick. We identify a simple sufficient condition for determining whether a certain graph algorithm is parallelly scalable on shared-nothing systems, by simulating existing efficient algorithms developed for shared-memory systems.

(3) *Parallelly scalable algorithms* (Section 4). Based on the identified condition, we show that parallel scalability is within the reach of PTIME computational problems. As constructive proofs, we develop parallel algorithms for graph connectivity and minimum spanning trees, two well-known PTIME problems in graph theory. We show that these algorithms are parallelly scalable under the GRAPE model. To the best of our knowledge, these are the first PTIME algorithms with provable parallel scalability.

(4) *Open problems* (Section 5). This position paper is by no means a survey of the development in this area. It rather aims to incite interest in this subject. Hence we identify open issues for future work.

Related work. We characterize the related work as follows.

Parallelization. Several parallel graph systems are already in place (see [14] for a survey). These systems, either vertex-centric or block-centric, require recasting sequential algorithms into a new model and revising the logic of the algorithms. Prior work on parallelization has focused on the instruction or operator level [15,16] by breaking dependencies via symbolic and automatic analyses. There has also been work at the data partition level [17], to perform multi-level partition ("parallel abstraction") and adapt locality-optimized access to different parallel abstractions. In contrast, GRAPE aims to parallelize sequential algorithms as a whole, and make parallel computation accessible to end users, while Refs. [15–17] target experienced developers of parallel algorithms. There have also been tools for translating imperative code to MapReduce, e.g., word count [18]. GRAPE advocates a different approach, by parallelizing the runs of sequential graph algorithms to benefit from data-partitioned parallelism, without translation.

Parallel scalability. Parallel scalability has been studied for PRAM (parallel random access machine), a shared-memory parallel model [6]. Under PRAM, all processors can directly access globally shared memory, each step of computation is either a RAM operation or a read/write access to the shared memory, and each read/write takes a unit time. PRAM has three variants, i.e., EREW, CREW, and CRCW [19], based on how read and write access conflicts to the same shared memory cell are resolved. For the strongest CRCW PRAM with n processors and m shared memory cells, one step can be simulated by the weakest EREW PRAM in $O(\log n)$ steps with n processors and $m + n$ shared memory cells [20]. Therefore, in this paper, we consider w.l.o.g. EREW, hereafter simply referred to as PRAM.

A number of parallelly scalable algorithms have been developed for PRAM, e.g., connectivity [21], bi-connectivity [22], co-connectivity [23], and minimum spanning tree [21]. Each of these problems has been shown to admit a PRAM algorithm in polylog$|G|$ time with $O(|G|)$ processors. Then by Brent's scheduling principle [24], for any $n \leqslant |G|$, there exists a PRAM algorithm running in $|G|$polylog$|G|/n$ time with n processors for these problems. In 1997, Spencer [25] showed that the directed breadth-first search problem and topological order problem both admit PRAM algorithms in $O(\frac{|V|\log^2 n}{n^{1/3}})$ time with n processors, for any $(|E|/|V|)^{3/2} \leqslant n \leqslant |V|^3$ (assuming $|V| = O(|E|)$). Note that if $n = (|E|/|V|)^{3/2}$, then the runtime is $O(\frac{|V|^{3/2}\log^2|E|}{\sqrt{|E|}})$. By Brent's scheduling principle [24], for any $n \leqslant (|E|/|V|)^{3/2}$, there exists a PRAM algorithm running in $O(\frac{|E|\log^2|E|}{n})$ time with n processors for each of them.

However, PRAM is a theoretical model and is hard to implement. In contrast to PRAM, parallel systems typically adopt the shared-nothing architecture in practice, for which communication cost is inevitable. A variety of simulation techniques have been developed for transforming the PRAM algorithms to the shared-nothing models and making them practical [26]. It has been shown that each PRAM algorithm with n processors and m memory, where m is polynomial in n, can be simulated deterministically on a module parallel computer (MPC) of n RAM processors with $O(\log m/\log\log m)$ memory redundancy and $O(\log n/\log\log n)$ slowdown [27]. MPC is a fully connected parallel computer comprising a number of RAM processors, each with an associated memory module. All requests that arrive at a memory module in a given cycle are processed sequentially. The MPC model is very different from GRAPE model. We will elaborate on the differences between our results and those of [26] in Section 3.

It is shown in [28, 29] that a PRAM algorithm using t time with n total memory and n processors can be simulated by a MapReduce algorithm, in $O(t)$ rounds using at most $O(n)$ reducers and memory. The work, however, does not consider communication cost. While some PRAM algorithms for graph connectivity have been transformed to a vertex-centric model [30], parallel scalability has not been considered. In contrast, we study simulation strategies for general PTIME graph problems not limited to graph connectivity, under GRAPE model that subsumes vertex-centric models and MapReduce [12].

On shared-nothing systems, parallelly scalable algorithms have been developed for graph pattern matching by subgraph isomorphism [7, 8] and homomorphism [9]. However, to our knowledge, no PTIME parallelly scalable graph algorithms are currently known. Worse yet, no parallelly scalable algorithm exists for graph simulation when the number of processors can be as large as the size of a graph [10]. This work provides the first PTIME algorithms with proven parallel scalability on shared-nothing systems.

2 Parallelizing sequential graph algorithms

In this section, we show how GRAPE parallelizes existing sequential graph algorithms.

We consider a graph $G = (V, E, L)$, 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 a label $L(v)$ (resp. $L(e)$), indicating its content, as found in social networks, knowledge bases and property graphs.

Shared-nothing system. GRAPE adopts a shared-nothing architecture, and supports data partitioned parallelism. Employing a master P_0 and a set of n workers (i.e., processors) $P_1, \ldots, P_n$, GRAPE operates on a graph G that is fragmented into $(F_1, \ldots, F_n)$. For $i \in [1, n]$, each worker P_i hosts a fragment F_i in G. Each fragment $F_i = (V_i, E_i, L_i)$ is a subgraph of G such that $E = \bigcup_{i\in[1,m]} E_i$, $V = \bigcup_{i\in[1,m]} V_i$.

To simplify the presentation, we consider edge-cut partitions [31, 32] in this paper. Nevertheless, GRAPE supports any partitioner strategy picked by users, e.g., vertex-cut partitions [33] and hybrid cut [34]. Under an edge-cut partition, denote by (1) $F_i.I$ the set of nodes $v \in V_i$ such that there exists a cut edge (v', v) from a node v' in F_j $(i \neq j)$; (2) $F_i.O$ the set of nodes v' in some F_j such that there exists a cut edge (v, v') from $v \in V_i$ $(i \neq j)$; and (3) $\mathcal{F}.O = \bigcup_{i\in[1,m]} F_i.O$, and $\mathcal{F}.I = \bigcup_{i\in[1,m]} F_i.I$. A cut edge from F_i to F_j has a copy in each of F_i and F_j. We refer to nodes in $F_i.I$ (or $F_i.O$) as border nodes of F_i. Note that $\mathcal{F}.I = \mathcal{F}.O$.

Programming model. Consider a graph computation problem $\mathcal{Q}$. Using our familiar terms, we refer to an instance Q of $\mathcal{Q}$ as a query of $\mathcal{Q}$. To answer queries $Q \in \mathcal{Q}$ on graphs G, GRAPE takes a PIE program $\rho =$ (PEval, IncEval, Assemble), which consists of three (existing) sequential algorithms as follows:

∘ PEval is a sequential algorithm for $\mathcal{Q}$. Given $Q \in \mathcal{Q}$ and G, it computes answers $Q(G)$ to Q in G.

∘ IncEval is a sequential incremental algorithm for $\mathcal{Q}$. Given Q, G, $Q(G)$ and updates M to graph G, it computes changes ΔO to $Q(G)$ such that $Q(G \oplus M) = Q(G) \oplus \Delta O$. Here $S \oplus \Delta S$ applies ΔS to S.

∘ Assemble collects partial answers computed locally at each worker by PEval and IncEval, and combines them into a complete answer; it is typically simple.

Both PEval and IncEval can be existing sequential algorithms, with the following additions. (1) PEval declares a set $\bar{x}$ of status variables for each fragment F_i, and a candidate set C_i of nodes in F_i. Intuitively, the status variables of nodes in C_i, denoted by $C_i.\bar{x}$, are the candidates to be updated during incremental computation (see below). We refer to $C_i.\bar{x}$ as the update parameters of fragment F_i. (2) PEval also defines an aggregate function f_{aggr} to resolve conflicts, when an update parameter in $C_i.\bar{x}$ is given multiple values by different workers in parallel computation. These definitions are shared with IncEval.

Example 1. Consider graph simulation [35], for which a query is a graph pattern $Q = (V_Q, E_Q, L_Q)$, where (a) V_Q is a set of query nodes, (b) E_Q is a set of query edges, and (c) each node u in V_Q carries a label $L_Q(u)$. A graph $G = (V, E, L)$ matches a pattern Q via simulation if there exists a binary relation $R \subseteq V_Q \times V$ such that (1) for each query node $u \in V_Q$, there exists a node $v \in V$ such that $(u, v) \in R$, and (2) for each pair $(u, v) \in R$, (a) $L_Q(u) = L(v)$, and (b) for each query edge (u, u') in E_q, there exists an edge (v, v') in graph G such that $(u', v') \in R$. There exists a unique maximum such relation (possibly empty) [35], denoted by $Q(G)$, which can be computed in $O((|V_Q| + |E_Q|)(|V| + |E|))$ time [36].

GRAPE parallelizes graph simulation with a PIE program $\rho =$ (PEval, IncEval, Assemble) as follows.

∘ PEval. PEval is the sequential simulation algorithm of [35]. It 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. PEval specifies min as its aggregate function f_{aggr}, taking the order false $\prec$ true (we will elaborate it shortly). As shown in Figure 1(a), PEval (lines 1–15) is identical to the algorithm of [35], except the underlined parts to cope with status variables. For each node $u \in V_Q$, it starts with a set $\mathsf{sim}(u)$ of candidate matches v in F_i (lines 1–5), and iteratively removes from $\mathsf{sim}(u)$ nodes that violate the simulation condition (lines 6–14). It uses $\mathsf{post}(v)$ and $\mathsf{pre}(v)$ for successors and predecessors of node v, respectively (see [35]). It refines matches $\mathsf{sim}(u)$ for all $u \in V_Q$, and ends up with $Q(F_i)$.

∘ IncEval. As shown in Figure 1(b), IncEval is the sequential incremental graph simulation algorithm of [37] in response to edge deletions. If $x_{(u,v)}$ is changed to false by message M_i, it is treated as deletion of "cross edges" from $v \in F_i.O$. 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 4–9).

IncEval is semi-bounded [37]: its cost is decided by the sizes of "updates" $|M_i|$ and changes to the affected area necessarily checked by all incremental algorithms for simulation; it is not decided by $|F_i|$.

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

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

/* Sequential algorithm for graph simulation */
1.  For each $u \in V_Q$ do
2.     if post($u$) = $\emptyset$ then
          sim($u$) := $\{v \in V_i \mid L_Q(u) = L_i(v)\}$;
3.     else sim($u$) := $\{v \in V_i \mid L_Q(u) = L_i(v)$ and
4.                    (post($v$) $\neq \emptyset$ or $x_{(u,v)}$ = true)$\}$;
5.     remove($u$) := pre($V$) \ pre(sim($u$)); /* Initialization */
6.     while there exists $u \in V_Q$ such that remove($u$) $\neq \emptyset$ do
7.        for each $u' \in$ pre($u$) do /* Refinement */
8.          for each $w \in$ remove($u$) do
9.            if $w \in$ sim($u'$) and $w \notin F_i.O$ then
10.             sim($u'$) := sim($u'$) \ $\{w\}$; $x_{(u',w)}$ := false;
11.             for each $w' \in$ pre($w$) do
12.               if post($w'$) $\cap$ sim($u'$) = $\emptyset$ then
13.                 remove($u'$) := remove($u'$) $\cup \{w'\}$;
14.       remove($u$) := $\emptyset$;
15. Partial result $Q(F_i)$ := sim;

Message segment: $C_i.\bar{x} = \{x_{(u,v)} \mid u \in V_Q, v \in F_i.I\}$;
                 $f_{\mathsf{aggr}}$ = min ($x_{(u,v)}$).
```

(a)

```
Input: $Q$, $F_i$, maximum match relation sim for
       $Q$ and $F_i$; message $M_i$ (batch update).
Output: Maximum match relation sim for $Q$ and $F_i \oplus M_i$.

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

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

(b)

Figure 1 GRAPE for graph simulation. (a) PEval and (b) IncEval for simulation.

○ Assemble. Assemble (not shown) simply takes $Q(G) = \bigcup_{i \in [1,n]} Q(F_i)$, the union of sim at each F_i. □

Parallel model. Under the bulk synchronous parallel model (BSP) [38], given a query $Q \in \mathcal{Q}$, GRAPE posts the same Q to all workers, and executes a PIE program in supersteps as a simultaneous fixpoint computation defined as follows, by treating IncEval as its intermediate consequence operator:

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

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

where $i \in [1, n]$, r is the superstep index, R_i^r is the partial result in step r at worker P_i, $F_i^0 = F_i$, $F_i^r[\bar{x}]$ is fragment F_i at the end of superstep r (including its update parameters $C_i.\bar{x}$), and M_i denotes messages sent to P_i, i.e., changes to $C_i.\bar{x}$. More specifically, the parallel computation is conducted as follows.

(1) *Partial evaluation* (PEval). In the first superstep, PEval computes the partial answers $R_i^0 = Q(F_i)$ at each worker P_i on its local fragment F_i, in parallel for all $i \in [1, n]$. After $Q(F_i)$ is computed, each worker generates a message consisting of update parameters $C_i.\bar{x}$ and sends it to master P_0.

Continuing with Example 1, at the end of the process, PEval sends $C_i.\bar{x} = \{x_{(u,v)} \mid u \in V_Q, v \in F_i.I\}$ to master P_0. Upon receiving messages from all workers, P_0 changes $x_{(u,v)}$ to false if it is false in one of the messages. This is handled by aggregate function f_{aggr} (min) specified in PEval. GRAPE identifies those variables that become false, groups them into messages M_j, and sends M_j to processor P_j.

The connection between PEval and partial evaluation is as follows. 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 [39]. 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 PEval at each processor P_i, the local fragment F_i is its known input s, while the data residing at other processors accounts for the yet unavailable input d. As the first step of parallel computation, PEval computes $Q(F_i)$ as partial evaluation.

(2) *Incremental computation* (IncEval). In the following supersteps, the partial answers $Q(F_i)$'s are iteratively updated by IncEval. More specifically, (a) master P_0 applies f_{aggr} to the messages from the last superstep to resolve conflicts. These aggregated values are routed to the relevant workers. (b) Upon receiving the message M_i, worker P_i incrementally computes $R_i^{r+1} = Q(F_i \oplus M_i)$ with IncEval in parallel for $i \in [1, n]$, by treating M_i as updates. At the end of each superstep, worker P_i sends a message to P_0 that consists of changes to the update parameters $C_i.\bar{x}$ of F_i just as in PEval.

Continuing with Example 1, at the end of its process, IncEval sends to master P_0 updated values of status variables in $C_i.\bar{x}$, i.e., $x_{(u,v)}$ that is changed from true to false in this superstep for $v \in F_i.I$.

Intuitively, graph computations are often iterative. If $Q(G)$ cannot be obtained in one round by PEval, GRAPE exchanges selected partial results as messages between processors, and IncEval incrementally computes $Q(F_i \oplus M_i) = Q(F_i) \oplus \Delta O_i$, by treating message M_i to P_i as updates to the update parameters of F_i, reusing the old output $Q(F_i)$. Incremental computation is often more efficient than recomputing $Q(F_i \oplus M_i)$ from scratch, since in practice, M_i is typically small, and so is ΔO_i. Better still, it may be bounded: its cost depends only on the sizes of the message M_i to and changes ΔO_i to the output $Q(F_i)$, not on the size $|F_i|$ of the entire fragment F_i [40, 41], minimizing unnecessary recomputation.

(3) *Termination.* The process proceeds until it reaches a fixpoint, i.e., no more changes to update parameters. At this point Assemble is invoked to combine all partial answers into $Q(G)$.

Convergence. The correctness of the fixpoint computation is characterized as follows. For a graph computation problem $\mathcal{Q}$, we say that GRAPE correctly parallelizes a PIE program ρ if for all queries $Q \in \mathcal{Q}$ and graphs G, it reaches a fixpoint and returns $Q(G)$. Moreover, (a) its sequential algorithm PEval is correct for $\mathcal{Q}$ if for all queries $Q \in \mathcal{Q}$ and graphs G, it computes $Q(G)$; (b) its sequential incremental algorithm IncEval is correct for $\mathcal{Q}$ if it returns $Q(G \oplus M)$ by computing the changes ΔO to old output $Q(G)$, for changes (messages) M to update parameters; and (c) Assemble is correct for $\mathcal{Q}$ if it computes $Q(G)$ by assembling the partial answers from all workers, when GRAPE reaches a fixpoint.

It is shown that under BSP, GRAPE [12] correctly parallelizes a PIE program ρ for a graph computation problem $\mathcal{Q}$ if (a) PEval, IncEval, and Assemble of ρ are correct for $\mathcal{Q}$, and (b) PEval and IncEval satisfy a monotonic condition. The condition is as follows: for all status variables $x \in C_i.\bar{x}$, $i \in [1, n]$, (a) the values of x are from a finite set computed from the active domain of G, and (b) there exists a partial order p_x on the values of x such that IncEval updates x in the order of p_x. That is, x draws values from a finite domain (condition (a) above), and x is updated "monotonically" following p_x (condition (b)).

For example, the correctness of the PIE program of Figure 1 for graph simulation is warranted by the convergence condition. Indeed, the sequential algorithms [35, 37] (PEval and IncEval) are correct and monotonic: $x_{(u,v)}$ is initially true for each border node v, and is changed at most once to false.

Properties. GRAPE has the following unique features.

(1) GRAPE aims to help users develop parallel programs, especially those who are more familiar with conventional sequential programming. To program with GRAPE, one only needs to provide a PIE program, which consists of (existing) sequential algorithms with minor changes. GRAPE parallelizes the sequential algorithms as a whole. As a result, users do not have to "think parallel" when programming with GRAPE.

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

(3) As shown in [12], GRAPE substantially outperforms most state-of-the-art parallel graph systems in efficiency, for the following reasons. (a) GRAPE inherits existing optimization techniques developed for sequential graph algorithms, since it executes sequential algorithms on graph fragments, which are graphs themselves. (b) GRAPE reduces the costs of iterative graph computations by using IncEval, to minimize unnecessary recomputations, irrespective of whether IncEval is bounded or not. The performance improvement has been validated by Alibaba Group, where GRAPE has been deployed.

(4) As shown in [13], PIE programs also work under asynchronous models and guarantee to converge under a generic condition as long as the sequential algorithms plugged in are correct. Better yet, under an adaptive asynchronous model, GRAPE is 4.8 times faster than under BSP on average.

3 Parallel scalability

In this section, we first formalize the notion of parallel scalability following [6]. We then provide a simple condition for deciding whether a graph computation problem is parallelly scalable.

Parallel scalability. We start with parallel cost. Consider a PIE program ρ for a graph computation problem $\mathcal{Q}$. Given a query $Q \in \mathcal{Q}$, it computes $Q(G)$ in a graph G that is partitioned and distributed across a cluster of n workers, such that each worker P_i hosts a fragment F_i of G, and $\max_{i\in[1,n]}|F_i| = \tilde{O}(|G|/n)$, where $|G|$ denotes the size of G and the notation $\tilde{O}(\cdot)$ hides logarithmic factors. The workers are pairwise connected by bi-directional communication channels. Under BSP, assume that ρ takes k rounds to reach a fixpoint. Denote by $T_\rho(G,Q,n)$ the total time taken by ρ. Then

$$T_\rho(G,Q,n) = \sum_{r\in[1,k]}\left(\max\nolimits_{i\in[1,n]} t^{(i,r)}_{\mathsf{Cmp}}\right) + \sum_{r\in[1,k]}\left(\max\nolimits_{i\in[1,n]} t^{(i,r)}_{\mathsf{Cmm}}\right),$$

where $t^{(i,r)}_{\mathsf{Cmp}}$ (resp. $t^{(i,r)}_{\mathsf{Cmm}}$) denotes the computational (resp. communication) cost incurred by worker i in round r. Intuitively, the parallel cost of ρ in each round is determined by the maximum computational and communication costs inflicted by individual workers. The runtime of ρ is the sum of the costs in k rounds. While the cost can be further reduced by overlapping computation and communication, to simplify the presentation, we simply take the sum of $t^{(i,r)}_{\mathsf{Cmp}}$ and $t^{(i,r)}_{\mathsf{Cmm}}$ for each worker under BSP.

Parallel scalability. We revise the criterion of [6] for the effectiveness of parallel algorithms. Consider a sequential algorithm $\mathcal{A}$ for $\mathcal{Q}$, referred to as a yardstick for $\mathcal{Q}$. Denote by $T^{\max}_{\mathcal{A}}(m,Q)$ the maximum time taken by $\mathcal{A}$ for computing $Q(G)$ on a single machine over all possible graphs G, where $|G| = m$, and by $T^{\max}_{\rho}(m,Q,n)$ the maximum $T_\rho(G,Q,n)$ using n processors.

We say that a PIE program ρ is parallelly scalable for $\mathcal{Q}$ relative to $\mathcal{A}$ if for all queries $Q \in \mathcal{Q}$,

$$T^{\max}_{\rho}(m,Q,n) = \tilde{O}\left(\frac{T^{\max}_{\mathcal{A}}(m,Q)}{n}\right),$$

where $n \ll m$, i.e., the number of workers is much smaller than the sizes of graphs, and $|Q| \ll m$, where $|Q|$ is the size of Q, as commonly found in practice. We assume w.l.o.g. that $T^{\max}_{\mathcal{A}}(m,Q)$ is $\Omega(m)$ for all $Q \in \mathcal{Q}$ and sequential algorithms $\mathcal{A}$, since reading the input G and Q alone takes $|G| + |Q| = m + |Q|$ time.

Intuitively, parallel scalability measures parallel speedup over sequential algorithms. It is a relative measure w.r.t. a yardstick algorithm $\mathcal{A}$. A parallelly scalable PIE program ρ reduces the sequential runtime of $\mathcal{A}$ when n increases. Hence it is able to scale with large G by adding processors as needed.

We say that ρ is parallelly scalable for $\mathcal{Q}$ if ρ is parallelly scalable relative to all sequential algorithms. A computation problem $\mathcal{Q}$ is parallelly scalable if there exists a parallelly scalable PIE program for $\mathcal{Q}$.

A sufficient condition. We next identify a condition under which $\mathcal{Q}$ is parallelly scalable.

Theorem 1. Let $\mathcal{Q}$ be a graph computation problem such that $T^{\max}_{\mathcal{A}}(m,Q)$ is $\Omega(m)$ for all queries $Q \in \mathcal{Q}$ and sequential algorithms $\mathcal{A}$ for $\mathcal{Q}$. Then $\mathcal{Q}$ is parallelly scalable if there exists a PRAM algorithm such that for each $Q \in \mathcal{Q}$, it computes $Q(G)$ with $O(|G|)$ processors in polylog($|G|$) time, as long as $n \leqslant \sqrt{|G|}$, where n is the number of workers used by PIE programs. □

Intuitively, Theorem 1 allows us to identify parallelly scalable problems by capitalizing on existing PRAM algorithms. A number of PRAM algorithms are already in place. Many graph computation problems are known to admit a PRAM algorithm in polylog($|G|$) time with $O(|G|)$ processors, e.g., graph connectivity [21], minimum spanning trees [21], bi-connectivity [22], co-connectivity [23], and ear decomposition [42,43]. Theorem 1 tells us that parallelly scalable PIE programs exist for these problems on shared-nothing GRAPE. Here we consider $|Q| \ll |G|$ and hence do not list $|Q|$ as a separate parameter.

The condition in Theorem 1 differs from the simulation theorem of [26] as follows. The simulation of [26] incurs heavy memory redundancy, which is more staggering than slowdown in practice. Moreover, the simulation technique in [26] is not constructive, i.e., it only shows the existence of such an MPC algorithm but does not tell us how to develop one. In contrast, Theorem 1 does not inflict substantial memory redundancy, and its proof below shows how to construct a parallelly scalable GRAPE program.

Proof. Let $\mathcal{B}$ be a PRAM algorithm having the properties described in Theorem 1. We provide a PIE program ρ to simulate $\mathcal{B}$ such that $T_\rho(G,Q,n) = \tilde{O}(|G|/n)$ for each $Q \in \mathcal{Q}$ when $n \leqslant \sqrt{|G|}$. To avoid confusion, we refer to the computing nodes of $\mathcal{B}$ and ρ as processors and workers, respectively.

As a roadmap, below we first outline how to simulate the processors and memory cells of $\mathcal{B}$ with their counterparts of ρ. We then provide the three functions of ρ, i.e., PEval, IncEval, and Assemble. Finally, we show that ρ takes $\tilde{O}(|G|/n)$ time in total and is parallelly scalable; hence so is problem $\mathcal{Q}$.

Simulation strategy. We use the following notations. (1) Denote by k the number of processors used by $\mathcal{B}$. (2) The memory of $\mathcal{B}$ can be divided into two parts, namely, input memory cells and extra memory cells. Denote by l and q the sizes of the two parts, respectively. One can verify that k, l, and q are all $\tilde{O}(|G|)$ as follows: (a) $l = |Q| + |G| = O(|G|)$ for each given query Q by $|Q| \ll |G|$; (b) $k = O(|G|)$ since $\mathcal{B}$ uses $O(|G|)$ processors; and (c) $\mathcal{B}$ accesses $q = O(|G|\text{polylog}(|G|)) = \tilde{O}(|G|)$ extra memory cells since $\mathcal{B}$ computes $Q(G)$ with $O(|G|)$ processors in polylog$(|G|)$ time. To simplify the discussion, we assume w.l.o.g. that k, l, and q are all multiples of n; it is easy to extend our proof to cover generic k, l, and q. (3) We denote the processors and the memory cells of $\mathcal{B}$ with their IDs. For example, extra cell j of $\mathcal{B}$ stands for the j-th extra cell. (4) We use $f(t)$, $g(t)$, and $h(t)$ to denote functions $\lceil tn/k \rceil$, $\lceil tn/l \rceil$, and $\lceil tn/q \rceil$, respectively, where $\lceil \cdot \rceil$ is the ceiling function such that $\lceil a \rceil$ is the least integer no less than a. (5) We use $\lambda(t)$ to denote $t - q \cdot (\lceil tn/q \rceil - 1)/n$. That is, $\lambda(t)$ is q/n if t is divisible by q/n, otherwise $\lambda(t)$ is the remainder.

The PIE program ρ simulates the processors, input cells, and extra cells of $\mathcal{B}$ as follows. (1) Each worker of ρ simulates k/n processors of $\mathcal{B}$. Processor t of $\mathcal{B}$ is simulated by worker $P_{f(t)}$ of ρ. (2) Let $d_1, \ldots, d_l$ be the inputs of $\mathcal{B}$. Then the inputs of ρ are $(1, d_1), \ldots, (l, d_l)$, each identified by an ID. The inputs are distributed across the workers such that graph G is partitioned and distributed evenly. Thus, the size of inputs on each P_i is $l_i = \tilde{O}(|G|/n)$. PEval of ρ redistributes the inputs such that input (j, d_j) resides at worker $P_{g(j)}$. (3) Each worker of ρ accesses q/n extra cells in addition to its local input. We use (i, j) to denote the j-th extra cell on P_i. The extra cell s of $\mathcal{B}$ is simulated by the extra cell $(h(s), \lambda(s))$ of ρ.

In our simulation, (1) if processor t conducts local computation, then worker $P_{f(t)}$ carries out the same computation, (2) if t accesses input d_j, then $P_{f(t)}$ accesses input (j, d_j) at $P_{g(j)}$ via communication, and (3) if t accesses extra cell s, then $P_{f(t)}$ accesses extra cell $(h(s), \lambda(s))$ via communication.

The PIE program. We next implement the simulation under GRAPE. Observe that each step of a PRAM program has three phases: (1) read from a shared memory cell; (2) conduct a local computation; and (3) write to a shared memory cell [19]. The PIE program ρ simulates one step of $\mathcal{B}$ using two supersteps: one for the local computation and the other for auxiliary operations of memory access.

Complications arise from two mismatches. (a) Each processor of $\mathcal{B}$ executes different instructions in different steps, and different processors execute different instructions in the same step. In contrast, the PIE program ρ has only three functions, i.e., PEval, IncEval, and Assemble. (b) A PRAM algorithm $\mathcal{B}$ can directly access all memory cells, while each worker of ρ can directly access only its local memory cells.

We deal with the mismatches as follows. (a) We program PEval and IncEval to treat diverse instructions of $\mathcal{B}$ as subroutines. PEval employs an index from input to decide which subroutines to run, and IncEval uses an ID tuple encoded in update parameters to select right subroutines. (b) We simulate the reading and writing of $\mathcal{B}$ by storing the memory access requests as memory access tuples in the update parameters of ρ. GRAPE realizes the memory access by encoding these tuples as messages from workers to master P_0. The master P_0 collects messages from workers and routes them to the corresponding workers.

To pass these messages, we employ a graph G_W as additional input along the same lines as [12,13]. Here G_W is an undirected graph with n^2 nodes and $n(n-1)/2$ edges. For each $i \in [1, n]$, nodes $w_{i,1}, \ldots, w_{i,n}$ in G_W are assigned to worker P_i. For each $i \neq j \in [1, n]$, there is an edge between $w_{i,j}$ and $w_{j,i}$. The ID tuple is stored in the status variable of $w_{i,i}$ as part of its update parameter. Meanwhile, the memory access tuples are stored in the status variable of $w_{i,j}$, which encode requests for P_j to read from P_i and for P_i to write to P_j. We also use an index array D_I as additional input, based on which PEval picks the appropriate subroutines to execute. Here D_I is the array $1, 2, \ldots, n$. For each $i \in [1, n]$, element i in D_I is assigned to worker P_i. Observe that the sizes of G_W and D_I do not exceed $|G|$ since $n \leqslant \sqrt{|G|}$.

We next present PIE program ρ: its update parameters, aggregate f_{aggr}, PEval, IncEval, and Assemble.

(1) *Update parameters.* The update parameters of ρ encode the ID tuples and memory access tuples as mentioned above. An ID tuple is of the form $\langle$ID-step, phase, ID-worker$\rangle$, where ID-step is the step ID of $\mathcal{B}$, phase is either "computing" or "auxiliary", and ID-worker is the worker ID. A memory access

tuple is of the form ⟨operation-section, address, data⟩, where address is the address of the accessed cell in $\mathcal{B}$, data denotes the data to be read or written, and operation-section is "transfer-input", "read-input", "read-extra" or "write-extra". While "read-input", "read-extra", and "write-extra" are self-explanatory, "transfer-input" is to redistribute the input across the n workers at the end of PEval.

(2) PEval. It declares status variables of update parameters, and redistributes graph G and query Q by referencing D_I. By citing the element i at P_i, the ID tuple in $w_{i,i}$ is initialized as $\langle 0$, "auxiliary", $i\rangle$. Moreover, if input (s, d_s) is at P_i, then P_i stores memory access tuple ⟨"transfer-input", s, d_s⟩ in $w_{i,g(s)}$.

(3) IncEval. It reads the ID tuple from $w_{i,i}$ and simulates the PRAM instructions of $\mathcal{B}$ based on the tuple.

(a) If the ID tuple is $\langle 0$, "auxiliary", $i\rangle$, IncEval reads the fragmental input from the update parameters and prepares for the reading in the first step of $\mathcal{B}$. (i) Referencing the ID tuple, P_i reads all ⟨"transfer-input", s, d_s⟩ from $w_{1,i}, \ldots, w_{n,i}$, where $g(s) = i$, and stores (s, d_s) in the local memory. These memory access tuples are stored in $w_{1,i}, \ldots, w_{n,i}$ by PEval. (ii) Worker P_i prepares for the reading of inputs d_{s_1} and extra cells s_2 in the first step of $\mathcal{B}$, where $g(s_1) = i$ and $h(s_2) = i$. More specifically, if processor t reads these inputs or extra cells in the first step of $\mathcal{B}$, then P_i stores the memory access tuples in $w_{i,f(t)}$.

(b) If the ID tuple is $\langle r$, "computing", $i\rangle$, where $r \geqslant 1$, IncEval implements the operations in step r of $\mathcal{B}$. Referencing the ID tuple, worker P_i simulates all the processors t one by one as follows, where $f(t) = i$. Assume w.l.o.g. that in step r of $\mathcal{B}$, processor t reads data D from extra cell s_1, does some local computation and writes the result D' to extra cell s_2. (i) Then the data D is stored in a memory access tuple at $w_{h(s_1),i}$ before this superstep. This is carried out when the ID tuple is $\langle r-1$, "auxiliary", $h(s_1)\rangle$. (ii) To simulate the operations of processor t, P_i simply reads D from $w_{h(s_1),i}$, performs the same local computation of t, and stores the result D' in a memory writing tuple ⟨"write-extra", s_2, D'⟩ at $w_{i,h(s_2)}$. (iii) The data D' stored at $w_{i,h(s_2)}$ is written to cell $(h(s_2), \lambda(s_2))$ when the ID tuple is $\langle r$, "auxiliary", $h(s_2)\rangle$. Similarly, if processor t reads input d_{s_1}, then (s_1, d_{s_1}) is stored in a memory access tuple at $w_{g(s_1),i}$ before this superstep. This is implemented when the ID tuple is $\langle r-1$, "auxiliary", $g(s_1)\rangle$.

(c) If the ID tuple is $\langle r$,"auxiliary", $i\rangle$, where $r \geqslant 1$, IncEval completes the writing in step r and prepares for the reading in step $r+1$ of $\mathcal{B}$. Referencing the ID tuple, worker P_i simulates the access to all the inputs d_{s_1} and extra cells s_2, where $g(s_1) = i$ and $h(s_2) = i$. Assume w.l.o.g. that processor t_1 writes data D into extra cell s in step r, and processor t_2 reads D from s in step $r+1$, where $h(s) = i$. (i) Then D is stored in a memory access tuple at $w_{f(t_1),i}$ before this superstep, when the ID tuple is $\langle r$, "computing", $f(t_1)\rangle$. (ii) To simulate the writing and reading, P_i reads D from $w_{f(t_1),i}$, writes it to local cell $(h(s), \lambda(s))$, and stores D in a memory reading tuple ⟨"reading-extra", s, D⟩ at $w_{i,f(t_2)}$. (iii) Worker $P_{f(t_2)}$ reads D from $w_{i,f(t_2)}$ when the ID tuple is $\langle r+1$, "computing", $f(t_2)\rangle$. Similarly, if processor t_2 reads input d_s in step $r+1$, where $g(s) = i$, then P_i stores the memory reading request at $w_{i,f(t_2)}$.

At the end of each superstep, P_i updates the ID tuple at $w_{i,i}$: (i) if the tuple is $\langle r$, "computing", $i\rangle$, it is updated to $\langle r$, "auxiliary", $i\rangle$; and (ii) if it is $\langle r$, "auxiliary", $i\rangle$, it becomes $\langle r+1$, "computing", $i\rangle$.

(4) *Message passing.* The master P_0 groups and routes messages as follows. (a) It first takes a union of the update parameters of all nodes $w_{i,j}$ $(i, j \in [1, n])$. (b) It then sends (the changed values of) the update parameters of all $w_{j,i}$ to worker P_i $(i, j \in [1, n])$. The aggregate function is defined as $f_{\mathsf{aggr}}(x) = x$, since no update parameter is assigned different values from different workers.

(5) Assemble. Assemble simply collects partial results from all workers and returns them all.

One can easily verify that the PIE program ρ correctly simulates PRAM algorithm $\mathcal{B}$. That is, for each graph G and query $Q \in \mathcal{Q}$, ρ takes G, Q, G_W, and D_I as input and returns the same result $Q(G)$ as $\mathcal{B}$.

Time complexity. We next show that the total time of ρ is $T_\rho(G, Q, n) = \tilde{O}(|G|/n)$ for all queries $Q \in \mathcal{Q}$. Hence $T_\rho^{\max}(m, Q, n) = \tilde{O}(m/n)$. From this and the fact that $T_{\mathcal{A}}^{\max}(m, Q)$ is $\Omega(m)$ for all $Q \in \mathcal{Q}$ and sequential algorithms $\mathcal{A}$, it follows that $\mathcal{Q}$ is parallelly scalable.

Observe the following. (1) The additional inputs G_W and D_I can be constructed in $O(n) = O(|G|/n)$ time with n workers as follows: (a) D_I is built at one worker in $O(n)$-time since the size of D_I is n. (b) To build G_W, each worker P_i only needs to generate nodes $w_{i,j}$, $w_{j,i}$, $w_{i,i}$ and edges between $w_{i,j}$ and $w_{j,i}$ for all $j \neq i \in [1, n]$. This takes $O(n)$ time. (2) Since $\mathcal{B}$ has polylog$(|G|)$ steps, it is easy to see that ρ also has polylog$(|G|)$ supersteps. Thus, it suffices to show that the maximum computational and communication costs incurred by each worker are both $\tilde{O}(|G|/n)$ in each superstep of ρ. Here we only prove the bound for the communication cost; the proof for the computational cost is similar. In each superstep, the memory access tuples are encoded as messages between the workers and master P_0. We show that the total lengths of messages sent and received by each worker are both $\tilde{O}(|G|/n)$. There are two cases to consider.

(a) In PEval, the memory access tuples about inputs d_s are stored at $w_{i,1}, \ldots, w_{i,n}$, where (s, d_s) is at worker P_i. As graph G is evenly partitioned across the workers, there are $\tilde{O}(|G|/n)$ inputs at worker P_i. Thus, there are also $\tilde{O}(|G|/n)$ memory access tuples stored at $w_{i,1}, \ldots, w_{i,n}$. As each memory access tuple has a constant size, the total length of the messages sent from P_i to P_0 during PEval is $\tilde{O}(|G|/n)$. Moreover, only the memory access tuples about inputs d_s are stored at one of $w_{1,i}, \ldots, w_{n,i}$, where $g(s) = i$. By $g(s) = \lfloor sn/l \rfloor$, there are $l/n = \tilde{O}(|G|/n)$ memory access tuples stored at one of $w_{1,i}, \ldots, w_{n,i}$. Thus the total length of the messages received by P_i from P_0 during PEval is also $\tilde{O}(|G|/n)$.

(b) In IncEval, if the ID tuple in $w_{i,i}$ is $\langle r$, "computing", $i\rangle$, the memory access tuples for processor t to write some extra cell in step r of $\mathcal{B}$ are stored at one of $w_{i,1}, \ldots, w_{i,n}$, where $f(t) = i$. Note that each processor writes at most one cell in each step of $\mathcal{B}$, and $f(t) = \lfloor tn/k \rfloor$. Hence, at most k/n memory access tuples are stored at $w_{i,1}, \ldots, w_{i,n}$. Thus, at the end of this superstep, the total length of the messages sent from P_i to P_0 is $O(k/n) = \tilde{O}(|G|/n)$. One can verify that the total length of the messages received by P_i from P_0 is also $\tilde{O}(|G|/n)$, since $\mathcal{B}$ is EREW and the total size of the inputs and extra cells at P_i is $l_i + q/n = \tilde{O}(|G|/n)$. Along the same lines, one can verify that when the ID tuple at $w_{i,i}$ is $\langle r$,"auxiliary", $i\rangle$, the total length of messages sent from P_i to P_0 and from P_0 to P_i are both $\tilde{O}(|G|/n)$. □

4 Polynomial-time parallelly scalable algorithms

In this section, we develop parallelly scalable algorithms for two PTIME graph-computation problems: graph connectivity (GC) and minimum spanning trees (MST). Both problems admit PRAM algorithms that use $\tilde{O}(|G|)$ processors and run in polylog$|G|$ time [19]. Hence, by Theorem 1, they are parallelly scalable. We further confirm Theorem 1 by providing parallelly scalable PIE programs on GRAPE.

In this section, we adopt the BSP model [38], edge-cut partitions [31, 32], and at most $\sqrt{|G|}$ workers.

4.1 Graph connectivity

We start with graph connectivity. Given an undirected graph G, a connected component (CC) of G is an induced subgraph such that (a) it is connected, i.e., for any pair of its nodes, there exists a path between them; and (b) it is maximum, i.e., adding any node makes the induced subgraph no longer connected.

Given an undirected graph $G = (V, E)$, the graph connectivity problem is to compute all connected components of G. We refer to the problem as GC. This problem is known to be in $\tilde{O}(|G|)$ time [44]. Note that the class $\mathcal{Q}$ of queries for GC consists of a single query of constant size.

Below we give an overview of our algorithm (Subsection 4.1.1), followed by the PIE program (Subsection 4.1.2).

4.1.1 *Algorithm sketch for graph connectivity*

As suggested by Theorem 1, we develop our PIE program by converting the PRAM algorithm of [19].

The algorithm is iterative. Each iteration starts with disjoint blocks $C_1, C_2, \ldots, C_{n_i}$ of nodes, and merges "neighboring" blocks into larger ones, such that all nodes of each block belong to the same CC of G. Initially, $n_1 = |V|$, i.e., each block simply consists of a single node. Its key ideas are as follows.

Maintaining blocks as stars. Each block is maintained as a star with a root node and a number of leaves. Each node u is associated with a pointer $D(u)$. At a root node r, the pointer is a self-loop, i.e., $D(r) = r$, and r is treated as the id of the block. At each leaf node u, $D(u)$ points to the root node r of the block to which u belongs. Initially, for each node $u \in V$, $D(u) = u$. When the algorithm terminates, it guarantees that for any nodes $u, v \in V$, $D(u) = D(v)$ if and only if u and v belong to the same CC. Hence, to decide whether nodes u and v are in the same block, it suffices to check whether $D(u) = D(v)$.

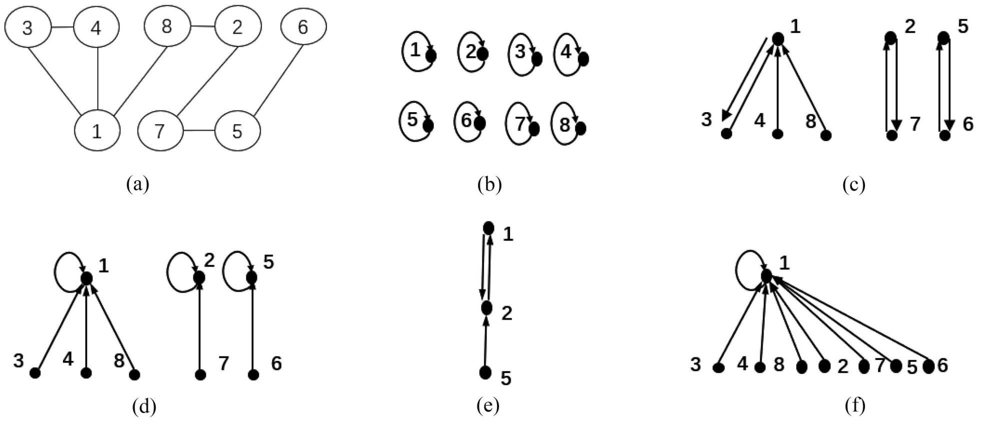

Figure 2 An example illustrating the PIE program for GC.

Merge proposals. Each iteration merges some of the blocks that are "neighbors". More specifically, for each block C_j (with root r_j), we find its neighboring blocks C_i, i.e., there is an edge $(v, u) \in E$ such that $D(v) = r_j$, $D(u) = r_i$ and $i \neq j$. Let C_k be the neighboring block of C_i with the minimum id. We say that block C_i proposes to merge with C_k, and set $p(r_j) = r_k$. If C_j has no neighboring blocks, we set $p(r_j) = r_j$.

We set $D(r_j) = p(r_j)$ for each root r_j, and treat $(u, D(u))$ as a directed edge from node u to $D(u)$. These form a pseudo-forest T with nodes in V. Each connected component C'_j in T is a pseudo-tree, which is a tree plus one extra edge from the root r_k of C'_j (owning to $p(r_k)$). Note that each pseudo-tree contains exactly one cycle. One can easily verify that each cycle in T is either a self-loop or contains exactly two directed edges, because the pointer p points to the root of the minimum neighboring block.

Transforming pseudo-trees into stars. All nodes in each pseudo-tree are in the same CC of G. However, a pseudo-tree may not have a star shape. To transform the pseudo-forest into stars, we perform $\log |V|$ rounds of pointer jumping, by setting $D(u) = D(D(u))$ for each node u in each round. Note that $D(u)$ points to one of at most two nodes in the cycle of the pseudo-tree containing u. As a final step, we set $D(u) = \min(D(u), p(D(u)))$, so that all nodes in the same pseudo-tree point to the same root. This yields stars.

Example 2. Consider graph G depicted in Figure 2(a). Initially, $D(i) = i$ and each node is a root, as shown in Figure 2(b). In the first iteration, since node 1 is the minimum neighbor for each of nodes 3, 4, and 8, the three nodes propose to merge with node 1, i.e., $p(3) = p(4) = p(8) = 1$, and $p(1) = 3$. Similarly, the pointers p are determined for other nodes. These form a pseudo-forest, as shown in Figure 2(c). After transforming each pseudo-tree into a star, we obtain three blocks, as shown in Figure 2(d).

In the second iteration, we set $p(2) = 1$ because node 1 is the root of the minimum block neighboring $\{2, 7\}$, i.e., block $\{2, 7\}$ proposes to merge with $\{1, 3, 4, 8\}$. Similarly, $p(1) = 2$ and $p(5) = 2$, as shown in Figure 2(e). By transforming pseudo-trees into stars, we merge the three blocks into the one shown in Figure 2(f). Now the algorithm terminates and outputs the single block as the CC of G. □

4.1.2 *A* PIE *program for graph connectivity*

We are now ready to present the PIE program for GC, as shown in Figure 3. It operates on a graph G evenly partitioned into $(F_1, \ldots, F_n)$, where $F_i = (V_i, E_i)$. In a nutshell, it adopts a sequential GC algorithm as PEval. PEval computes the local CCs of each fragment F_i, and maintains CCs as stars. Following the algorithm sketch given above, IncEval is recursively applied and merges blocks into bigger ones, until no more blocks can be merged. It consists of many branches. Each worker employs a stack S, which controls the switches among these branches. The details of the PIE program are as follows.

(1) PEval. As shown in Figure 3(a), PEval constructs graph G_W as suggested in the proof of Theorem 1. At each fragment F_i, it declares a status variable $w_{ij}.x$ for each node w_{ij} of G_W, such that messages from P_i to P_j are stored in $w_{ij}.x$ (line 1). It defines variables $D(u)$, $p(u)$, and $q(u)$ for each node u in V_i, all initialized as u (line 1). Here we use $q(u)$ to record the minimum $D(v)$ such that $D(v) \neq D(u)$ and

```
Input: A fragment $F_i$ of graph $G$.
Output: $D(u)$ for all $u \in V_i$.
Message preamble: Construct $G_W$. For each node $w_{ij}$ in $G_W$,
                  declare variable $w_{ij}.x$, initialized as $\perp$;
1. Create $D(u) := u; p(u) := u; q(u) := u$ for each $u \in V_i$;
2. $\mathbb{C} := \mathsf{DFS}(F_i)$;
3. for each $C \in \mathbb{C}$ do
4.   $u_c := \mathsf{argmin}\{u.\mathsf{id} \mid u \in C\}$;
5.   for each $u \in C$ do
6.     $D(u) := u_c$;
7. Create a stack $S$ and push the tuple ('MP', 0) into $S$;
8. for each $j$ do
9.   $w_{i,j}x := \{(u, D(u)) \mid u \in F_i.I \cap F_j.O, i \neq j\}$;
Message segment: $M_{(i,j)} = w_{ij}.x$ if $w_{ij}.x \neq \perp$
                 $f_{\mathsf{aggr}}(x) = x$.
```

(a)

```
Input: A fragment $F_i$, partial results $Q(F_i)$, and
       message $M_i$.
Output: $Q(F_i \oplus M_i)$.
1. Set each $w_{ij}.x$ to $\perp$;
2. $s$:= $S$.pop();
3. switch($s[0]$)
4.   Case 'MP': execute Branch MP($s[1]$);
5.   Case 'SORT': execute Branch SORT($s[1]$);
6.   Case 'TPTS': execute Branch TPTS($s[1]$);
7.   Case 'CA': execute Branch CA($s[1]$).
```

(b)

Figure 3 PIE program for graph connectivity. (a) PEval and (b) IncEval for GC.

$(u, v) \in E$. It then employs a standard sequential traversal (e.g., DFS, depth-first search) to compute the local CCs of F_i (line 2). For each local CC C, PEval (a) finds the node u_c with minimum id and treats it as the root of C, (b) sets $D(u)$ to u_c for each node $u \in C$, (c) initializes a stack S, and (d) exchanges the status variables of border nodes with neighboring fragments (lines 3–9). We use aggregate function $f_{\mathsf{aggr}}(x) = x$ because there is no conflict when updating status variables $w_{ij}.x$.

(2) IncEval. As shown in Figure 3(b), IncEval consists of multiple branches, such that different invocations of IncEval conduct different computations. The branches are classified into four types, i.e., MP, SORT, TPTS, and CA, to perform merge proposal (MP), the sorting algorithm (SORT) [29, 45], transforming pseudo-trees into stars (TPTS), and concurrent access (CA), respectively. Each invocation of IncEval executes one of these branches.

More specifically, each worker maintains a stack S to determine which branch to execute. Initially, S contains a single tuple, i.e., ("MP", 0) pushed into S by PEval. In the first invocation of IncEval, ("MP", 0) is popped, and Branch MP(0) is executed, in which IncEval pushes ("MP", 1) into S, followed by ("SORT", 0). Hence, in the second invocation of IncEval, ("SORT", 0) is popped out of S and Branch SORT(0) is executed. After these, the invocations of IncEval can be grouped into phases, where each phase consists of the invocations between two Branch MP(0) executions. In each phase, the branches are executed in the following order: first MP(0)$\rightarrow$SORT$\rightarrow$MP(1)$\rightarrow$MP(2)$\rightarrow$TPTS(0); then repeat CA(0)$\rightarrow$SORT$\rightarrow$CA(1)$\rightarrow \cdots \rightarrow$CA(5) for $\log |V| + 1$ times; finally TPTS(1). Moreover, each phase is divided into two stages: MP(0)$\rightarrow \cdots \rightarrow$MP(2) and TPTS(0)$\rightarrow \cdots \rightarrow$TPTS(1).

Denote by s and $s[i]$ a tuple in S and the i-th element of s, respectively. For example, when s is the tuple popped in the first invocation of IncEval, $s[0] =$"MP" and $s[1] = 0$.

To realize the ideas of Subsection 4.1.1, two issues need to be resolved: (a) in merge proposal, how to find the node with the minimum $q(u)$ among all the nodes in a block, when the nodes are distributed to different workers? (b) For pointer jumping, how to obtain $D(D(u))$ ($p(D(u))$) for each node u, which may extend across workers? We develop two procedures for these two issues (see below), referred to as concurrent minimization and concurrent access, respectively, following the concurrent write and read operations of EREW PRAM [20].

Merge proposal. For each block C_j with root r_j, we identify its neighboring block with the minimum root r_k such that there exists an edge $(u, v) \in E$ with $D(u) = r_j$ and $D(v) = r_k$. The strategy is simple: (a) for each $u \in V$, IncEval identifies the minimum $D(v) \neq D(u)$ such that $(u, v) \in E$, and records it in $q(u)$; and (b) for each block C_j, IncEval finds the minimum $q(u)$ for all $u \in C_j$, i.e., pointer $p(r_j)$. If $p(r_j) = \infty$, then the block with root r_j has no neighboring blocks, and we change $p(r_j)$ to r_j.

Part (a) can be implemented locally without the need for communication, shown as Branch MP(0) in Figure 4. It is nontrivial, however, to implement Part (b) efficiently. As each block C_j may be distributed across different workers, communication among the workers is inevitable.

We develop procedure concurrent minimization to implement Part (b), treated as branches in IncEval.

```
Branch MP(0):
1. for each $u \in V_i$ do
2.   $q(u) := \min\{D(v) \mid (u,v) \in E_i$ and $D(u) \neq D(v)\}$;
3.   Set up a pair $(D(u), q(u))$;
4. Push ('MP', 1) and ('SORT', 0) into $S$;

Branch MP(1):
1. for each $(D(u), q(u)) \in I_i$ do
2.   if $q(u) = \min\{q(v) \mid D(v) = D(u)\}$;
3.     Add $(D(u), q(u))$ to $w_{i,j}.x$ where $D(u) \in V_j$;
4. Push ('MP', 2) into $S$;

Branch MP(2):
1. for each $(r, q)$ received do
2.   $p(r) := q$;
3.   if $p(r) = \infty$ do
4.     $p(r) := r$;
5. Push ('TPTS', 0) into $S$;

Branch TPTS(0):
1. for each root node $r \in V_i$ do
2.   $D(r) := p(r)$;
3. Push ('MP', 1) and ('CA', 0, 1) into $S$;

Branch TPTS(1):
1. for each $u \in V_i$ do
2.   $D(u) := \min(D(u), p(D(u)))$;
3. for each $j$ do
4.   $w_{i,j}x := \{(u, D(u)) \mid u \in F_i.I \cap F_j.O, i \neq j\}$;
5. Push ('MP', 0) into $S$;

Branch CA(0):
1. for each $u \in V_i$ do
2.   Set up a pair $(D(u), u)$;
3. Push ('CA', 1, $s[2]$) and ('SORT', 0) into $S$;
```

(a)

```
Branch CA(1):
1. for each $(D(u), u) \in I_i$ do
2.   if $u = \min\{v \mid D(v) = D(u)\}$;
3.     Add $(D(u), i)$ to $w_{ij}.x$ where $D(u) \in V_j$;
4. Push ('CA' ,2, $s[2]$) into $S$;

Branch CA(2):
1. for each $(r, j)$ received do
2.   if $s[2] \leqslant \log|V|$ do
3.     Add $(D(r), r)$ to $w_{ij}.x$;
4.   else do
5.     Add $(p(r), r)$ to $w_{ij}.x$;
6. Push ('CA', 3, $s[2]$) into $S$;

Branch CA(3):
1. for each $(t, r)$ received do
2.   for each $(D(v), v) \in I_i$ where $D(v) = r$ do
3.     Update $(D(v), v)$ to $(t, v)$;
4.   if $r$ is the first element of the largest tuple in $I_i$;
5.     for each $j > i$ do
6.     $w_{ij}.x := \{(t, r)\}$;
7. Push ('CA', 4, $s[2]$) into $S$;

Branch CA(4):
1. for each $(t, r)$ received do
2.   for each $(D(u), u) \in I_i$ where $D(u) = r$ do
3.     Update $(D(u), u)$ to $(t, u)$;
4. for each $j$ do
5.   $w_{ij}.x := \{(t, u) \in I_i \mid u \in V_j\}$;
6. Push ('CA', 5, $s[2]$) into $S$;

Branch CA(5):
1. if s[2]$\leqslant \log|V|$ do
2.   for each $(t, u)$ received do
3.     $D(u) := D(D(u))$;
4.   Push ('CA', 0, $s[2]+1$) into $S$;
5. else do
6.   Record $t$ as $p(D(u))$.
```

(b)

Figure 4 Branches in IncEval for GC.

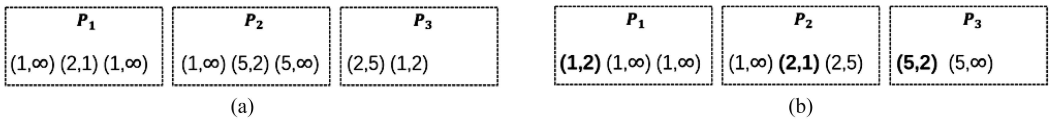

Figure 5 An example for the concurrent minimization procedure.

It employs a sorting algorithm along the same lines as how EREW PRAM simulates concurrent write operations [20]. Consider a set Γ of comparable items across different workers such that each worker hosts at most m items. The sorting algorithm of [29,45] completes the job in $O(m \cdot \log(mn))$ time under BSP. It can be adapted to GRAPE as a list of branches referred to as SORT, such that Branch SORT(i) simulates the i-th round in the BSP model. The switches among the branches of SORT are also controlled by the stack S: if SORT(i) is not the last branch of SORT, then in the execution of Branch SORT(i), we push ("SORT", $i+1$) into S. SORT contains $O(\log(mn)/\log m)$ branches, and the computational cost and communication cost for executing each branch are both $O(m \log m)$ [29,45].

More specifically, concurrent minimization is implemented by branches SORT, MP(1) and MP(2) (see Figure 4). Suppose that pairs $(D(u), q(u))$ are available for all $u \in V$. In Branch SORT, IncEval sorts the pairs into a lexicographically nondecreasing order such that each worker P_i holds a subset I_i of pairs. Then in Branch MP(1), P_i scans I_i and selects pairs $(D(u), q(u))$ such that $q(u)$ is the minimum among all those pairs with the same $D(u)$. It sends each such pair $(D(u), q(u))$ to the worker whose fragment contains the node $D(u)$. Finally, in Branch MP(2), for each message $(D(u) = r, q(u))$ received, worker P_i sets $p(r) = q(u)$, which is $\min\{q(v) \mid D(v) = r\}$. If $p(r) = \infty$, worker P_i changes $p(r)$ to r.

Example 3. Continuing with Example 2, we show how "merge proposal" works in Figure 5. Suppose $V_1 = \{1,2,3\}$, $V_2 = \{4,5,6\}$, and $V_3 = \{7,8\}$. Each worker P_i first computes $q(u)$ and sets up pair $(D(u), q(u))$ for each node u in its local V_i (Figure 5(a)). For example, by $D(2) = 2$ and $q(2) = 1$, the pair $(2,1)$ is set up by P_1. Then the pairs are sorted (Figure 5(b)). One pass of these sorted pairs in order suffices to identify all the pairs $(D(u), q(u))$ such that $q(u)$ is minimum among all the pairs with the same $D(u)$, marked in **bold**. These bold pairs are the pointers $(r, p(r))$ for all roots r (Figure 2(e)). □

Transforming pseudo-trees into stars. After setting $D(r) = p(r)$ for each root r, we obtain a pseudo-forest with nodes in V by Branch TPTS(0) (Figure 4(a)). Recall from Subsection 4.1.1 that each cycle in the pseudo-forest contains one or two directed edges. We transform these pseudo-trees to stars by performing $\log|V|$ rounds of pointer jumping, followed by concurrent access invocation (see below) to obtain $p(D(u))$ for each node u. Here each tuple s pushed into the stack S contains a new element $s[2]$, which records the number of concurrent access invocations in this phase. Next, we set $D(u)$ as $\min(D(u), p(D(u)))$ in Branch TPTS(1) such that all nodes in the same pseudo-tree point to the same root.

A nontrivial issue concerns how to let all nodes u simultaneously know $D(D(u))$ or $p(D(u))$, when the block is distributed across different workers. For example, when $u \in V_i$ wants to know to which node $D(u) \in V_j$ points, i.e., $D(D(u))$, communication between P_i and P_j is necessary.

To support this efficiently, we present a procedure concurrent access, which is implemented as branches in IncEval. This is carried out along the same lines as how EREW PRAM simulates concurrent read operations [19]. As shown in Figure 4(a), IncEval first sets up pair $(D(u), u)$ for each $u \in V_i$ in Branch CA(0), and sorts the pairs into a lexicographically nondecreasing order in Branch SORT. Subsequently, worker P_i holds a subset I_i of pairs. Then, the following branches are executed in turn:

Branch CA(1): Each worker P_i scans I_i and finds $(D(u), u)$ in which the second component is the minimum among all pairs with the same $D(u)$. It sends each $(D(u), i)$ to the worker where node $D(u)$ resides.

Branch CA(2): For each message (r, j) received, worker P_i sends $(D(r), r)$ back to P_j if $s[2] \leqslant \log|V|$, and it sends $(p(r), r)$ back to P_j otherwise.

Branch CA(3): Upon receiving (t, r), P_i changes tuple $(D(v), v) \in I_i$ to (t, v), where $D(v) = r$. If r is the first element of the last (i.e., the largest) tuple in I_i, P_i sends (t, r) to all workers P_j with $j > i$.

Branch CA(4): For each message (t, r) received, P_i changes tuple $(D(v), v) \in I_i$ to (t, v), where $D(v) = r$. Then for each tuple $(t, v) \in I_i$, P_i sends it to the worker where node u resides.

Branch CA(5): For each message (t, u) received, if $s[2] \leqslant \log|V|$, worker P_i changes $D(u)$ to $D(D(u))$ and pushes ("CA", 0, $s[2] + 1$) into S; otherwise ($s[2] = \log|V| + 1$) P_i records t as $p(D(u))$.

Example 4. Continuing with Example 3, we set $D(r) = p(r)$ and obtain a pseudo-forest formed by pointers $D(u)$, as depicted in Figure 6(a). The pseudo-forest contains only one pseudo-tree. After $1 \leqslant \log|V| = 3$ rounds of pointer jumping, the pointers $D(u)$ are shown in Figure 6(b). By setting $D(u) = \min\{D(u), p(D(u))\}$, we obtain the star-shaped structure depicted in Figure 2(f).

We next illustrate how concurrent access is conducted in Figure 7. Consider the pointers $D(u)$ of Figure 6(a) in which all nodes u need to access $D(D(u))$. Each worker P_i first sets up pair $(D(u), u)$ for each u in its local set V_i of nodes, as shown in Figure 7(a). The globally sorted pairs are given in Figure 7(b). Branch CA(1) performs one pass of the sorted pairs in order, and identifies all pairs $(D(u), u)$ such that u is the minimum among all pairs with the same $D(u)$. These pairs are $(1,2) \in I_1$, $(2,1) \in I_2$ and $(5,6) \in I_3$. Then worker P_1 sends $(1,1)$ to the worker where node 1 resides (i.e., itself in this case), where $(1,1)$ indicates that worker P_1 is accessing $D(1)$. Similarly, P_2 sends $(2,2)$ to P_1, and P_3 sends $(5,3)$ to P_2.

The rest of the actions are as follows (shown in Figure 7(c)): (a) In Branch CA(2), upon receiving $(1,1)$ and $(2,2)$, worker P_1 sends $(2,1)$ to P_1 (itself) and $(1,2)$ to P_2. Similarly, P_2 receives $(5,3)$, and sends $(2,5)$ to P_3. (b) In Branch CA(3), P_1 receives $(2,1)$ and updates I_1 to $\{(2,2),(2,3),(2,4)\}$. It then sends $(2,1)$ to P_2 and P_3. Worker P_2 receives $(1,2)$ and updates $(2,1),(2,5) \in I_2$ to $(1,1)$ and $(1,5)$, respectively; it then sends $(1,2)$ to P_3. Worker P_3 receives $(2,5)$ and updates $(5,6) \in I_3$ to $(2,6)$. (c) In Branch CA(4), P_1 sends $(2,2),(2,3) \in I_1$ to itself and sends $(2,4) \in I_1$ to P_2. Worker P_2 receives $(2,1)$ and updates $(1,8) \in I_2$ to $(2,8)$; it then sends $(1,1) \in I_2$ to P_1, $(1,5) \in I_2$ to itself, and $(2,8) \in I_2$ to P_3. Worker P_3 receives $(2,1),(1,2)$ and updates $(2,7) \in I_3$ to $(1,7)$; it then sends $(1,7)$ to itself and $(2,6)$ to P_2. □

(3) Assemble. When no further changes can be made, Assemble is triggered. It returns $D(u)$ for all

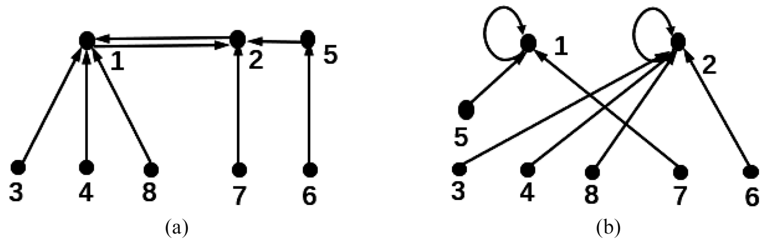

Figure 6 An example of pointer jumping.

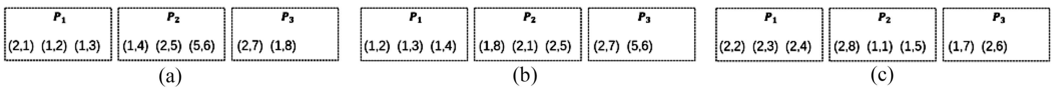

Figure 7 An example of the concurrent access procedure.

$u \in V$, where a pair of nodes u and v are connected if and only if $D(u) = D(v)$.

Analysis. We next show the correctness and parallel scalability of the PIE program.

For the correctness, observe the following: (a) Each node u and its root node $D(u)$ are connected in G. Hence, if u and v are not connected in G, then the PIE program finds that $D(u) \neq D(v)$. (b) The PIE program gradually merges the blocks, and terminates only if no more blocks can be merged. Hence, if u and v are connected in G, then the program guarantees to find that $D(u) = D(v)$.

We verify the scalability of the PIE program as follows:

Proposition 1. Under edge-cut partition $(F_1, \ldots, F_n)$ of G, the runtime of the PIE program for GC is $\tilde{O}(|G|/n)$ as long as $\max_i |F_i| = \tilde{O}(|G|/n)$ and $n \leqslant \sqrt{|G|}$. □

Proof. PEval takes $O(n)$ time to construct G_W in parallel (see the proof of Theorem 1) and to initialize the status variables $w_{ij}.x$. It takes $O(|F_i|)$ time for the rest. Hence the computational cost of each worker P_i is $O(n + |F_i|)$, i.e., $\tilde{O}(|G|/n)$. In addition, to exchange the $D(u)$s of all border nodes with its neighboring workers, P_i sends at most $|F_i.I| \leqslant |F_i|$ messages and receives $|F_i.O| \leqslant |F_i|$ messages. Therefore, the computation cost and the communication cost of each worker P_i are both $\tilde{O}(|G|/n)$. Thus the runtime of PEval is $\tilde{O}(|G|/n)$.

We next show that the runtime of IncEval is also $\tilde{O}(|G|/n)$. It suffices to prove the following three claims: (1) there are at most $\log |V|$ phases; (2) each phase invokes IncEval at most $O(\log^2 |V|)$ times; and (3) the computational cost and the communication cost of each invocation are both $\tilde{O}(|G|/n)$, regardless of which branch is executed. From the above points, it follows that IncEval is in $\tilde{O}(|G|/n)$ time.

(1) Consider a connected component C of G. In each phase, as long as the nodes of C are distributed in more than one block, each of the blocks of C merges with at least another block of C. Hence, the number of blocks is reduced by half. Thus, after at most $\log |V|$ phases, C ends up in a single block.

(2) Each phase executes each branch in MP or TPST only once, each in CA $\log |V| + 1$ times, and each in SORT $\log |V| + 2$ times. Because there are a constant number of branches in MP, TPTS, and CA, and at most $\log |V|$ branches in SORT, IncEval is invoked at most $O(\log^2 |V|)$ times in each phase.

(3) In each invocation of IncEval, each worker P_i takes $O(n)$ time to set each $w_{ij}.x$ to $\perp$ i.e., $O(|G|/n)$ time by $n \leqslant \sqrt{|G|}$. Moreover, one can check that the computational cost and communication cost for executing each of Branches MP(0), MP(2), TPTS(0), TPTS(1), CA(0), CA(2), CA(4), and CA(5) are $\tilde{O}(|G|/n)$. As shown previously, the computational cost and communication cost of each SORT(i) are also $\tilde{O}(|G|/n)$. What remains is the analysis of Branches MP(1), CA(1), and CA(3). For Branch MP(1), as the pairs in I_i are in order, P_i only needs $O(|I_i|)$ time to select the pairs $(D(u), p(u))$ such that $p(u)$ is the minimum among all pairs with the same $D(u)$; it then sends at most $|I_i|$ messages. One can verify that P_i also receives at most $|V_i|$ messages. Thus its computational cost and communication cost are both $O(\max_i |V_i|)$, i.e., $\tilde{O}(|G|/n)$ when $\max_i |F_i| = \tilde{O}(|G|/n)$. This analysis also applies to Branch CA(1). For Branch CA(3), it is easy to see that its computational cost is $\tilde{O}(|I_i|)$, each worker sends $n - i$ messages and receives i messages; hence the communication cost is at most $O(n)$, i.e., $O(|G|/n)$ by $n \leqslant \sqrt{|G|}$. □

4.2 Minimum spanning tree

Next, we develop another parallelly scalable PIE program, for finding minimum spanning trees.

Consider a connected undirected graph $G = (V, E, W)$, where $W : E \to \mathbf{R}$ assigns weights to edges. An MST T of G is a subgraph of G that connects all the nodes of V, without any cycles and with the minimum total edge weight $W(T) = \sum_{e \in T} W(e)$. To simplify the discussion, we assume that all edges have different weights, which guarantees the uniqueness of MST [46].

The MST problem is to compute, given a connected undirected graph $G = (V, E, W)$ as above, the minimum spanning tree of G. It is known that the MST problem can be solved in $O(|G| \log |G|)$ time. Like GC, the query class $\mathcal{Q}$ of queries for MST consists of a single query of constant size.

Below we first present the key ideas behind the PIE program and then give the algorithm.

4.2.1 *Algorithm sketch for minimum spanning tree*

The PIE program will use the following lemma verified in [19].

Lemma 1 (Lemma 5.4 in [19])**.** Let $G = (V, E, W)$ be a connected undirected graph, and $V = \bigcup V_i$ be an arbitrary partition of V. For each i, let e_i be the minimum-weight edge connecting a vertex in V_i to a vertex in $V - V_i$. Then all such edges e_i belong to an MST of the graph G. □

We develop the PIE program by converting a PRAM algorithm for MST [19], following Theorem 1. The algorithm of [19] begins with the forest $F_0 = (V, \emptyset)$, and runs in iterations to merge trees in the forest. Each iteration finds the minimum-weight edge incident on each tree, adds these new edges to the current forest F_s, and obtains a new forest F_{s+1}. This process proceeds until only a single tree remains.

This algorithm can be implemented along the same lines as the GC algorithm. The only major difference is the definition of pointer p. More specifically, each node u maintains a pointer $D(u)$ to the root of the tree to which it belongs, and each edge e carries a Boolean variable $B(e)$ recording whether e is in the current forest or not. Initially, for each node $u \in V$, $D(u) = u$, and for each edge e, $B(e) = 0$. Then each iteration adds new edges to the forest and merges some trees into a larger one. More specifically, for each tree T_j with root r_j, the algorithm selects the minimum-weight edge $e = (u, v) \in E$ such that $D(u) = r_j$ and $D(v) = r_k \neq r_j$. If so, we say that tree T_j proposes to merge with T_k and set $p(r_j) = r_k$ and $B(e) = 1$. By setting $D(r_j) = p(r_j)$ for each root r_j and treating $(u, D(u))$ as a directed edge for each node $u \in V$, we obtain a pseudo forest covering all the nodes in V. Similarly to its GC counterpart, one can verify that each cycle in each pseudo tree contains exactly two directed edges.

We transform the pseudo-forest into stars as in the GC algorithm, such that in each pseudo tree, the pointers $D(u)$ of all nodes u link to the same node, which is actually the root of the larger tree. This process continues until there is only a single tree. The algorithm returns $\{e \mid B(e) = 1\}$ as the MST.

Example 5. Given the weighted graph G depicted in Figure 8(a), the first iteration of the algorithm is illustrated Figures 8(b)–(d). Initially, each node is viewed as a tree and $D(i) = i$. As edge $(1, 4)$ has minimum weight among those adjacent to node 1, we set $p(1) = 4$. Similarly, we set pointers p for all the other nodes, as shown in Figure 8(b). By transforming pseudo-trees into stars, we obtain two trees shown in Figure 8(c), and their corresponding pointers D are depicted in Figure 8(d). □

4.2.2 *The* PIE *algorithm for minimum spanning tree*

We present our PIE program for MST in Figure 9. PEval initializes a tree with each node $u \in V$. IncEval iteratively merges trees into bigger ones until a single tree remains as the MST. It consists of multiple branches. Each worker maintains a stack S to control which switches to execute.

(1) PEval. As shown in Figure 9(a), PEval constructs graph G_W (see the proof of Theorem 1), and declares a status variable $w_{ij}.x$ for each node w_{ij} to store messages from P_i to P_j. It defines (a) a variable $D(u)$ for each node $u \in V_i$, initialized as $D(u) = u$; (b) a variable $B(e)$ for each edge $e \in E_i$, initialized as 0, to record whether e is in the current forest; and (c) a stack S as mentioned above.

It defines aggregate function $f_{\mathsf{aggr}}(x) = x$ as there is no conflict when updating status variables.

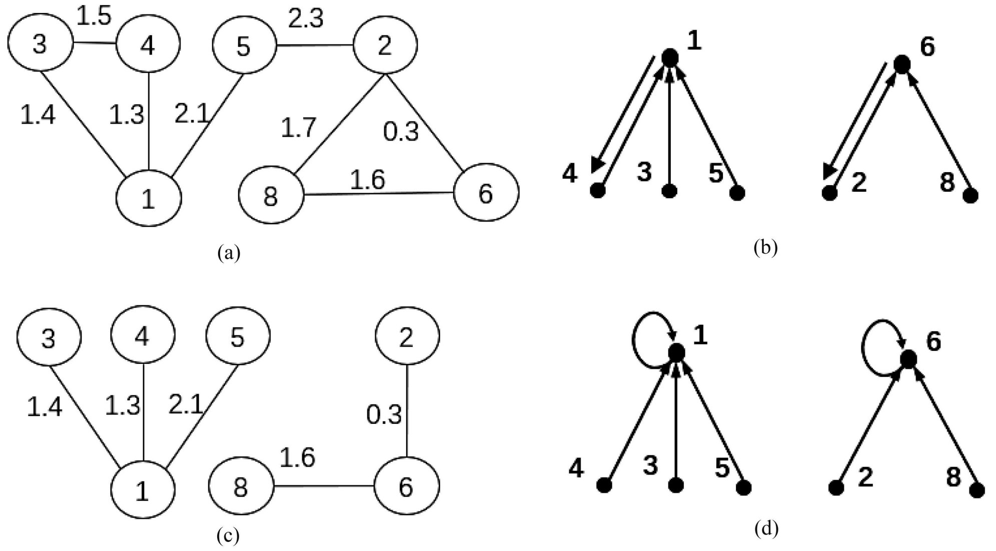

Figure 8 An example illustrating the PIE program for MST.

```
Input: A fragment $F_i$ of graph $G$.
Output: $Q(F_i)$ consists of $D(u)$ for all $u \in V_i$
        and $B(e)$ for all $e \in E_i$.

Message preamble: Construct $G_W$. For each node $w_{ij}$ in $G_W$,
        a variable $w_{ij}.x := \perp$ is declared;

1. Create a variable $D(u)$ for each $u \in V_i$, initialized as $u$;
2. Create a variable $B(e)$ for each $e \in E_i$, initialized as 0;
3. Create a stack $S$ and push the tuple ('MP', 0) into $S$.

Message segment: $M_{(i,j)} = w_{ij}.x$ if $w_{ij} \neq \perp$;
        $f_{\mathsf{aggr}}(x) = x$.
```

(a)

```
Input: A fragment $F_i$, partial results $Q(F_i)$, and
       message $M_i$.
Output: $Q(F_i \oplus M_i)$.

1. Set each $w_{ij}.x$ to $\perp$;
2. $s := S$.pop();
3. switch($s[0]$)
4.   Case 'MP': execute Branch MP($s[1]$);
5.   Case 'SORT': execute Branch SORT($s[1]$);
6.   Case 'TPTS': execute Branch TPTS($s[1]$);
7.   Case 'CA': execute Branch CA($s[1]$).
```

(b)

Figure 9 PIE program for MST. (a) PEval and (b) IncEval for MST.

(2) IncEval. As shown in Figure 9(b), IncEval consists of branches MP, SORT, TPTS, and CA. Here SORT, TPTS, and CA are the same as their GC counterparts. Each invocation of IncEval executes one branch, and the invocation is grouped into phases as in the PIE program for GC. Each phase is divided into two stages, for merge proposal and transforming pseudo-trees into stars, respectively, as follows:

Merge proposal. For each tree T_j, IncEval identifies the minimum-weight edge $e = (u, v)$ such that u is in T_j but v is not. It does the following: (a) For each node u, it finds the minimum-weight edge (u, v) such that $D(v) \neq D(u)$; it stores edge (u, v), pointer $D(v)$, and weight $W(u, v)$ in three variables $e(u)$, $p(u)$, and $w(u)$, respectively. (b) For each tree T_j with root r_j, IncEval identifies node u with the minimum $w(u)$ among all the nodes in T_j; then it updates $p(r_j)$ to $p(u)$ and $B(e(u))$ to 1.

As shown in Figure 10, these are implemented like their counterparts in the GC algorithm. In Branch MP(0), IncEval sets up a tuple $(D(u), w(u), p(u), e(u))$ for each $u \in V$. Then in Branch SORT, IncEval sorts the tuples into a lexicographically nondecreasing order, such that worker P_i holds a subset I_i of these tuples. In Branch MP(1), each worker P_i scans I_i and selects tuples $(D(u), w(u), p(u), e(u))$ such that $p(u)$ is the minimum among all tuples with the same $D(u)$; it then sends these $(D(u), w(u), p(u), e(u))$ to the worker where the node $D(u)$ resides and the worker where the node u resides. In Branch MP(2), for each received message $(D(u), w(u), p(u), e(u))$, worker P_i sets $p(D(u)) = p(u)$ and updates $B(e(u))$ to 1.

Transforming pseudo trees into stars. This stage is exactly the same as its counterpart in the PIE program for GC. By setting $D(r) = p(r)$ for each root r, we obtain a pseudo-forest formed by the pointers, connecting nodes of V. As remarked earlier, each cycle in the pseudo-forest contains exactly two directed edges. After performing $\log |V|$ rounds of pointer jumping and setting $D(u) = \min(D(u), p(D(u)))$, all nodes in the same pseudo tree point to the same node, i.e., the root. This yields a star shape.

```
Branch MP(0):
1. for each $u \in V_i$ do
2.    Find the minimum-weight edge $(u, v)$ such that $D(u) \neq D(v)$;
3.    $w(u) := W(u, v)$; $p(u) := D(v)$; $e(u) := (u, v)$;
4.    Set up a tuple $(D(u), w(u), p(u), e(u))$;
5. Push ('MP', 1) and ('SORT', 0) to $S$;

Branch MP(1):
1. for each $(D(u), w(u), p(u), e(u)) \in I_i$ do
2.    if $w(u) = \mathsf{min}\{w(v) \mid D(v) = D(u)\}$;
3.       Add $(D(u), w(u), p(u), e(u))$ to $w_{ij}.x$ where $D(u) \in V_j$ or $u \in V_j$;
4. Push ('MP', 2) to $S$;

Branch MP(2):
1. for each $(D(u), w(u), p(u), e(u))$ received do
2.    if $D(u) \in V_i$ do
3.       Set $p(D(u)) := p(u)$;
4.    if $e(u) \in E_i$ do
5.       Set $B(e(u)) := 1$;
6. Push ('TPTS', 0) to $S$.
```

Figure 10 Branches in IncEval for MST.

(3) Assemble. When no further changes can be made, Assemble returns $B(e)$ for each edge $e \in E$, such that the set $\{e \mid B(e) = 1\}$ of edges forms the MST of graph G.

Analysis. For the correctness of the PIE program, observe the following. (a) For each tree T_j, IncEval selects the minimum-weight edge $e = (u, v)$, where $u \in T_j$ and $v \notin T_j$, and updates $B(e)$ to 1; Lemma 1 guarantees that such edges belongs to MST. (b) The set $\{e : B(e) = 1\}$ is monotonically increasing, until it forms a tree. Thus, the PIE program will terminate and correctly return the final MST.

The parallel scalability of the PIE program is verified as follows.

Proposition 2. Under an edge-cut partition $(F_1, \ldots, F_n)$ of G, the runtime of the PIE program for MST is $\tilde{O}(|G|/n)$ as long as $\max_i |F_i| = \tilde{O}(|G|/n)$ and $n \leqslant \sqrt{|G|}$. □

Proof. First, it is easy to see that the computational cost and communication cost of PEval are $\tilde{O}(|G|/n)$ and 0, respectively. Second, in each phase, if there is more than one tree, each tree merges with at least another one. Hence the number of trees is reduced by half. Thus the number of phases is at most $\log |V|$. Finally, similar to the proof of Proposition 1, one can verify that in each phase, (a) IncEval is invoked at most $O(\log |V|^2)$ times, and (b) the computational cost and communication cost of each invocation are both $\tilde{O}(|G|/n)$. From these points, it follows that the runtime of the PIE program is $\tilde{O}(|G|/n)$. □

5 Open research issues

We have shown that it is possible to parallelize existing sequential graph algorithms and guarantee the convergence of the parallelized computations under a generic condition. We have also shown that parallel scalability is within the reach of PTIME problems. However, the study of parallel graph computations has raised as many questions as it has answered. Below we suggest three directions for future study.

(1) *Classification.* Can we effectively determine whether algorithms are parallelly scalable? In the presence of such a characterization, we can classify our algorithms and allocate resources to the ones that can make effective use of the resources. Theorem 1 is a simple condition that helps us identify some parallelly scalable algorithms, but it does not tell us what algorithms may not capitalize on additional resources.

(2) *Hierarchy.* One step further, can we classify which computational problems are parallelly scalable? We want to identify complete problems for the class of parallelly scalable algorithms, i.e., the "hardest ones" in the class. We also want to define reductions to reduce our problems to ones that we know how to solve on shared-nothing systems. These depart from the classical complexity theory, since some intractable problems are parallelly scalable but some tractable ones are not. A hierarchy for parallel scalability is nontrivial to develop, since it has to incorporate both parallel computational cost and communication cost.

(3) *Incrementalization.* Is it possible to incrementalize existing batch algorithms, analogous to how we parallelize sequential algorithms? The need for incremental algorithms is evident not only for GRAPE

but also for coping with the velocity of big data, by reducing computations on big data to computations to small changes. However, although many batch algorithms are in place, few incremental algorithms have been developed, and fewer have performance guarantees. This suggests that we need to develop a systematic method to deduce incremental algorithms from batch ones, and ideally, ensure that the cost of the incremental algorithms depends only on the size of changes instead of the size of the entire big dataset.

Concerning GRAPE, the following issues remain open and require further study.

(1) *Programming with* GRAPE. GRAPE aims to parallelize existing sequential algorithms and simplify parallel programming. This said, programming with GRAPE still requires domain knowledge of algorithm design, to declare update parameters and design an aggregate function. An immediate topic for future work is to develop an interactive user interface to help users deduce update parameters and aggregate functions from single-machine algorithms, and make GRAPE even easier to use.

(2) *Graph partitioning for* GRAPE. As remarked earlier, although GRAPE works regardless of graph partitions, the choice of partitioning strategies may have an impact on not only the performance of GRAPE, but also the design of the PIE programs. For an algorithm of our interest, what partitioning strategy fits it the best and improves its parallel execution? It has been shown that the traditional criteria for evaluating graph partitions, e.g., load balancing and replication, may not work best for a given application [34]. Moreover, is it possible to develop graph algorithms with partition transparency, such that the algorithms work under different partitions without changes? In addition, how should we incrementally partition graphs in response to updates while retaining the partition quality? While we have proposed an application-driven partitioning strategy [34] and an incrementalization method [47] to address these issues, the problems deserve a full treatment [34], especially to cope with multiple applications that run on the same graph.

(3) *Dynamic scaling and streaming updates.* In the real world, e-commerce systems often experience load surges, as triggered by, e.g., holidays and unexpected events. This gives rise to a natural question: how many processors should we use to configure GRAPE? Obviously, it is too costly to maintain sufficient resources just to meet peak requirements. Then, how should we adaptively scale GRAPE out and in, i.e., add and remove processors when load jumps up and down, respectively, to improve resource utilization and reduce costs? We have done preliminary work on the issue [48]. However, much more needs to be done, e.g., to adjust graph partitions in response to load surges without interrupting ongoing computations.

(4) *Uniform optimization schemes.* As remarked in Section 2, GRAPE is able to inherit existing optimization techniques developed for single-machine graph algorithms. However, graph computations are often costly. To this end, a variety of optimization strategies have been developed for speeding up graph computations, e.g., indexing, compression, and graph summarization. Prior studies have typically targeted an individual application. However, multiple applications often run on the same graph in practice. It is too costly and even infeasible to build, e.g., a separate indexing structure for each of these applications. Is it possible to develop a generic and uniform optimization scheme that is capable of speeding up different applications at the same time, without the need to make changes and without loss of information? This remains an important open question for all graph systems, and is not limited to GRAPE.

Acknowledgements This work was supported in part by Shenzhen Institute of Computing Sciences, Beijing Advanced Innovation Center for Big Data and Brain Computing (Beihang University), Royal Society Wolfson Research Merit Award (Grant No. WRM/R1/180014), European Research Council (Grant No. 652976), and Engineering and Physical Sciences Research Council (Grant No. EP/M025268/1).

References

1 Malewicz G, Austern M H, Bik A J C, et al. Pregel: a system for large-scale graph processing. In: Proceedings of International Conference on Management of Data, 2010
2 Low Y, Gonzalez J, Kyrola A, et al. Distributed GraphLab: a framework for machine learning in the cloud. Proc VLDB Endow, 2012, 5: 716–727

3 Tian Y Y, Balmin A, Corsten S A, et al. From “think like a vertex” to “think like a graph”. Proc VLDB Endow, 2013, 7: 193–204

4 Wang G Z, Xie W L, Demers A J, et al. Asynchronous large-scale graph processing made easy. In: Proceedings of Conference on Innovative Data Systems Research, 2013

5 Xie C N, Chen R, Guan H B, et al. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2015

6 Kruskal C P, Rudolph L, Snir M. A complexity theory of efficient parallel algorithms. Theory Comput Sci, 1990, 71: 95–132

7 Fan W F, Wang X, Wu Y H, et al. Association rules with graph patterns. Proc VLDB Endow, 2015, 8: 1502–1513

8 Fan W F, Hu C M, Liu X L, et al. Discovering graph functional dependencies. In: Proceedings of International Conference on Management of Data, 2018. 427–439

9 Fan W F, Lu P, Tian C, et al. Deducing certain fixes to graphs. Proc VLDB Endow, 2019, 12: 752–765

10 Fan W F, Wang X, Wu Y H, et al. Distributed graph simulation: impossibility and possibility. Proc VLDB Endow, 2013, 7: 1083–1094

11 Papadimitriou C H. Computational Complexity. Boston: Addison-Wesley, 1994

12 Fan W F, Yu W Y, Xu J B, et al. Parallelizing sequential graph computations. In: Proceedings of International Conference on Management of Data, 2017. 495–510

13 Fan W F, Lu P, Luo X J, et al. Adaptive asynchronous parallelization of graph algorithms. In: Proceedings of International Conference on Management of Data, 2018. 1141–1156

14 Yan D, Bu Y Y, Tian Y Y, et al. Big graph analytics platforms. FNT Databases, 2015, 7: 1–195

15 Raychev V, Musuvathi M, Mytkowicz T. Parallelizing user-defined aggregations using symbolic execution. In: Proceedings of Symposium on Operating Systems Principles, 2015

16 Pingali K, Nguyen D, Kulkarni M, et al. The tao of parallelism in algorithms. In: Proceedings of Programming Language Design and Implementation, 2011

17 Zhou Y, Liu L, Lee K, et al. Fast iterative graph computation with resource aware graph parallel abstractions. In: Proceedings of High Performance Distributed Computing, 2015

18 Radoi C, Fink S J, Rabbah R M, et al. Translating imperative code to MapReduce. In: Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications, 2014

19 JáJá J. An Introduction to Parallel Algorithms. Boston: Addison-Wesley, 1992

20 Karp R M, Ramachandran V. Parallel algorithms for shared-memory machines. In: Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity. Amsterdam: Elsevier, 1990

21 Chong K W, Han Y J, Lam T W. On the parallel time complexity of undirected connectivity and minimum spanning trees. In: Proceedings of Symposium on Discrete Algorithms, 1999. 225–234

22 Tarjan R E, Vishkin U. An efficient parallel biconnectivity algorithm. SIAM J Comput, 1985, 14: 862–874

23 Chong K W, Nikolopoulos S D, Palios L. An optimal parallel co-connectivity algorithm. Theory Comput Syst, 2004, 37: 527–546

24 Brent R P. The parallel evaluation of general arithmetic expressions. J ACM, 1974, 21: 201–206

25 Spencer T H. Time-work tradeoffs for parallel algorithms. J ACM, 1997, 44: 742–778

26 Harris T J. A survey of PRAM simulation techniques. ACM Comput Surv, 1994, 26: 187–206

27 Herley K T, Bilardi G. Deterministic simulations of prams on bounded degree networks. SIAM J Comput, 1994, 23: 276–292

28 Karloff H J, Suri S, Vassilvitskii S. A model of computation for MapReduce. In: Proceedings of Symposium on Discrete Algorithms, 2010. 938–948

29 Goodrich M T, Sitchinava N, Zhang Q. Sorting, searching, and simulation in the MapReduce framework. In: Proceedings of International Symposium on Algorithms and Computation, 2011. 374–383

30 Yan D, Cheng J, Xing K, et al. Pregel algorithms for graph connectivity problems with performance guarantees. Proc VLDB Endow, 2014, 7: 1821–1832

31 Andreev K, Racke H. Balanced graph partitioning. In: Proceedings of ACM Symposium on Parallel Algorithms and Architectures, 2006

32 Bourse F, Lelarge M, Vojnovic M. Balanced graph edge partition. In: Proceedings of Knowledge Discovery and Data Mining, 2014. 1456–1465

33 Fan W F, Liu M Y, Xu R Q, et al. Think sequential, run parallel. In: Proceedings of Symposium on Real-Time and Hybrid Systems — Essays Dedicated to Professor Chaochen Zhou on the Occasion of His 80th Birthday, 2018

34 Fan W F, Jin R C, Liu M Y, et al. Application driven graph partitioning. In: Proceedings of ACM SIGMOD International Conference on Management of Data, 2020

35 Henzinger M R, Henzinger T, Kopke P. Computing simulations on finite and infinite graphs. In: Proceedings of Foundations of Computer Science, 1995

36 Fan W F, Li J Z, Ma S, et al. Graph pattern matching: from intractability to polynomial time. Proc VLDB Endow, 2010, 3: 264–275

37 Fan W F, Wang X, Wu Y H. Incremental graph pattern matching. In: Proceedings of International Conference on Management of Data, 2011

38 Valiant L G. A bridging model for parallel computation. Commun ACM, 1990, 33: 103–111

39 Jones N D. An introduction to partial evaluation. ACM Comput Surv, 1996, 28: 480–503

40 Ramalingam G, Reps T. On the computational complexity of dynamic graph problems. Theory Comput Sci, 1996,

158: 233–277

41 Fan W F, Hu C M, Tian C. Incremental graph computations: doable and undoable. In: Proceedings of International Conference on Management of Data, 2017. 155–169

42 Maon Y, Schieber B, Vishkin U. Parallel ear decomposition search (EDS) and st-numbering in graphs. Theory Comput Sci, 1986, 47: 277–298

43 Miller G L, Ramachandran V. Efficient parallel ear decomposition with applications. 1986. http://www.cs.cmu.edu/~glmiller/Publications/Papers/MillerRamachandran86.pdf

44 Bang-Jensen J, Gutin G Z. Digraphs: Theory, Algorithms and Applications. Berlin: Springer, 2008

45 Goodrich M T. Communication-efficient parallel sorting. SIAM J Comput, 1999, 29: 416–432

46 Gallager R G, Humblet P A, Spira P M. A distributed algorithm for minimum-weight spanning trees. ACM Trans Program Lang Syst, 1983, 5: 66–77

47 Fan W F, Liu M Y, Tian C, et al. Incrementalization of graph partitioning algorithms. Proc VLDB Endow, 2020, 13: 1261–1274

48 Fan W F, Hu C M, Liu M Y, et al. Dynamic scaling for parallel graph computations. Proc VLDB Endow, 2019, 12: 877–890