Graph Algorithms with Partition Transparency

Wenfei Fan Muyang Liu Ping Lu Qiang Yin

Abstract—Graph computations often have to be conducted in parallel on partitioned graphs. The choice of graph partitioning strategies, however, has strong impact on the design of graph computation algorithms. A graph algorithm developed under edge-cut partitions may not work correctly under vertex-cut, and vice versa. We often have to rewrite our algorithms when we switch from, *e.g.*, edge-cut to vertex-cut. To cope with this, we propose a notion of *partition transparency*, such that graph algorithms are able to work correctly under different partitions without changes and moreover, benefit from recent hybrid partitions to speed up computations. Furthermore, we identify conditions under which graph algorithms are guaranteed to be partition-transparent, in graph-centric and vertex-centric models. We show that a variety of graph algorithms can be made partition-transparent. Using real-life and synthetic graphs, we experimentally verify that partition-transparent algorithms compute correct answers under different partitions; better still, under hybrid partitions these algorithms perform better than algorithms tailored for edge-cut and vertex-cut partitions in efficiency.

Index Terms—graph partition, partition transparency, graph-centric algorithms, vertex-centric algorithms

1 Introduction

To handle large-scale real-life graphs, it is often necessary to conduct parallel computations on partitioned graphs. The idea is to cut a large graph G into smaller fragments and distribute the fragments to a cluster of processors (*a.k.a.* workers), such that computations on G can be conducted in parallel by the processors on their local fragments, subject to message passing among different processors.

A variety of graph partitioning algorithms (*a.k.a.* partitioners) have been developed. These partitioners are often either *edge-cut* [8], [30], which evenly partitions vertices and cuts edges, or *vertex-cut* [11], [25], [31], which evenly partitions edges by replicating vertices. There have also been recent work on *hybrid* partitioners, which cut both edges and vertices [20], [14], [17], [52], [33], [9], [50], [43], to overcome the limitations of edge-cut and vertex-cut partitions. It has been shown that the hybrid partitioners often make graph computations faster than edge-cut and vertex-cut.

The choice of a partitioning strategy has strong impact on the performance of graph algorithms. Neither vertex-cut nor edge-cut consistently outperforms the other for different algorithms. Worse still, an algorithm developed under edge-cut may not work correctly under vertex-cut, and vice versa. Hence, when developing a parallel algorithm for a graph computation problem, one has to decide in advance which partitioning strategy to use. If the strategy picked does not work well, we may want to switch to the other method, but then we may have to rewrite our algorithms. Moreover, algorithms developed for edge-cut and vertex-cut often do not work correctly under hybrid partitions, and hence cannot benefit from the state-of-the-art hybrid partitioners.

- Wenfei Fan is with Shenzhen Institute of Computing Sciences, University of Edinburgh, and BDBC, Beihang University Email: wenfei@inf.ed.ac.uk.
- Muyang Liu is with University of Edinburgh Email: muyang.liu@ed.ac.uk.
- Ping Lu (corresponding author) is with SKLSDE, Beihang University Email: luping@buaa.edu.cn
- Qiang Yin is with Shanghai Jiao Tong University and Alibaba Group Email: q.yin@sjtu.edu.cn

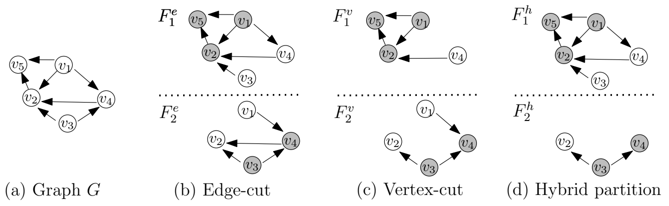


Fig. 1: Graph partitions

Example 1: Consider the common neighbor problem (CN), which computes the number of common neighbors for each pair of vertices in a graph. It is widely used in link prediction, product recommendation and fraud detection [35], [16]. For a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, denote by $\Gamma^+(v)$ (resp. $\Gamma^-(v)$) the set of outgoing (resp. incoming) neighbors of vertex v in V. To simplify the discussion, here we consider common *outgoing neighbors*, *i.e.*, given G, CN is to compute $\mathsf{CN}(u, v) = |\Gamma^+(u) \cap \Gamma^+(v)|$ for all pairs $(u, v) \in V \times V$.

Consider graph G shown in Figure 1 and its partitions (F_1^e, F_2^e) and (F_1^v, F_2^v) under edge-cut (Figure 1(a)) and vertex-cut (Figure 1(b)), respectively. Observe the following. (1) Edge-cut partitioning cuts edges to generate vertex disjoint fragments, *e.g.*, $\{v_1, v_2, v_5\}$ and $\{v_3, v_4\}$. A vertex is assigned to only one fragment [8], [30] (colored in gray in Figure 1), referred to as *a master vertex*; the others are *mirrors*, *e.g.*, v_3 in F_2^e is a master vertex and v_3 in F_1^e is a mirror. Each master and all of its incident edges reside in the same fragment. (2) In contrast, vertex-cut partitioning cuts vertices to produce edge disjoint partitions. Thus all edges incident to a master vertex may not be necessarily assigned to the same fragment, *e.g.*, v_2 is a master vertex in F_1^v and its edge set at F_1^v does not include the edge (v_3, v_2), which resides in fragment F_2^v (see Figure 1(b)).

Algorithms developed for edge-cut partitions may not work under vertex-cut partitions, and vice versa. To see these, let us consider the following algorithms.

(1) Edge-cut. Under edge-cut, an algorithm $\mathcal{A}_e$ for CN works as follows. (a) For each master vertex v in a fragment F_k^e, it increases the local count $\mathsf{CN}_k(v_i, v_j)$ for each incoming

neighbor pair (v_i, v_j) of v. (b) After this, $\mathcal{A}_e$ collects the local CN counts from all fragments, and computes their sum for all vertex pairs (v_i, v_j). It returns the aggregate values as the final result. One can verify that $\mathcal{A}_e$ correctly returns count $\mathsf{CN}(v_1, v_3) = 2$ in the partition of Figure 1(a). Indeed, $\mathsf{CN}(v_1, v_3)$ is set to be 1 after step (a) in each fragment, and is then increased to 2 after the aggregation in step (b).

However, $\mathcal{A}_e$ erroneously returns $\mathsf{CN}(v_1, v_3)$=1 when it runs on the vertex-cut partition of Figure 1(b). That is the count in fragment F^v_2 due to the common neighbor v_4. It misses the common neighbor v_2 for v_1 and v_3 since the edge (v_3, v_2) is missing for master vertex v_2 in fragment F^v_1.

(2) Vertex-cut. Under vertex-cut, an algorithm $\mathcal{A}_v$ works in three steps. (a) It first computes the local count CN for each master vertex v. (b) It then collects all incoming edges of v from other fragments, and increases the CN counts using these received edges. Denote by $\Gamma^-_\Delta(v)$ the set of incoming neighbors of v adjacent to received edges. Then $\mathcal{A}_v$ increases $\mathsf{CN}(v_i, v_j)$ for each $v_i \in \Gamma^-(v)$ and $v_j \in \Gamma^-_\Delta(v)$, where v_j is a vertex on a received edge. Note that $\mathcal{A}_v$ has all incoming edges $\Gamma^-(v)$ of v after collecting edges from other fragments. (c) Finally, it collects and aggregates local CN counts from all fragments. One can verify that A_v correctly computes CN given the vertex-cut partition in Figure 1(b). For instance, $\mathcal{A}_v$ first increases the $\mathsf{CN}(v_1, v_3)$ in fragment F^v_1 to 1, and then increases it to 2 after receiving (v_3, v_2).

In contrast, algorithm $\mathcal{A}_v$ does not work correctly when given the edge-cut partition of Figure 1(a). To see this, observe that $\mathcal{A}_v$ would return $\mathsf{CN}(v_1, v_3) = 4$, which is wrong. This is because there exist duplicated edges in the edge-cut partition (*e.g.,* (v_1, v_4) and (v_3, v_2) are in both F^e_1 and F^e_2); after collecting such edges from other fragments, $\mathcal{A}_v$ counts $\mathsf{CN}(v_1, v_3)$ twice for each duplicated edge.

The problem with algorithm $\mathcal{A}_e$ is that it assumes a master vertex to have all its edges in its local fragment, which is not true under vertex-cut. Algorithm $\mathcal{A}_v$ gets wrong answers on edge-cut partitions due to the duplicated edges, which do not exist under vertex-cut partitions.

(3) Hybrid partition. For the same reason, neither algorithm $\mathcal{A}_e$ nor $\mathcal{A}_v$ works under the hybrid partition (F^h_1, F^h_2) of Figure 1(c). More specifically, master vertex v_4 in fragment F^h_2 does not have all of its incident edges in F^h_2, and edge (v_3, v_2) replicates in F^h_1 and F^h_2. As a result, $\mathcal{A}_e$ finds $\mathsf{CN}(v_1, v_3)$=1, and $\mathcal{A}_v$ gets $\mathsf{CN}(v_1, v_3)$=3. None is correct. $\square$

This example gives rise to several questions. Is it possible to make an algorithm $\mathcal{A}$ *transparent* to different partitions of a graph G, *i.e.,* it works correctly no matter how G is partitioned? If so, we do not have to rewrite our algorithms when, *e.g.,* switching from edge-cut partitions to vertex-cut; and better still, we can capitalize on the state-of-the-art hybrid partitions to speed up graph computations. Another question concerns under what conditions algorithms are transparent to different partitions? Moreover, is it within the reach in practice to develop transparent algorithms?

Contributions & organization. This paper aims to answer the questions above, all in the affirmative. We consider parallel graph-centric [23], [7] and vertex-centric [25], [36] algorithms, which will be reviewed in Section 2.

(1) Partition transparency (Section 3). We introduce a notion of partition transparency for parallel graph algorithms. A *partition-transparent* algorithm $\mathcal{A}$ works correctly under both edge-cut and vertex-cut without requiring any change to $\mathcal{A}$. Better yet, they work correctly under hybrid partitions, and hence are able to reduce the cost of graph computations by leveraging, *e.g.,* application-driven partitions [20].

(2) Transparency conditions (Section 4). We identify conditions for algorithm $\mathcal{A}$ to be guaranteed *partition-transparent*, *i.e.,* $\mathcal{A}$ works correctly under edge-cut, vertex-cut and hybrid partitions without requiring any change to $\mathcal{A}$. We provide such conditions for both graph-centric programs of GRAPE [23] and vertex-centric GAS programs of PowerGraph [25].

(3) Transparent algorithms (Section 5). We show that partition-transparent algorithms are within the reach of a variety of problems, including common neighbor (CN), single source shortest path (SSSP), weakly connected component (WCC), PageRank (PR), strongly connected components (SCC), and maximum cliques (MaxClique). We show that these algorithms work correctly no matter what partitions are given.

(4) Experimental study (Section 6). Using real-life and synthetic graphs, we verify the effectiveness and efficiency of partition-transparent algorithms. We find the following. (a) Transparent algorithms work correctly regardless of what partitions are adopted, without changes, in both graph-centric PIE mode and vertex-centric GAS model. (b) Transparent algorithms $\mathcal{A}$ under hybrid partitions of [20] are on average 2.3 times faster than non-transparent $\mathcal{B}$ under vertex-cut or edge-cut, while $\mathcal{B}$ may not work correctly under hybrid partitions. (c) Even when all algorithms run under vertex-cut and edge-cut, transparent $\mathcal{A}$ performs comparably to $\mathcal{B}$ developed for vertex-cut and edge-cut. The performance gap is less than 5.8%. (d) Under hybrid partitions, transparent algorithms scale well with both the size of graphs and the number of processors used, *e.g.,* transparent WCC and PR take on average 66.5s on graphs of 500 million nodes and 6 billion edges with 90 processors.

Related work. This paper extends its conference version [20] as follows. (1) While [20] targets hybrid partitioners, this paper focuses on partition-transparent algorithms. We have substantially reorganized and rewritten a large part of the paper, from motivation and examples to technical discussions (Sections 1–7). (2) We have provided a detailed analysis of partition transparency conditions for graph-centric algorithms, from examples to proofs; we have also developed new transparency conditions for vertex-centric GAS programs (Section 4). (3) We have developed new partition-transparent algorithms as proof of concept (Section 5: SSSP, WCC, SCC and MaxClique). (4) The experimental study is almost entirely new, and evaluates partition-transparent algorithms with more cases and datasets (Section 6).

We discuss the other related work as follows.

A host of graph partitioners have been developed for edge-cut and vertex-cut (see [13], [10] for surveys). Edge-cut (resp. vertex-cut) aims to (a) partition vertices (resp. edges) into disjoint subsets of even sizes for load balancing, and (b) reduce replicated edges (resp. vertices). Popular edge-cut partitioners include exact algorithms [8], [32] such as METIS [28], [29] and its parallel version ParMETIS [27], as well as parallel heuristics XtraPuLP [44] and stream partitioner FENNEL [46]. Vertex-cut partitioners include spectral algorithm of [41] and heuristics Grid [26], SHEEP [38], NE [49] and HDRF [40], just to name a few.

Edge-cut promotes locality: for each vertex v in graph G, it keeps all edges emanating from v in the same fragment; however, it often leads to imbalanced workload, especially when G is skewed. In contrast, vertex-cut makes it easier to balance partitions, but may have a lower level of locality and increase communication cost for high-degree vertices.

To rectify these limitations, hybrid partitioners have been studied, which are neither pure edge-cut nor pure vertex-cut. PowerLyra [14] and IOGP [17] combine edge-cut and vertex-cut by cutting only high-degree vertices, controlled by a user-defined threshold. TopoX [33] not only splits high-degree vertices, but also merges neighboring low-degree vertices into super nodes to prevent splitting such vertices. Gemini [52] and MDBGP [9] balance hybrid workload by combining vertex and edge loads based on a balancing metric. Gluon [18] implements four hybrid partitioning strategies by restricting the outgoing and incoming edges of master and mirrors. CUBE [50], [34] first partitions properties of vertices, and generates duplicated graphs with different sets of properties; then it partitions each duplicated graph with vertex-cut. GraBi [43] combines partitioning strategies such as Hybrid-cut [14], Bi-cut [15], and 3D-partitioner [50], to improve the performance on bipartite graphs.

In particular, an application-driven partitioning strategy was proposed in [20]. Given an algorithm $\mathcal{A}$, it learns a cost model of $\mathcal{A}$ beyond balance and replication; it generates a hybrid partition guided by the cost model, and speeds up parallel execution of $\mathcal{A}$ and reduces the cost of the execution.

As opposed to the previous work, this work studies partition-transparent algorithms, *i.e.,* the behaviors of graph algorithms under various partitions, rather than to develop yet another partitioner. We are not aware of any prior work that has considered partition-transparent algorithms.

(1) It is the first study of partition-transparent algorithms. While some systems (*e.g.,* PowerLyra) support hybrid partitions, algorithms on these systems are essentially vertex-cut or edge-cut, and the underlying systems take charge of communication to ensure the correctness of these algorithms; *e.g.,* the PR algorithm on PowerLyra is the same as the one in PowerGraph (a vertex-cut system), and PowerLyra has to differentiate the processing of high-degree vertices and low-degree vertices to ensure the correctness. In contrast, partition transparency allows us to develop graph algorithms without worrying about what partitioning strategy to use.

(2) We provide the first conditions under which algorithms are transparent to underlying partitions, for both graph-centric and vertex-centric parallel models. The conditions help us determine what algorithms are transparent and also guide us to develop transparent algorithms.

(3) Moreover, we show that transparent algorithms are able to capitalize on the state-of-the-art application-driven partitioners and speed up graph computations. In contrast, algorithms tailored for edge-cut and vertex-cut partitions may not even work correctly under hybrid partitions.

Symbols	Notations
G, V, E	graph, vertex set and edge set of G
$F_i.O, \mathcal{F}.O$	border nodes of fragment F_i and graph G
$\mathcal{Q}, Q, Q(G)$	query class, a query, and query result
$\mathsf{HP}(n)$	a n-way hybrid partition
R_i^r	partial results in round r at worker P_i
$\mathcal{A}(Q, \mathsf{HP}(n))$	query result by algorithm $\mathcal{A}$ over $\mathsf{HP}(n)$
$\preceq$	partial order on query results
$\mathcal{A}_e/\mathcal{A}_v/\mathcal{A}_h^e, \mathcal{A}_h^v$	transparent alg. $\mathcal{A}$ under e-cut/v-cut/hybrid par.

TABLE 1: Notations

2 Parallel Graph Programming

We will study partition transparency for graph-centric algorithms and vertex-centric algorithms. Hence in this section we start with a review of these two graph programming models. We first consider the graph-centric model of GRAPE (Section 2.1), an open-source parallel system [23], [7]. We then review the vertex-centric GAS model of PowerGraph [25] (Section 2.2). Transparent graph algorithms in other parallel graph programming models can also be developed. The notations of the paper are summarized in Table 1.

We consider graphs $G = (V, E)$, where V is a finite set of vertices, and $E \subseteq V \times V$ is its set of edges.

2.1 Graph-Centric Programming

Consider a class $\mathcal{Q}$ of queries, *i.e.,* a graph computation problem. Given a query $Q \in \mathcal{Q}$ and a graph G, we want to compute the set $Q(G)$ of answers to Q in G.

PIE algorithms [23]. To develop a parallel algorithm for $\mathcal{Q}$ under GRAPE, one only needs to specify three functions.

(1) PEval: A *sequential* algorithm that given a query $Q \in \mathcal{Q}$ and a graph G, computes the answer $Q(G)$ to Q in G.

(2) IncEval: A *sequential incremental* algorithm that given Q, G, $Q(G)$ and updates ΔG to G, computes updates ΔO to the old output $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$, where $G \oplus \Delta G$ denotes G updated by ΔG [42].

(3) Assemble: A function that collects partial answers computed locally at each worker by PEval and IncEval, and assembles the partial results into complete answer $Q(G)$. This function is typically straightforward.

The three functions are referred to as *a* PIE *program for* $\mathcal{Q}$ (PEval, IncEval and Assemble). PEval and IncEval can be any *existing sequential* (incremental) algorithms for $\mathcal{Q}$.

The only additions are the following declarations in function PEval, which are shared by IncEval.

(a) Update parameters. PEval declares (a) a set C_i of vertices in fragment F_i as the *update region* of F_i (see Example 5); and (b) *status variables* $\bar{x}$ for C_i. We denote by $C_i.\bar{x}$ the set of *update parameters* of F_i, *i.e.,* the status variables associated with the vertices in C_i. As will be seen shortly, $C_i.\bar{x}$ marks candidates to be updated by incremental steps of IncEval.

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

We defer examples of PIE programs to Sections 3 and 4, where we can specify update parameters *w.r.t.* partitions.

Parallel computation. GRAPE executes a PIE program via data-partitioned parallelism. It works with n share-nothing workers $P_1, \ldots, P_n$ and a master P_0. It partitions graph G into n fragments $(F_1, \ldots, F_n)$ by adopting an existing partitioner, and distributes the fragments to workers such that fragment F_i resides in worker P_i for $i \in [1, n]$.

Upon receiving a query $Q \in \mathcal{Q}$ at master P_0, GRAPE posts Q to all workers and computes $Q(G)$ as follows. To simplify the discussion, we present the parallel computation under the Bulk Synchronous Parallel (BSP) model [47], which separates the computation into supersteps, and terminates when no more change can be made. Note that GRAPE also works under asynchronous models [22].

(1) Partial evaluation (**PEval**). In the first superstep, GRAPE computes partial results $R_i^0 = \mathsf{PEval}(Q, F_i)$ in fragment F_i at each worker P_i by invoking function PEval, in parallel ($i \in [1, n]$). Here R_i^r denotes *partial results* in superstep r at worker P_i. At the end of the superstep, worker P_i sends the set $C_i.\bar{x}$ of update parameters to master P_0 as a message.

For each status variable $x \in C_i.\bar{x}$, master P_0 collects a multi-set S_x of values from messages of all workers. It computes $x_{\mathsf{aggr}} = f_{\mathsf{aggr}}(S_x)$ by applying the aggregate function f_{aggr} declared in PEval, to resolve conflicts. It generates message M_i to worker P_i, which includes only those $f_{\mathsf{aggr}}(S_x)$'s such that $f_{\mathsf{aggr}}(S_x) \neq x$, *i.e.*, only the *changed* values of the update parameters of fragment F_i.

(2) Incremental computation (**IncEval**). In superstep $r + 1$, upon receiving message M_i from master P_0, each worker P_i invokes function IncEval to *incrementally* compute $R_i^{r+1} = \mathsf{IncEval}(Q, R_i^r, F_i, M_i)$ by *treating message M_i as updates*, in parallel for $i \in [1, n]$. It refines its partial results R_i^r based on the information of M_i from other workers. At the end of the superstep, P_i sends a message to P_0 that consists of *updated values* of $C_i.\bar{x}$, if any. After receiving messages from all workers, master P_0 deduces a message M_i just like in PEval. It sends M_i to worker P_i in the next superstep.

(3) Termination (**Assemble**). The computation terminates when it reaches a fixpoint, *i.e.*, $R_i^{r+1} = R_i^r$ for all $i \in [1, n]$. At this point, GRAPE invokes Assemble at P_0, which pulls partial results from all workers, takes a union and then aggregates them to the final result at P_0, denoted by $\mathcal{A}(Q, G)$.

As shown in [23], under a generic contracting condition, the execution of a PIE program guarantees to terminate and return correct query answer $Q(G)$ to query Q in graph G.

2.2 Vertex-Centric Programming

Unlike graph-centric programming that conducts computations directly on (sub)graphs (*i.e.*, fragments), vertex-centric programming requires users to *think like a vertex* [39] and write vertex programs. A vertex program is "pivoted" at a vertex; it may only directly access information on the current vertex, adjacent vertices and adjacent edges [37], [25].

Several vertex-centric programming models are in place, notably the Pregel model [37] and the GAS model of PowerGraph [25]. Below we present the GAS model; the results of the paper can be adapted to other vertex-centric models.

GAS model. The GAS programming model introduces a three-phases abstraction, namely, *Gather, Apply and Scatter*. Given a graph G, a GAS algorithm $\mathcal{B}$ iteratively executes the three phases at each vertex, in parallel at all vertices, until no more changes can be made [25]. More specifically, $\mathcal{B}$ updates the status D_v of vertex v in each iteration as follows.

```
gather($D_v$, $D_u$, $D_{(u,v)}$)
    return $D_{(u)}$.rank/$d^+_G(u)$;

sum($x$, $y$)
    return $x + y$;

apply($D_v$, $a_v$)
    $rank\_new = d \cdot a_v + (1 - d)$;
    $D_v.\delta = rank\_new - D_v$.rank;
    $D_v$.rank = $rank\_new$;

scatter($D_v$, $D_u$, $D_{(v,u)}$)
    if $|D_v.\delta| > \epsilon$ then activate($v$)
    return $D_v.\delta$;
```

Fig. 2: PageRank algorithm in GAS model [25]

(1) Gather phase. In this phase, each vertex first gathers the status of its adjacent vertices and edges by function gather, and then "sums up" the status via function sum. Let D_v (resp. $D_{(v,u)}$) be the status of a vertex v (resp. an edge (v, u)). The functions gather and sum are defined as follows.

$$\begin{aligned} a'_v &= \mathsf{gather}(D_v, D_u, D_{(v,u)}), \\ a_v &= \mathsf{sum}(a_v, a'_v). \end{aligned}$$

In the GAS model, the sum function is required to be associative and commutative [25]. It loops over all incident edges of a vertex v to aggregate the gathered values. This phase generates a summarized value a_v for each vertex v.

(2) Apply phase. This phase updates each vertex status D_v of v to D_v^{new}, using the summarized values a_v generated in the gather phase. The apply function is defined as follows.

$$D_v^{\mathsf{new}} = \mathsf{apply}(D_v, a_v).$$

(3) Scatter phase. With the new status D_v^{new}, each vertex updates the status $D_{(v,u)}^{\mathsf{new}}$ for each adjacent edge (v, u) of v via function scatter, which can be defined as follows.

$$D_{(v,u)}^{\mathsf{new}} = \mathsf{scatter}(D_v^{\mathsf{new}}, D_u, D_{(v,u)}).$$

The new edge status $D_{(v,u)}^{\mathsf{new}}$ is then processed in the gather phase of subsequent iterations. The entire process terminates when no more changes can be made, and it returns the collection of all vertices' status as the final result.

The GAS model adopts vertex-cut [25]. When a vertex v is cut, multiple copies of v may reside in multiple fragments. In an iteration, each copy of v first gathers locally and sends its update to the master copy of v for aggregation via function sum; then the master copy of v runs apply function and sends updates back to all its mirrors; finally, function scatter is executed in parallel on all copies of v.

Similar to algorithms for CN in Example 1, vertex-centric algorithms are not necessarily partition-transparent.

Example 2: Consider PageRank for ranking Web pages. Given a directed graph $G = (V, E)$, PageRank iteratively updates a score $\mathsf{rank}(v) = d * \sum_{(u,v) \in G} \mathsf{rank}(u)/d^+_G(u) + (1-d)$ for each $v \in G$, where d is a damping factor in the range of $[0, 1]$, and $d^+_G(u)$ is the out-degree of v in G.

A PageRank algorithm in the GAS model is shown in Figure 2. In each iteration, each vertex v first aggregates the ranking scores of its neighbors along its incoming edges in the gather phase; it then applies the aggregated result to update its own ranking score D_v.rank; it also computes score changes $D_v.\delta$ for termination decision, which is scattered in the scatter phase. The process terminates when the changes generated from all edges are blow a pre-defined threshold ϵ.

The algorithm correctly computes PageRank scores under vertex-cut. However, it does not work under edge-cut. Indeed, observe that in the gather phase, each vertex accumulates changes from its incoming edges (see the definition of sum). Since there are edge replications in edge-cut partitions, *e.g.*, (v_2, v_3) resides in both F_1^e and F_2^e of the edge-cut partition shown in Figure 1(a), the changes of such replicated edges would be added multiple times, which yields incorrect scores. This does not happen under vertex-cut since there is no edge replication there. Hence this PageRank algorithm is not "transparent" to partitions. □

3 Partition Transparency

In this section, we first formulate hybrid partitions, which subsume edge-cut and vertex-cut partitions as special cases. We then introduce the notion of partition transparency.

Hybrid partition. Given a natural number n, a *n-way hybrid partition* $\mathsf{HP}(n) = (F_1, \ldots, F_n)$ of a graph G divides G into n small fragments $F_1, \ldots, F_n$ such that (a) $F_i = (V_i, E_i)$, (b) $V = \bigcup_{i=1}^{n} V_i$, and (c) $E = \bigcup_{i=1}^{n} E_i$. We refer to $\mathsf{HP}(n)$ as a hybrid partition of G when n is clear in the context.

Our familiar edge-cut partitions [8], [30] and vertex-cut partitions [25], [31] are special cases of hybrid partitions. To see this, we use the following notations. Denote by E^v (resp. E_i^v) the set of edges incident to vertex v in G (resp. F_i).

(1) A vertex v is *v-cut* in $\mathsf{HP}(n)$ if the set of edges incident to v is not "complete" at any F_i, *i.e.*, $E^v \neq E_i^v$ ($\forall i \in [1, n]$).

(2) A vertex v is *e-cut* if there exists a fragment F_i such that all edges incident to v are included in F_i, *i.e.*, $E_i^v = E^v$. If there are multiple copies of v in $\mathsf{HP}(n)$, we refer to its copy in F_i as an *e-cut node* and the others as its *mirrors*.

(3) Denote by $F_i.O = \{v \in V_i \mid \exists j (v \in V_j \wedge i \neq j)\}$ the set of *border nodes* of F_i. Intuitively, a border node is replicated among fragments. We denote $\mathcal{F}.O = \bigcup_{i=1}^{n} F_i.O$. For each vertex $v \in \mathcal{F}.O$, we designate one copy of v as its *master*.

Example 3: Consider graph G in Figure 1 and its hybrid partition (F_1^h, F_2^h) in Figure 1(c). Observe the following.

(1) Vertex v_4 is v-cut since edge (v_3, v_4) is missing from fragment F_1^h, and (v_4, v_2) is missing from F_2^h.

(2) Vertex v_2 is e-cut since its all incident edges reside in F_1^h together with v_2; vertex v_2 in fragment F_1^h is the master while the copy of v_2 in F_2^h is a mirror node. Similarly v_3 is e-cut. Vertices v_1 and v_5 are also e-cut since they are not replicated and their incident edges are all kept locally.

(3) $F_1^h.O$=$F_2^h.O$=$\{v_2, v_3, v_4\}$; thus $\mathcal{F}^h.O$=$\{v_2, v_3, v_4\}$. □

One can easily verify the following.

(1) Partition $\mathsf{HP}(n)$ is *edge-cut* if (a) all vertices are e-cut; and (b) the e-cut node sets of the fragments are pairwise disjoint.

(2) Partition $\mathsf{HP}(n)$ is *vertex-cut* if the edge sets are disjoint, *i.e.*, $E_i \cap E_j = \emptyset$ for $i \neq j$, while v-cut nodes are replicated.

Example 4: Consider the partitions depicted in Figure 1.

(1) The partition (F_1^e, F_2^e) of Figure 1(a) is an edge-cut partition of graph G given in Figure 1, since (a) all vertices of G are e-cut; and (b) the e-cut node sets of F_1^e and F_2^e are disjoint, *i.e.*, $\{v_1, v_2, v_5\} \cap \{v_3, v_4\} = \emptyset$. Here the e-cut nodes are colored in gray, and the other vertices are mirrors.

(2) Partition (F_1^v, F_2^v) of Figure 1(b) is a vertex-cut partition of G, since the edge sets of F_1^v and F_2^v are disjoint.

(3) In contrast, partition (F_1^h, F_2^h) of Figure 1(c) is a hybrid partition, and it is neither edge-cut (since vertex v_4 is v-cut), nor vertex-cut (since edge (v_3, v_2) is replicated). □

Partition Transparency. PIE programs have been developed for both edge-cut [23] and vertex-cut partitions [21]. Under different partitions, these programs differ in their update parameters and hence aggregate functions. More specifically,

- under vertex-cut, update region C_i ($i \in [1, n]$) is typically the set of v-cut nodes of fragment F_i.
- In contrast, under edge-cut, C_i is the set of vertices that are incident to cut edges of F_i.

In light of the difference, a PIE program developed for edge-cut may not work under vertex-cut, and vice versa (Example 1); similarly for vertex-centric algorithms (Example 2). Such algorithms may not work under hybrid partitions.

Partition transparency. Consider algorithm $\mathcal{A}$ for a class $\mathcal{Q}$ of queries, either a PIE program or a vertex-centric algorithm. Denote by $\mathcal{A}(Q, \mathsf{HP}(n))$ the result of running algorithm $\mathcal{A}$ on a hybrid partition $\mathsf{HP}(n)$ of a graph G.

We say that $\mathcal{A}$ is *partition-transparent* if given any query $Q \in \mathcal{Q}$ and any hybrid partition $\mathsf{HP}(n)$ of a graph G, $\mathcal{A}(Q, \mathsf{HP}(n)) = Q(G)$, *i.e.*, $\mathcal{A}$ correctly computes the answer $Q(G)$ to query Q given the partition $\mathsf{HP}(n)$ of G.

Intuitively, a transparent algorithm $\mathcal{A}$ works correctly under edge-cut and vertex-cut partitions without requiring any change to $\mathcal{A}$. Hence we can uniformly use the same algorithm without worrying about the choice of graph partitioning strategies. Better still, we can run $\mathcal{A}$ under partitions of the state-of-the-art hybrid strategies that improve edge-cut and vertex-cut, and speed up query answering of $\mathcal{Q}$.

Example 5: Recall CN stated in Example 1. We present a partition-transparent PIE program for CN.

As shown in Example 1, the difficulty for a partition-transparent algorithm concerns how to compute $\mathsf{CN}(u, v)$ when not all incoming edges of u or v are in the same fragment, and when edges are replicated in multiple fragments.

To cope with these, we adopt the following strategies. (1) For each border node v, PEval gathers all its incoming edges in its local fragment, and synchronizes copies of v among all fragments. After this, v has all its incoming edges in the same fragment. (2) To avoid repeatedly increasing $\mathsf{CN}(\cdot, \cdot)$ for the same common neighbor on replicated edges, IncEval only increases $\mathsf{CN}(\cdot, \cdot)$ for the "new edges" from other fragments, *i.e.*, IncEval incrementally updates $\mathsf{CN}(\cdot, \cdot)$.

PIE *algorithm*. We present the PEval algorithm and IncEval algorithm for CN in Figures 3 and 4, respectively.

(1) PEval. Consider an arbitrary hybrid partition $\mathsf{HP}(n)$ =

```
Input: A fragment $F_i(V_i, E_i)$.
Output: A set $Q(F_i)$ consisting of $\mathsf{CN}(u, v)$ for $u, v \in V_i$.

1.  for each $u_1, u_2 \in V$ and $u_1 \neq u_2$ do
2.      $\mathsf{CN}(u_1, u_2) := 0$;
3.  for each vertex $v \in V_i$ do
4.      for each $u_1, u_2 \in \Gamma^-(v)$ and $u_1 \neq u_2$ do
5.          $\mathsf{CN}(u_1, u_2) := \mathsf{CN}(u_1, u_2) + 1$;
6.      if $v \in F_i.O$ then
7.          $\Gamma^-(v) := \{u \mid (u, v) \in E_i\}$; $\Gamma^+(v) := \{u \mid (v, u) \in E_i\}$;
8.  $Q(F_i) := \{\mathsf{CN}(u_1, u_2) \mid u_1, u_2 \in V_i\}$;

Message segment: $M_{(i)} := \{(v, \Gamma_i^-(v), \Gamma_i^+(v)) \mid v \in F_i.O\}$;
                 $f_{\mathsf{aggr}}(v) := (v, \cup\, \Gamma_j^-(v), \cup\, \Gamma_j^+(v))$;
```

Fig. 3: PEval for CN

```
Input: A fragment $F_i(V_i, E_i)$, partial result $Q(F_i)$, and message $M_i$.
Output: New output $Q(F_i \oplus M_i)$.

1.  $\Delta := \emptyset$;
2.  for each $(v, \Delta\Gamma_j^-(v), \Delta\Gamma_j^+(v)) \in M_i$ do
3.      $\Delta\Gamma^-(v) := \Delta\Gamma^-(v) \cup \Delta\Gamma_j^-(v)$;
4.      for each $u \in \Delta\Gamma_j^+(v)$ do
5.          $\Delta\Gamma^-(u) := \Delta\Gamma^-(u) \cup \{v\}$;
6.      $\Delta := \Delta \cup \{v\} \cup \Delta\Gamma_j^-(v) \cup \Delta\Gamma_j^+(v)$;
7.  for each vertex $v \in \Delta$ do
8.      for each $u_1 \in \Delta\Gamma^-(v) \backslash \Gamma^-(v)$ and $u_2 \in \Gamma^-(v) \cup \Delta\Gamma^-(v)$ do
9.          if $u_1 \neq u_2$ then $\mathsf{CN}(u_1, u_2) := \mathsf{CN}(u_1, u_2) + 1$;
10. $Q(F_i) := \{\mathsf{CN}(u_1, u_2) \mid u_1, u_2 \in V_i \cup \Delta\}$;
```

Fig. 4: IncEval for CN

$(F_1, \ldots, F_n)$ of graph G. At each fragment F_i, PEval first defines the updated region C_i as the set of the border nodes, *i.e.,* the nodes that are replicated among fragments; and for each vertex v, PEval declares a status variable $v.x = (v, \Gamma^-(v), \Gamma^+(v))$ to maintain both incoming neighbors $\Gamma^-(v)$ and outgoing neighbors $\Gamma^+(v)$. Intuitively, we use $v.x$ to collect all neighbors of v from other fragments, and synchronize this information among all copies of v.

In the first superstep, PEval computes $\mathsf{CN}(\cdot, \cdot)$ using vertices in F_i, and collects neighbors of border nodes. More specifically, it does the following (see Figure 3): (a) for each vertex v, it first increases $\mathsf{CN}(u_1, u_2)$ for its incoming neighbors u_1 and u_2 in F_i (lines 4-5); then (b) for each vertex v in $F_i.O$, $v.x$ collects its local incoming and outgoing neighbors (lines 6-7). It collects $\mathsf{CN}(\cdot, \cdot)$ for all vertices in F_i (line 8). After that it sends the status variable $v.x$ of each border node v in F_i to master P_0 for aggregating the counts of common neighbors across different fragments.

When PEval terminates, the master P_0 collects remote changes to the neighbor sets of each border node u of F_i, and disseminates them as messages (recall Section 2). To resolve conflicts, the aggregate function f_{aggr} takes the union of $u.x$ for border nodes across different fragments.

(2) IncEval. Upon receiving message M_i from master P_0, IncEval incrementally updates the partial result $Q(F_i)$ at each fragment F_i (see Figure 4). It consists of two stages.

(a) Preprocessing. IncEval first complements the incoming neighbors of each vertex using the received message M_i. More specifically, (i) for each $(v, \Delta\Gamma_j^-(v), \Delta\Gamma_j^+(v)) \in M_i$, IncEval extends the accumulated incoming neighbors $\Delta\Gamma^-(v)$ of v (line 3). (ii) For each outgoing neighbor u of v in $\Delta\Gamma_j^+(v)$, IncEval extends the incoming neighbors $\Delta\Gamma^-(u)$ of u with v (lines 4-5). Note that for any pair (u_1, u_2) of vertices in F_i, IncEval has all outgoing edges of u_1 and u_2 after the preprocessing, and thus can correctly update $\mathsf{CN}(u_1, u_2)$.

(b) Updates. Then for each received vertex v, IncEval incrementally updates $\mathsf{CN}(\cdot, \cdot)$ using "new" incoming neighbors u_1 of v (lines 7-9). More specifically, for each vertex $u_1 \in \Delta\Gamma^-(v) \setminus \Gamma^-(v)$, *i.e.,* when u_1 is not in the local fragment F_i but has an edge to v in other fragments, IncEval increases $\mathsf{CN}(u_1, u_2)$ for all incoming neighbors u_2 of v.

(3) Assemble. At the end of the process of IncEval, no message is sent and the computation terminates. That is, IncEval is executed only once for CN. At this point, Assemble simply takes a union of partial result $Q(F_i)$ in each fragment F_i, which is the final result $Q(G)$.

One can verify that the PIE program is partition-transparent. Indeed, PEval collects all incoming and outgoing edges of each border node (lines 6-8 of PEval), and IncEval only updates $\mathsf{CN}(\cdot, \cdot)$ with new incoming neighbors (lines 8-9 of IncEval). Thus duplicated edges from multiple fragments do not lead to repeated CN counts, and all incoming edges of each vertex are taken into account. □

This example raises a natural question. Do there exist generic conditions under which an algorithm is guaranteed partition-transparent? Such conditions could guide us to systematically develop partition-transparent algorithms.

4 Conditions for Partition Transparency

In this section, we provide sufficient conditions under which graph algorithms are guaranteed partition-transparent. We first give the conditions for graph-centric PIE programs, and then revise the conditions for vertex-centric GAS algorithms.

4.1 Conditions for Graph-Centric Algorithms

Consider a PIE program $\mathcal{A}$ = (PEval, IncEval, Assemble) for a class $\mathcal{Q}$ of queries. To specify the conditions we use the following notations. (a) Partial results R_i^l consist of a set of status variables (see Section 2.1 for R_i^l). We assume the existence of a partial order $\preceq$ on R_i^l: for each status variable y in fragment F_i, let $\leq_s$ be a partial order on the domain of y; then we say that $R_i^l \preceq R_i^j$ if $y.\mathsf{val}^l \leq_s y.\mathsf{val}^j$ for each status variable y in fragment F_i, where $y.\mathsf{val}^l$ and $y.\mathsf{val}^j$ denote the values of y in partial results R_i^l and R_i^j in F_i, respectively. (b) Denote by $G_1 \sqsubseteq G_2$ if graph G_1 is a subgraph of G_2.

We say that an algorithm $\mathcal{A}$ is *monotonic* if for all queries $Q \in \mathcal{Q}$ and graphs G_1 and G_2, if $G_1 \sqsubseteq G_2$ then $\mathcal{A}(Q, G_1) \preceq \mathcal{A}(Q, G_2)$ by the partial order on query results.

In the graph-centric PIE model, to ensure the monotonicity of a PIE program $\mathcal{A}$, we require both PEval and IncEval to be monotonic [22]. The function PEval (resp. IncEval) is *monotonic* if for all queries $Q \in \mathcal{Q}$ and graphs G_1 and G_2, if $G_1 \sqsubseteq G_2$ (resp. $R_i^s \preceq R_i^t$) then $\mathsf{PEval}(Q, G_1) \preceq \mathsf{PEval}(Q, G_2)$ (resp. $R_i^{s+1} \preceq R_i^{t+1}$). Here R_i^s and R_i^t denote partial results in (possibly different) runs of the PIE program $\mathcal{A}$.

We say that an algorithm $\mathcal{A}$ is *correct under vertex-cut* if for all queries $Q \in \mathcal{Q}$, all graphs G and any vertex-cut partition $\mathsf{HP}(n)$ of graph G, $\mathcal{A}(Q, \mathsf{HP}(n)) = Q(G)$.

Example 6: We show that the PIE algorithm $\mathcal{A}$ for CN given in Section 3 is monotonic and is correct under vertex-cut.

(1) To see that algorithm $\mathcal{A}$ is monotonic, we define partial order $\preceq$ as follows. Denote by R_i (resp. $\mathcal{R}_i'$) the common neighbor counts for all vertex pairs of G computed in fragment F_i (resp. F_i'). We say that $R_i \preceq R_i'$ if $\mathsf{CN}_i(u, v) \leq \mathsf{CN}_i'(u, v)$ for each pair (u, v) in G. Here $\mathsf{CN}_i(u, v)$ and

$\mathsf{CN}'_i(u,v)$ are the CN counts in F_i and F'_i, respectively. When $F_i \sqsubseteq F'_i$, *i.e.*, F_i is a subgraph of F'_i, then $\mathsf{CN}_i(u,v) \leq \mathsf{CN}'_i(u,v)$. Indeed, there are more edges in F'_i than F_i; thus the same vertex w may contribute more to $\mathsf{CN}'_i(\cdot,\cdot)$ than to $\mathsf{CN}_i(\cdot,\cdot)$ in the processes of PEval and IncEval.

(2) The PIE algorithm $\mathcal{A}$ works correctly under vertex-cut because given any vertex-cut partition $\mathsf{HP}(n)$, each vertex v contributes exactly 1 to the count $\mathsf{CN}(u,w)$, where u and w are incoming neighbors of v in graph G. Note that no edge is duplicated in a vertex-cut partition, and no duplicate is counted by $\mathcal{A}$ since v increases $\mathsf{CN}(u,w)$ at most once. Moreover, no count is missed by $\mathcal{A}$. To see this, suppose that two incoming edges (u,v) and (w,v) of v reside in different fragments. Then both edges will be shipped to the fragment where the master copy of v resides. After that, IncEval increases $\mathsf{CN}(u,w)$ by 1 (see lines 7-9 of Figure 4). □

Transparency condition. We now present two sufficient conditions for a PIE program $\mathcal{A}$ to be partition-transparent.

P1. Algorithm $\mathcal{A}$ is *monotonic*.

P2. Algorithm $\mathcal{A}$ is *correct under vertex cut*.

The theorem below shows that these two conditions suffice to guarantee the partition transparency of PIE programs.

Theorem 1: *A* PIE *algorithm $\mathcal{A}$ is partition-transparent if $\mathcal{A}$ satisfies conditions* P1 *and* P2. □

Proof: We prove that if $\mathcal{A}$ satisfies both P1 and P2, then for any graph G, any query $Q \in \mathcal{Q}$ and any hybrid partition $\mathsf{HP}(n)$ of G, we have that $\mathcal{A}(Q,\mathsf{HP}(n)) = Q(G)$. To do so, we construct two partitions $\mathsf{HP}^u(n)$ and $\mathsf{HP}^l(n)$ of G such that (a) $\mathcal{A}$ correctly computes $Q(G)$ on both partitions $\mathsf{HP}^u(n)$ and $\mathsf{HP}^l(n)$, *i.e.*, $\mathcal{A}(Q,\mathsf{HP}^l(n)) = \mathcal{A}(Q,\mathsf{HP}^u(n)) = Q(Q)$; and (b) $\mathcal{A}(Q,\mathsf{HP}(n))$ is lower bounded by $\mathcal{A}(Q,\mathsf{HP}^l(n))$ and upper bounded by $\mathcal{A}(Q,\mathsf{HP}^u(n))$, *i.e.*, $\mathcal{A}(Q,\mathsf{HP}^l(n)) \preceq \mathcal{A}(Q,\mathsf{HP}(n)) \preceq \mathcal{A}(Q,\mathsf{HP}^u(n))$. Thus $\mathcal{A}(Q,\mathsf{HP}(n)) = Q(G)$.

Let $\mathsf{HP}(n) = (F_1,\ldots,F_n)$. We next construct partitions $\mathsf{HP}^u(n)$ and $\mathsf{HP}^l(n)$ and verify their properties one by one.

(1) We start with the construction of $\mathsf{HP}^l(n)$ and show that $Q(G) = \mathcal{A}(Q,\mathsf{HP}^l(n)) \preceq \mathcal{A}(Q,\mathsf{HP}(n))$. Note that in the hybrid partition $\mathsf{HP}(n)$ there may exist duplicated edges among fragments. Let $\mathsf{HP}^l(n) = (F'_1,\ldots,F'_n)$ be a vertex-cut partition of G obtained by removing these duplicated edges from $\mathsf{HP}(n)$. Since $\mathcal{A}$ is correct under vertex-cut (condition P2), we have that $Q(G) = \mathcal{A}(Q,\mathsf{HP}^l(n))$.

It remains to show that $\mathcal{A}(Q,\mathsf{HP}^l(n)) \preceq \mathcal{A}(Q,\mathsf{HP}(n))$. Observe that by the construction, we have that $F'_i \sqsubseteq F_i$ for $i \in [1,n]$, since we only remove edges from $\mathsf{HP}(n)$, where F'_i is the fragment in $\mathsf{HP}^l(n)$ that corresponds to fragment F_i in $\mathsf{HP}(n)$. By the monotonicity of $\mathcal{A}$ (condition P1), we have the following: (a) $R_i^{0'} = \mathsf{PEval}(Q,F'_i) \preceq \mathsf{PEval}(Q,F_i) = R_i^0$, and (b) $R_i^{t+1'} \preceq R_i^{t+1}$ for $i \in [1,n]$ and $t \geq 0$. From these it follows that $Q(G) = \mathcal{A}(Q,\mathsf{HP}^l(n)) \preceq \mathcal{A}(Q,\mathsf{HP}(n))$.

(2) We continue with the construction of $\mathsf{HP}^u(n)$, and verify that $\mathcal{A}(Q,\mathsf{HP}(n)) \preceq \mathcal{A}(Q,\mathsf{HP}^u(n))$ and $\mathcal{A}(Q,\mathsf{HP}^u(n)) = Q(G)$. Since both PEval and IncEval of $\mathcal{A}$ are monotonic (P1), to upper bound $\mathcal{A}(Q,\mathsf{HP}(n))$, we can replicate vertices and edges to add to each fragment of $\mathsf{HP}(n)$, and meanwhile ensure that $\mathcal{A}(Q,\mathsf{HP}^u(n))=Q(G)$. In particular, we define $\mathsf{HP}^u(n) = (G,\ldots,G)$, which duplicates G for n times.

It remains to show that $\mathcal{A}(Q,\mathsf{HP}(n)) \preceq \mathcal{A}(Q,\mathsf{HP}^u(n)) = Q(G)$. Note that $F_1,\ldots,F_n$ are subgraphs of G, *i.e.*, $F_i \sqsubseteq G$ for $i \in [1,n]$. By condition P1 and an argument similar to (1) above one can show that $\mathcal{A}(Q,\mathsf{HP}(n)) \preceq \mathcal{A}(Q,\mathsf{HP}^u(n))$. We next show that $\mathcal{A}(Q,\mathsf{HP}^u(n)) = Q(G)$ under the GRAPE model (Section 2.1). Observe the following: (a) since each fragment contains the entire graph G, after PEval terminates the partial result at each fragment F_i is $Q(G)$; (b) IncEval is not invoked since all fragments yield the same partial result; and (c) Assemble simply takes a union of the partial results and aggregates these results, which returns $Q(G)$ by the semantics of Assemble. Therefore, $\mathcal{A}(Q,\mathsf{HP}^u(n)) = Q(G)$. □

Example 7: As shown in Example 6, the PIE algorithm for CN (Section 3) satisfies conditions P1 and P2. Hence it is partition-transparent by Theorem 1, and correctly computes CN under edge-cut, vertex-cut and hybrid partitions. □

4.2 Conditions for Vertex-Centric Algorithms

We next consider conditions that make GAS algorithms partition-transparent. Conditions P1 and P2 do not suffice to ensure the partition transparency for GAS. Consider the GAS algorithm for PR shown in Section 2.2. It is not partition-transparent, but it is both (1) monotonic when the initial values are 0 [51], and (2) correct under vertex-cut [25].

A closer look at the proof of Theorem 1 reveals that the proof relies on a property of graph-centric PEval, *i.e.*, it is able to compute $Q(G)$ directly in a given graph G. This is an inherent property of PIE computation model. GAS algorithms do not necessarily have this property (see Section 2.2), since they only gather information of neighbors of vertices in the gather phase. To ensure the partition transparency, we need one more condition as follows.

P3. Algorithm $\mathcal{A}$ is *correct when graphs are replicated*.

Here algorithm $\mathcal{A}$ is *correct when graphs are replicated* if for all queries $Q \in \mathcal{Q}$ and all graphs G, $\mathcal{A}(Q,\mathsf{HP}^d(n)) = Q(G)$, where $\mathsf{HP}^d(n) = (G,\ldots,G)$ that duplicates G.

Corollary 2: *A vertex-centric algorithm $\mathcal{B}$ of the GAS model is partition-transparent if $\mathcal{B}$ satisfies conditions* P1, P2 *and* P3. □

Proof: Similar to the proof of Theorem 1, we build partitions $\mathsf{HP}^l(n)$ and $\mathsf{HP}^u(n)$ to bound $\mathcal{B}(Q,\mathsf{HP}(n))$.

(1) $\mathsf{HP}^l(n)=(F'_1,\ldots,F'_n)$ is a vertex-cut by removing duplicated edges from $\mathsf{HP}(n)$. It follows from conditions P1 and P2 that $Q(G) = \mathcal{B}(Q,\mathsf{HP}^l(n)) \preceq \mathcal{B}(Q,\mathsf{HP}(n))$.

(2) $\mathsf{HP}^u(n)=\mathsf{HP}^d(n)$. By P3, $\mathcal{B}(Q,\mathsf{HP}^u(n))=Q(G)$. Using condition P1, we have that $\mathcal{B}(Q,\mathsf{HP}(n))\preceq\mathcal{B}(Q,\mathsf{HP}^u(n))$. □

Example 8: One can verify that the vertex-centric algorithm for PR in Section 2.2 does not satisfy condition P3, since in the gather phase, it directly adds received values from other fragments to the sum a_v, which is incorrect when there exist duplicated edges. We will provide a partition-transparent PIE algorithm for PR (Section 5.3), which employs a global replication count to handle such duplicates. □

Remarks. We remark following about conditions P1-P3.

(1) Conditions P1-P3 specify properties of parallel graph computations under different partitioning strategies: P1 re-

quires the computation to be monotonic, while P2 and P3 ask for the correctness under vertex-cut and a special hybrid partition (*i.e.,* when graphs are replicated), respectively. They impose no constraint on the computation model itself.

(2) Conditions P1-P3 can be extended to ensure the partition transparency of other parallel graph computation models beyond PIE and GAS. For example, (a) Giraph++ [45] is a graph-centric model, which can run sequential algorithms in the first superstep, and then vertex-centric algorithms in the following steps. If an algorithm $\mathcal{B}$ in Giraph++ satisfies P1-P3, one can prove its partition transparency along the same lines as Theorem 1 and Corollary 2. (b) Blogel [48] is block-centric, which adopts edge-cut to form blocks, *e.g.,* it groups URL links of the same host as a block in a Web graph. One can make Blogel to work with vertex-cut and hybrid, and use conditions P1-P3 to ensure its partition transparency.

(3) Condition P3 is an inherent property of the PIE model. In [20], we modeled a vertex-centric algorithm $\mathcal{B}$ as a special PIE program when each fragment consists of a single vertex. That is, $\mathcal{B}$ essentially runs under the PIE model and is assured to satisfy condition P3. Instead, we adopt the GAS mode of [36] for vertex-centric algorithms here and hence need P3 as an additional condition to ensure the correctness.

5 Partition-Transparent Algorithms

As proof of concept, in this section we provide PIE algorithms for SSSP, WCC and PR. We show that these programs are partition-transparent, *i.e.,* the same algorithms work correctly no matter what partitions are given. We also outline transparent algorithms for SCC and MaxClique.

5.1 Single Source Shortest Path

We start with the *single source shortest path* problem (SSSP).

Consider a directed weighted graph $G = (V, E, W)$, where each edge $e \in E$ carries a positive weight $W(e)$. The length of a path $P = (v_0, v_1, ..., v_k)$ in G is defined as $W(P) = \Sigma_{i\in[1,k]} W(v_{i-1}, v_i)$. Denote by $\mathsf{dist}(u, v)$ the length of the shortest path from u to v in G.

The single source shortest path problem is as follows.

- ◦ Input: A graph $G = (V, E, W)$, and a vertex $v_s \in V$.
- ◦ Output: Distance $\mathsf{dist}(v_s, v)$ for all $v \in V$.

(1) Algorithm. A PIE program for SSSP is outlined as follows. For each vertex v in fragment F_i, PEval declares a status variable $v.x = \mathsf{dist}(v_s, v)$, *i.e.,* its distance from source v_s, initialized as 0 if $v_s = v$ and $+\infty$ otherwise. The update region C_i is $F_i.O$. Messages are aggregated using min as f_{aggr}.

PEval is simply algorithm Dijkstra [19]. At the end of PEval, f_{aggr} takes the minimum $\mathsf{dist}(v_s, u)$ for each border node u of F_i, to reconcile $\mathsf{dist}(v_s, u)$'s from different fragments. IncEval is simply the incremental algorithm of [42]. It incrementally updates $\mathsf{dist}(\cdot)$ starting from border nodes.

The computation terminates when no more status variables can be updated. Now Assemble takes the union of status variable $v.x$ for each v in V as the final result.

(2) Transparency. For the partition transparency, PEval and IncEval are monotonic, since adding edges may only reduce shortest distances, not to make them longer. One can verify that the PIE program satisfies condition P2 since it is essentially the same algorithm developed in [21] for vertex-cut.

5.2 Weakly Connected Component

We now study *weakly connected component* (WCC).

Consider an undirected graph $G = (V, E)$, where each vertex $v \in V$ carries a unique vertex id, denoted by $v.\mathsf{id}$. We say that two vertices u and v are in the same *connected component* (WCC) if there exists a path between u and v. We define the id of a WCC to be minimum vertex id in it, and denote by $\mathsf{cid}(v)$ the id of the WCC in which v is. Then WCC is as follows.

- ◦ Input: An undirected graph $G = (V, E)$.
- ◦ Output: The $\mathsf{cid}(v)$ for each vertex $v \in V$.

(1) Algorithm. Our PIE program for WCC declares a status variable $\mathsf{cid}(v)$ for each vertex v, denoted by $v.x = \mathsf{cid}(v)$ and initialized as $v.\mathsf{id}$. For each fragment F_i, its update region is $F_i.O$. The aggregate function f_{aggr} is defined as min.

PEval identifies local WCCs via DFS (depth-first search). It finds the "root" v_c for each WCC such that $v_c.\mathsf{id}$ is the minimum within the WCC, and links v_c to all other vertices in the same WCC. At the end of PEval, f_{aggr} takes the minimum $\mathsf{cid}(u)$ of each border node u among all fragments.

Given the new $\mathsf{cid}(u)$ of a border node u, IncEval incrementally updates the local WCC that contains u, by updating $\mathsf{cid}(v)$ for all vertices v in the WCC and by using the links of the old root. The process proceeds until no $\mathsf{cid}(\cdot)$ changes. Assemble is then triggered to return $\mathsf{cid}(v)$ for all vertices v.

(2) Transparency. It is easy to verify that the program satisfies condition P2. Moreover, PEval and IncEval are monotonic since for each vertex v in G_1, if $G_1 \sqsubseteq G_2$ then the WCC C_1 containing v in G_1 has no more vertices than the WCC C_2 that contains v in G_2; as a consequence, the id of C_1 is no smaller than the id of C_2. Thus the program also satisfies condition P1, and it is partition-transparent by Theorem 1.

5.3 PageRank

Next we consider PageRank (PR) for ranking Web pages and links (see Section 2 for details). PR is defined as follow.

- ◦ Input: A directed graph $G = (V, E)$.
- ◦ Output: Ranking score $\mathsf{rank}(v)$ for each vertex $v \in V$.

(1) Algorithm. For each fragment F_i, the PIE program $\mathcal{A}$ declares its update region C_i as $F_i.O$; and for each vertex v, the status variable $v.x$ is its PR score $\mathsf{rank}(v)$, initialized as 0. The aggregate function f_{aggr} is defined as sum. To handle hybrid-cut partitions, for each edge e associated with border nodes, we maintain its global replication count, denoted as $||e||$, which is the total number of copies of e in all fragments.

At fragment F_i, PEval computes PR scores as follows: (a) For each master vertex v that has gathered all its incoming edges, it updates $\mathsf{rank}(v)$ using update function, and (b) for other vertices u, it computes partial ranking score:

$$\mathsf{rank}(v) = d \sum_{(u,v)\in E_i} \frac{\mathsf{rank}(u)}{d^+_G(v)||(u,v)||} + (1-d). \qquad (1)$$

To reduce the communication cost, if not all incoming edges of a vertex v are in place, we compute the partial ranking scores of v before sending messages. At the end of PEval, f_{aggr} aggregates partial scores of each border node with sum.

Upon receiving messages, IncEval first updates the score $\mathsf{rank}(u)$ of each border node u. It then iteratively updates ranking scores starting from border nodes, and propagates the updates through outgoing edges using update function.

The process proceeds until the sum of changes to scores $\mathsf{rank}(\cdot)$ is below a user-defined threshold ϵ. At this moment Assemble simply returns $\mathsf{rank}(v)$ for all vertices v.

(2) Transparency. We cannot apply P1-P3 to verify the partition transparency of algorithm $\mathcal{A}$, since $\mathcal{A}$ terminates when the change of the aggregate score is below a threshold, rather than when reaching a fix-point as other algorithms.

Instead, we can verify the partition transparency of the program as follows. Given any hybrid partition $\mathsf{HP}(G)$, we construct an edge-cut partition $\mathsf{HP}'(G)$ by letting all masters carry all their adjacent edges locally, and removing vertices and edges not linked to any master vertices. One can verify that (a) $\mathcal{A}(Q, \mathsf{HP}(G))$=$\mathcal{A}(Q, \mathsf{HP}'(G))$ and (b) $\mathcal{A}(Q, \mathsf{HP}'(G))$ correctly computes PR scores [51]. Hence the program correctly computes PR scores under hybrid partition $\mathsf{HP}(G)$.

Remarks. One can also develop other partition-transparent algorithms along the same lines as above.

SCC. Given a directed graph G, the strongly connected components problem, denoted by SCC, is to identify maximal subgraphs of G in which there exists a path between any pair of vertices. A PIE algorithm $\mathcal{A}$ can be extended from the graph-centric parallel algorithm in [24], and is shown to be partition-transparent using conditions P1-P2. Given G, the algorithm first picks a pivot v, and computes two sets $\mathsf{prec}(v)$ and $\mathsf{post}(v)$ consisting of vertices that have paths to and from v, respectively; it computes $\mathsf{prec}(v) \cap \mathsf{post}(v)$ to find an SCC. It iteratively conducts these steps until all SCCs are identified. $\mathcal{A}$ is correct under vertex-cut, *i.e.,* P2 holds, since $\mathsf{prec}(v)$ and $\mathsf{post}(v)$ can be correctly computed under vertex-cut partitions. Condition P1 can also be verified.

MaxClique. Given an undirected graph G, the maximum clique problem, denoted by MaxClique, is to find all maximum cliques in G. We adapt the vertex-centric algorithm in [12] for MaxClique, which identifies the maximum cliques by iteratively computing the intersection of neighbors of adjacent vertices. To make it partition-transparent, we add extra supersteps to synchronize both neighborhood information and messages among different copies of the same vertex just like the algorithm for CN in Example 5. In this way, the algorithm can correctly identify neighbors and compute intersections under vertex-cut, *i.e.,* it satisfies condition P2. One can verify that it also satisfies P1 and P3.

6 Experimental Study

Using real-life and synthetic graphs, we conducted three sets of experiments to evaluate partition-transparent algorithms for (1) effectiveness, (2) efficiency and (3) scalability.

Experimental setting. We start with the setting.

Datasets. We used six real-life graphs: (a) liveJournal [2], a social network with 4.8 million entities and 68 million relationships; (b) Twitter [5], a social network with 42 million users and 1.5 billion links; (c) UKWeb [6], a large Web graph with 106 million vertices and 3.7 billion edges; (d) DBpedia [1], a knowledge graph with 2.8 million entities and 33.4 million edges; (e) movieLens [3], a dense recommendation network with 25 million movie ratings between 162,000 users and 62,000 movies; and (f) traffic [4], a US road network with 23 million vertices and 58 million edges.

We also generated synthetic graphs with size up to 500 million vertices and 6 billion edges, to test scalability.

Partitions. To get a fair comparison when evaluating the effectiveness and efficiency of transparent graph algorithms, we partitioned the datasets with four different partitioners: (1) xtraPuLP [44], a state-of-the-art edge-cut partitioner; (2) NE [49], a state-of-the-art vertex-cut heuristic; and (3) ParE2H and ParV2H [20], two application-driven hybrid partitioners that refine edge-cut and vertex-cut to hybrid, respectively. For a fair comparison, in our experiments, ParE2H (resp. ParV2H) revises edge-cut (resp. vertex-cut) generated by xtraPuLP (resp. NE) *w.r.t.* the characteristics of the underlying applications (see more in [20]). We picked ParE2H (resp. ParV2H) since it revises conventional edge-cut (resp. vertex-cut) and hence demonstrates the improvement of graph computations under hybrid partitions.

Algorithms. We implemented a transparent version $\mathcal{A}$ for CN, PR, WCC, SSSP and TC (*i.e.,* triangle counting; see [20]) in the PIE model (see Section 5), and two other tailored versions, denoted as $\mathcal{B}_e$ and $\mathcal{B}_v$, where algorithm $\mathcal{B}_e$ (resp. $\mathcal{B}_v$) is designated for edge-cut (resp. vertex-cut). We also implement a transparent version $\mathcal{A}$ and two tailored versions, $\mathcal{B}_e$ and $\mathcal{B}_v$, for PR, WCC and SSSP in GAS. All algorithms are implemented on GRAPE [23], [7]. While GRAPE was developed for the PIE model, its open source version [7] supports GAS model. Note that since $\mathcal{A}$ is partition-transparent, it also works under both edge-cut and vertex-cut.

We will use the notations (i) $\mathcal{A}_e$ and $\mathcal{A}_v$ to denote that $\mathcal{A}$ runs on edge-cut and vertex-cut partitions, respectively; and (ii) $\mathcal{A}_h^e$ and $\mathcal{A}_h^v$ to denote that $\mathcal{A}$ runs on hybrid partitions generated by partitioners ParE2H and ParV2H, respectively.

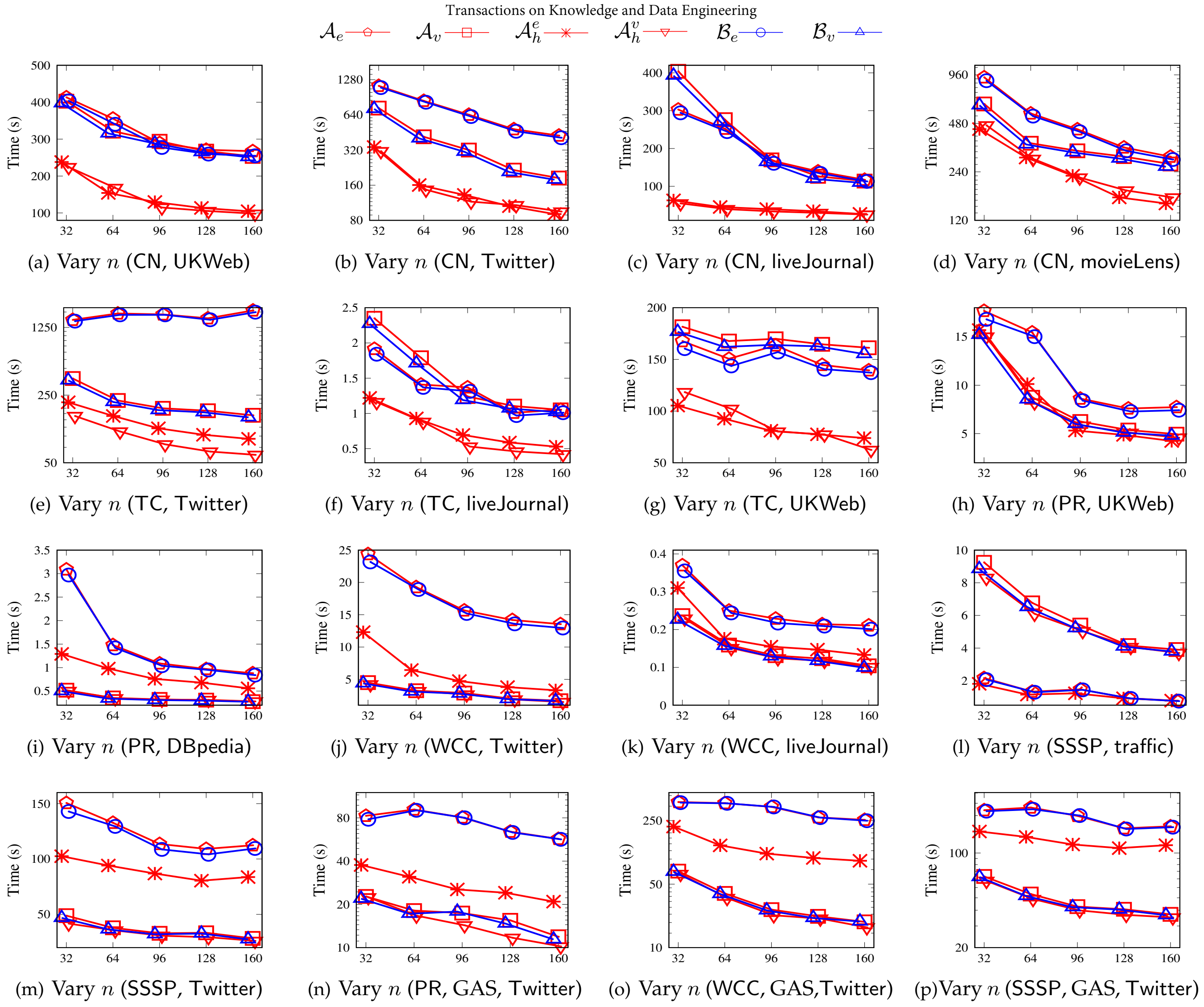
The experiments were conducted on open-source system GRAPE [23], [7] (see Section 2.1) deployed on 32 machines in an HPC cluster, each with 12 cores powered by Xeon 2.2GHz, 128GB RAM, and 10Gbps NIC. All experiments were repeated 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Effectiveness. We first tested the effectiveness of the partition-transparent algorithms. Varying the partition number n from 32 to 160, we tested transparent algorithms of CN, TC, PR, WCC, SSSP (section 5) and conventional algorithms for these problems tailored for edge-cut and vertex-cut partitions generated by xtraPuLP and NE.

(1) Under edge-cut, vertex-cut and hybrid partitions, transparent algorithms $\mathcal{A}$ return the same results as their counterparts $\mathcal{B}$ in both PIE model and GAS model. These verify the correctness of the transparent algorithms, *i.e.,* partition-transparent algorithms, either graph-centric or vertex-centric, work correctly no matter what partitions are given.

(2) Under hybrid partitions, transparent algorithms $\mathcal{A}$ (*i.e.,* $\mathcal{A}_h^e$ and $\mathcal{A}_h^v$) of these problems substantially outperform their counterparts $\mathcal{B}_e$ under edge-cut and $\mathcal{B}_v$ under vertex-cut (see detailed analysis in Exp-2), while $\mathcal{B}_e$ and $\mathcal{B}_v$ may not work correctly under hybrid partition. These justify the need for studying partition-transparent algorithms in order to benefit from the state-of-the-art graph partitioners.



(a) Vary n (CN, UKWeb) (b) Vary n (CN, Twitter) (c) Vary n (CN, liveJournal) (d) Vary n (CN, movieLens)

(e) Vary n (TC, Twitter) (f) Vary n (TC, liveJournal) (g) Vary n (TC, UKWeb) (h) Vary n (PR, UKWeb)

(i) Vary n (PR, DBpedia) (j) Vary n (WCC, Twitter) (k) Vary n (WCC, liveJournal) (l) Vary n (SSSP, traffic)

(m) Vary n (SSSP, Twitter) (n) Vary n (PR, GAS, Twitter) (o) Vary n (WCC, GAS,Twitter) (p)Vary n (SSSP, GAS, Twitter)

Fig. 5: Efficiency Evaluation

(3) When all algorithms run under edge-cut, the performance gap between transparent algorithms $\mathcal{A}_e$ and its tailored versions $\mathcal{B}_e$ is below 5.4% and 5.8% in PIE model and GAS model, respectively. This is because the transparent algorithms incur additional communication overhead to aggregate results during synchronization. The results over vertex-cut partitions are similar. The performance gap between transparent algorithms $\mathcal{A}_v$ and tailored versions $\mathcal{B}_v$ is smaller than 5.5% and 5.3% in PIE and GAS, respectively.

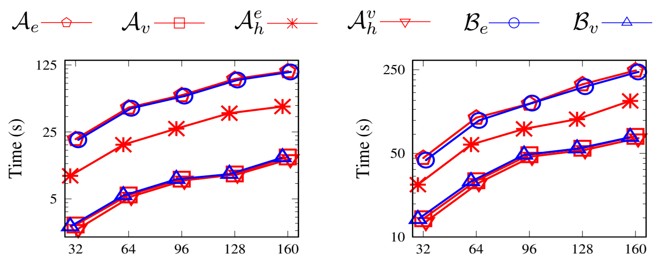
Exp-2: Efficiency. Varying n from 32 to 160, we evaluated the time taken by CN, TC, WCC, PR and SSSP under edge-cut, vertex-cut and their hybrid refinements, which are generated by xtraPuLP, NE, ParE2H and ParV2H, respectively. The results on PIE model and GAS model are shown in Figures 5(a)-5(m) and Figures 5(n)-5(p), respectively.

(1) CN in PIE model. Figures 5(a) to 5(d) report the performance of transparent $\mathcal{A}_h^e, \mathcal{A}_h^v$ under hybrid partitions, transparent $\mathcal{A}_e$, $\mathcal{A}_v$ and their counterparts $\mathcal{B}_e$ and $\mathcal{B}_v$ under edge-cut and vertex-cut of UKWeb, Twitter, liveJournal and movieLens, respectively. We find the following.

(a) The transparent algorithm $\mathcal{A}_h^e$ (resp. $\mathcal{A}_h^v$) under hybrid partitions beats its tailored counterparts $\mathcal{B}_e$ and $\mathcal{B}_v$ under edge-cut and vertex-cut by 3.2 and 2.6 (resp. 3.4 and 2.9) times on average, up to 5.4 and 6.3 (resp. 6.2 and 7.2) times, respectively. This is because the workload of $\mathcal{B}_e$ (resp. $\mathcal{B}_v$) for CN under edge-cut (resp. vertex-cut) is imbalanced, while the hybrid partitions balance workload for CN. Observe that $\mathcal{B}_e$ and $\mathcal{B}_v$ for CN cannot benefit from hybrid partitions since they do not work correctly under such partitions. These verify the benefit of transparent algorithms.

(b) The transparent algorithm ($\mathcal{A}_h^e$ and $\mathcal{A}_h^v$) under hybrid partitions also works better than under edge-cut and vertex-cut ($\mathcal{A}_e$ and $\mathcal{A}_v$), by 3.3 times and 2.9 times, respectively.

(2) TC in PIE model. As shown in Figures 5(e) to 5(g), on Twitter, liveJournal and UKWeb, the transparent algorithm $\mathcal{A}_h^e$ (resp. $\mathcal{A}_h^v$) for TC outperforms its tailored versions $\mathcal{B}_e$ and $\mathcal{B}_v$ by 6.0 and 1.8 (resp. 8.3 and 2.1) times on average, up to 29.7 times, respectively. This is because the hybrid partitions improve workload balance of TC as in the CN case. Note that the performance gap between transparent $\mathcal{A}_e$, $\mathcal{A}_v$ and their tailored counterparts $\mathcal{B}_e$ and $\mathcal{B}_c$ is quite small under edge-cut and vertex-cut partitions (see Exp-1). That is, transparent TC algorithm ($\mathcal{A}_h^e$ and $\mathcal{A}_h^v$) under hybrid partitions also converges in much less time than under edge-cut and vertex-cut partitions ($\mathcal{A}_e$ and $\mathcal{A}_v$).



(a) Varying $|G|$ (WCC) (b) Varying $|G|$ (PR)

Fig. 6: Scalability Evaluation

(3) PR in PIE model. Figures 5(h) and 5(i) report the performance of PR on UKWeb and DBpedia, respectively. As shown there, the transparent algorithm for PR performs the best under the hybrid partitions. With n=160, the transparent PR takes only 4.3s and 4.6s under hybrid partitions of UKWeb generated by ParE2H and ParV2H, respectively. On average, $\mathcal{A}_e^h$ under hybrid partition outperforms baselines $\mathcal{A}_e$ and $\mathcal{B}_e$ under edge-cut by 1.6 and 1.5 times, respectively. In contrast, $\mathcal{A}_v^h$ performs comparably to $\mathcal{A}_v$ and $\mathcal{B}_v$ since the vertex-cut of NE has very good edge balance.

(4) WCC in PIE model. Figures 5(j) and 5(k) report the performance of WCC over Twitter and liveJournal, respectively. Like PR, the transparent WCC under hybrid partitions performs the best. It outperforms the baselines by 1.7 times on average, up to 4.1 times. Unlike PR, the tailored algorithms $\mathcal{B}_e$ and $\mathcal{B}_c$ for WCC can work correctly under edge-cut, vertex-cut and even hybrid partitions, since they also satisfy the conditions P1 and P2 of partition transparency (Section 4). As a result, $\mathcal{B}_e$ and $\mathcal{B}_v$ perform as well as the transparent algorithm for WCC under the same partitions.

(5) SSSP in PIE model. Figures 5(l) and 5(m) show the performance of SSSP over dataset traffic and Twitter, respectively. As reported there, the transparent SSSP algorithm works the best under hybrid partitions among all the variants. However, the performance gap is smaller compared to the previous cases. On average, transparent SSSP under hybrid partitions outperforms the other variants by 17%. This is because (i) the workload of SSSP under vertex-cut by NE is already balanced, and not much can be improved via hybrid partitions; and (ii) like WCC, the tailored algorithms $\mathcal{B}_e$ and $\mathcal{B}_v$ are also already partition-transparent.

(6) PR, WCC and SSSP in GAS model. Figures 5(n) to 5(p) report the performance of PR, WCC and SSSP over Twitter in the GAS model. Similar to the PIE case, transparent GAS algorithms for the three are on average 2.0, 1.9 and 1.3 times faster than their counterparts, respectively. These verify that transparent algorithms work well not only in the graph-centric PIE model, but also in the vertex-centric GAS model.

The results are consistent on other datasets (not shown).

Exp-3: Scalability. Fixing $n = 96$, we varied the size of synthetic graphs $|G| = (|V|, |E|)$ from $(100M, 1.2B)$ to $(500M, 6B)$ to test the scalability of the transparent algorithms. The results for WCC and PR in the PIE model are reported in Figures 6(a) and 6(b), respectively.

(a) Transparent algorithms scale well. Transparent PR and WCC perform the best under hybrid partitions; they are on average 1.4 and 1.7 times faster than the others, respectively.

(b) Under edge-cut and vertex-cut partitions, transparent PR and WCC perform comparably to $\mathcal{B}_e$ and $\mathcal{B}_v$, respectively.

(c) Transparent algorithms work well on large graphs. On average transparent WCC and PR take 29.7s and 103.2s on G of 500 million nodes and 6 billion edges, respectively.

The results of the other transparent algorithms and their performance on other graphs are consistent (not shown).

Summary. We find the following. (1) The transparent algorithms in both graph-centric and vertex-centric models work correctly under edge-cut, vertex-cut and hybrid partitions, without the need for any changes. (2) Under hybrid partitions, the transparent algorithms $\mathcal{A}$ are on average 2.3 times faster than $\mathcal{A}_e$ and $\mathcal{A}_v$ under edge-cut or vertex-cut, up to 21.2 times, respectively. They are also 2.8 and 1.6 times faster than tailored algorithms $\mathcal{B}_e$ and $\mathcal{B}_v$ under edge-cut or vertex-cut, respectively. (3) Even when all algorithms run under vertex-cut or edge-cut, the performance gap between transparent algorithms and those tailored for individual partitions is small, *i.e.,* no more than 5.8%. (4) Under hybrid-partitions, transparent algorithms scale well. They are on average 2.5 times faster when the number n of processors varies from 32 to 160. They work well on large graphs, *e.g.,* transparent WCC and PR take on average 66.5s on graphs with $500M$ vertices and $6B$ edges using 90 processors.

7 Conclusion

We have proposed a notion of partition transparency, for graph algorithms to work correctly regardless of what partitions are given. We have identified conditions under which graph algorithms are guaranteed to be partition-transparent, under graph-centric and vertex-centric programming models. We have shown that partition-transparent algorithms are within the reach of a variety of graph computation problems. We have also experimentally verified that transparent algorithms are able to leverage application-driven hybrid partitions and speed up graph computations.

One topic for future work is to study systematic methods for developing partition-transparent algorithms. Another topic is to experiment transparent algorithms with the state-of-the-art hybrid partitioners upon their availability.

Acknowledgment

Fan is supported in part by ERC 652976 and Royal Society Wolfson Research Merit Award WRM/R1/180014. Lu is supported in part by NSFC 61602023 and SKLSDE-2021ZX-11. Liu is supported in part by EPSRC EP/L01503X/1.

References

[1] DBpedia. *http://wiki.dbpedia.org/Datasets.*
[2] Livejournal. *http://snap.stanford.edu/data/soc-LiveJournal1.html.*
[3] Movielens. *http://grouplens.org/datasets/movielens/.*
[4] Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml.*
[5] Twitter. *http://twitter.com/.*
[6] UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05,* 2006.
[7] Graphscope. *https://graphscope.io/,* 2020.
[8] K. Andreev and H. Racke. Balanced graph partitioning.*TCS,* 2006.

[9] D. Avdiukhin, S. Pupyrev, and G. Yaroslavtsev. Multi-dimensional balanced graph partitioning via projected gradient descent. *PVLDB*, 12(8):906–919, 2019.

[10] C.-E. Bichot and P. Siarry. *Graph partitioning*. John Wiley & Sons, 2013.

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

[12] A. Brighen, H. Slimani, A. Rezgui, and H. Kheddouci. Listing all maximal cliques in large graphs on vertex-centric model. *J. Supercomput.*, 75(8):4918–4946, 2019.

[13] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering - Selected Results and Surveys*, pages 117–158. 2016.

[14] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[15] R. Chen, J. Shi, B. Zang, and H. Guan. Bipartite-oriented distributed graph partitioning for big learning. In *APSys*, 2014.

[16] W. Cukierski, B. Hamner, and B. Yang. Graph-based features for supervised link prediction. In *INCC*, pages 1237–1244. IEEE, 2011.

[17] D. Dai, W. Zhang, and Y. Chen. IOGP: An incremental online graph partitioning algorithm for distributed graph databases. In *HPDC*, pages 219–230, 2017.

[18] R. Dathathri, G. Gill, L. Hoang, H. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *PLDI*, 2018.

[19] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[20] W. Fan, R. Jin, M. Liu, P. Lu, X. Luo, R. Xu, Q. Yin, W. Yu, and J. Zhou. Application driven graph partitioning. In *SIGMOD*, 2020.

[21] W. Fan, M. Liu, R. Xu, L. Hou, D. Li, and Z. Meng. Think sequential, run parallel. *LNCS*, 11180:24, 2018.

[22] W. Fan, P. Lu, W. Yu, J. Xu, Q. Yin, X. Luo, J. Zhou, and R. Jin. Adaptive asynchronous parallelization of graph algorithms. *TODS*, 45(2):6:1–6:45, 2020.

[23] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu. Parallelizing sequential graph computations. *TODS*, 43(4):18:1–18:39, 2018.

[24] L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *IPDPS Workshops*, 2000.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[26] N. Jain, G. Liao, and T. L. Willke. Graphbuilder: Scalable graph etl framework. *Graph Data Management Experiences and Systems*, 2013.

[27] G. Karypis. Metis and parmetis. In *Encyclopedia of Parallel Computing*, pages 1117–1124. 2011.

[28] G. Karypis and V. Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[29] G. Karypis and V. Kumar. Metis: A software package for partitioning unstructured graphs. *Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version*, 4, 1998.

[30] G. Karypis and V. Kumar. Multilevelk-way Partitioning Scheme for Irregular Graphs. *JPDC*, 48(1):96–129, 1998.

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

[32] R. Krauthgamer, J. Naor, and R. Schwartz. Partitioning graphs into balanced components. In *SODA*, 2009.

[33] D. Li, Y. Zhang, J. Wang, and K. Tan. TopoX: Topology refactorization for efficient graph partitioning and processing. *PVLDB*, 12(8):891–905, 2019.

[34] X. Li, M. Zhang, K. Chen, Y. Wu, X. Qian, and W. Zheng. 3-d partitioning for large-scale graph processing. *IEEE Trans. Computers*, 70(1):111–127, 2021.

[35] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. *CIKM*, 2003.

[36] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.

[37] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[38] D. W. Margo and M. I. Seltzer. A scalable distributed graph partitioner. *PVLDB*, 8(12):1478–1489, 2015.

[39] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, Oct. 2015.

[40] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. HDRF: Stream-based partitioning for power-law graphs. In *CIKM*, 2015.

[41] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIMAX*, 11(3):430–452, 1990.

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

[43] F. Sheng, Q. Cao, H. Jiang, and J. Yao. Grabi: Communication-efficient and workload-balanced partitioning for bipartite graphs. In *ICPP*, pages 25:1–25:11, 2020.

[44] G. M. Slota, S. Rajamanickam, and K. Madduri. PuLP/XtraPuLP: Partitioning tools for extreme-scale graphs. Technical report, Sandia National Lab (SNL-NM), Albuquerque, NM, US, 2017.

[45] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From ”think like a vertex” to ”think like a graph”. *PVLDB*, 2013.

[46] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL: Streaming graph partitioning for massive scale graphs. In *WSDM*, pages 333–342, 2014.

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

[48] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

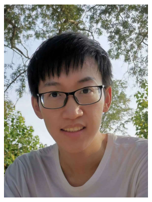
[49] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li. Graph edge partitioning via neighborhood heuristic. In *KDD*, 2017.

[50] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng. Exploring the hidden dimension in graph processing. In *OSDI*, 2016.

[51] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *TPDS*, 2013.

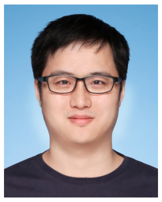
[52] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, 2016.

Wenfei Fan is the Chief Scientist in SICS and Chair Professor in the School of Informatics, University of Edinburgh. He is a fellow of the Royal Society, a foreign member of Chinese Academy of Sciences, a fellow of the Royal Society of Edinburgh, a member of Academia Europaea, and an ACM Fellow. He received his PhD from the University of Pennsylvania, and his MS and BS from Peking University. His research interests include database theory and systems.



Muyang Liu is a PhD student in the School of Informatics, University of Edinburgh, UK, working with Prof. Wenfei Fan. He received his BS degree in Electronic Engineering from Beihang University in 2018. His current research interests include big data analyses and parallel systems.

Ping Lu is an Associate Professor at the School of Computer Science and Engineering at Beihang University, China. He received his PhD from the University of Chinese Academy of Sciences in 2014. His research interests include big data, distributed computation and data quality.



Qiang Yin is an Assistant Professor at Shanghai Jiao Tong University. Before that, he worked as a senior engineer at Alibaba Damo Academy. He received his PhD from Shanghai Jiao Tong University in 2016. His research focuses on database theory and systems, especially in foundations and algorithms for big graph analytics.