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.

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 $G$, $\text{CN}$ is to compute $\text{CN}(u, v) = |\Gamma^+(u) \cap \Gamma^+(v)|$ for all pairs $(u, v) \in V \times V$.

Consider graph $G$ shown in Figure 1 and its partitions ($F^1_v, F^2_v$) and ($F^v_1, F^v_2$) 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 $F^2_v$ is a master vertex and $v_3$ in $F^v_1$ 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 $F^v_1$ and its edge set at $F^1_v$ does not include the edge $(v_3, v_2)$, which resides in fragment $F^2_1$ (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 $A_e$ for CN works as follows. (a) For each master vertex $v$ in a fragment $F_e^v$, it increases the local count $\text{CN}_k(v_i, v_j)$ for each incoming
neighbor pair \((v_i, v_j)\) of \(v\). (b) After this, \(A_e\) 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 \(A_e\) correctly returns count \(CN(v_3, v_2) = 2\) in the partition of Figure 1(a). Indeed, \(CN(v_3, v_2)\) is set to be 1 after step (a) in each fragment, and is then increased to 2 after the aggregation in step (b).

However, \(A_e\) erroneously returns \(CN(v_1, v_3)=1\) when it runs on the vertex-cut partition of Figure 1(b). That is the count in fragment \(F_2^v\) 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 \(F_1^v\).

(2) **Vertex-cut.** Under vertex-cut, an algorithm \(A_v\) 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 \(\Gamma_2^v(v)\) the set of incoming neighbors of \(v\) adjacent to received edges. Then, \(A_v\) increases \(CN(v_i, v_j)\) for each \(v_i \in \Gamma_2^v(v)\) and \(v_j \in \Gamma_2^v(v)\), where \(v_j\) is a vertex on a received edge. Note that \(A_v\) has all incoming edges \(\Gamma_2^v(v)\) 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, \(A_v\) first increases the \(CN(v_1, v_3)\) in fragment \(F_1^v\) to 1, and then increases it to 2 after receiving \((v_3, v_2)\).

In contrast, algorithm \(A_v\) does not work correctly when given the edge-cut partition of Figure 1(a). To see this, observe that \(A_v\) would return \(CN(v_1, v_3) = 4\), 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 \(F_1^e\) and \(F_2^e\); after collecting such edges from other fragments, \(A_v\) counts \(CN(v_1, v_3)\) twice for each duplicated edge.

The problem with algorithm \(A_v\) is that it assumes a master vertex to have all its edges in its local fragment, which is not true under vertex-cut. Algorithm \(A_e\) 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 \(A_e\) nor \(A_v\) works under the hybrid partition \((F_1^e, F_2^e, F_1^v, F_2^v)\) of Figure 1(c). More specifically, master vertex \(v_4\) in fragment \(F_2^e\) does not have all of its incident edges in \(F_2^v\), and edge \((v_3, v_2)\) replicates in \(F_1^h\) and \(F_2^h\). As a result, \(A_e\) finds \(CN(v_1, v_3)=1\), and \(A_v\) gets \(CN(v_1, v_3)=3\). None is correct. □

This example gives rise to several questions. Is it possible to make an algorithm \(A\) 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.
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 IOPG [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 $A$, it learns a cost model of $A$ beyond balance and replication; it generates a hybrid partition guided by the cost model, and speeds up parallel execution of $A$ 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.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G, V, E$</td>
<td>graph, vertex set and edge set of $G$</td>
</tr>
<tr>
<td>$F, O, T, O$</td>
<td>border nodes of fragment $F$ and graph $O$</td>
</tr>
<tr>
<td>$Q, Q, Q_{(G)}$</td>
<td>query class, a query, and query result</td>
</tr>
<tr>
<td>$\text{HP}(n)$</td>
<td>a $n$-way hybrid partition</td>
</tr>
<tr>
<td>$N_r$</td>
<td>partial results in round $r$ at worker $P_r$</td>
</tr>
<tr>
<td>$A(Q_r, \text{HP}(n))$</td>
<td>query result by algorithm $A$ over $\text{HP}(n)$</td>
</tr>
<tr>
<td>$\subseteq$</td>
<td>partial order on query results</td>
</tr>
<tr>
<td>$A_1, A_2, A_3, A_4$</td>
<td>transparent alg. $A$ under e-cut/v-cut/hybrid par</td>
</tr>
</tbody>
</table>

**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 $Q$ of queries, i.e., a graph computation problem. Given a query $Q \in Q$ 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 $Q$ under GRAPE, one only needs to specify three functions.

1. **PEval**: A sequential algorithm that given a query $Q \in Q$ 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 $\Delta G$ to $G$, computes updates $\Delta O$ 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 $\Delta G$ [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 $Q$ (PEval, IncEval and Assemble). PEval and IncEval can be any existing sequential (incremental) algorithms for $Q$.

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 $f_{\text{aggr}}$, e.g., $\text{min}$ and $\text{max}$, 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 \(Q \in Q\) 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 \(R_i^0 = \text{PEval}(Q,F_i)\) in fragment \(F_i\) at each worker \(P_i\) by invoking function \(\text{PEval}\), in parallel (\(i \in [1,n]\)). Here \(R_i^0\) 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 \(x_{agg} = f_{agg}(S_x)\) by applying the aggregate function \(f_{agg}\) declared in \(\text{PEval}\), to resolve conflicts. It generates message \(M_i\) to worker \(P_i\), which includes only those \(f_{agg}(S_x)\)'s such that \(f_{agg}(S_x) \neq x\), 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 \(\text{IncEval}\) to incrementally compute \(R_i^{r+1} = \text{IncEval}(Q,R_i^r,F_i,M_i)\) 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 \(\text{PEval}\). 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 \(\text{Assemble}\) 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 \(A(Q,G)\).

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 \(B\) iteratively executes the three phases at each vertex, in parallel at all vertices, until no more changes can be made [25]. More specifically, \(B\) updates the status \(D_v\) of vertex \(v\) in each iteration as follows.

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.

\[
\begin{align*}
    a'_v &= \text{gather}(D_v, D_u, D_{(v,u)}), \\
    a_v &= \text{sum}(a_u, a_v).
\end{align*}
\]

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 \(D_{newv}\), using the summarized values \(a_v\) generated in the gather phase. The apply function is defined as follows.

\[
D_{\text{new}} = \text{apply}(D_v, a_v).
\]

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

\[
D_{\text{new}}_{(v,u)} = \text{scatter}(D_{\text{new}}_v, D_u, D_{(v,u)}).
\]

The new edge status \(D_{\text{new}}_{(v,u)}\) 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 \(\text{sum}\); then the master copy of \(v\) runs \(\text{apply}\) function and sends updates back to all its mirrors; finally, function \(\text{scatter}\) 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 \(\text{rank}(v) = d \times \sum_{(u,v) \in E} \text{rank}(u) / d^+_v(u) + (1-d)\) for each \(v \in V\), where \(d\) is a damping factor in the range of \([0,1]\), and \(d^+_v(u)\) 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 \( D_v.\text{rank} \); 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 \( \epsilon \).

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 replicas in edge-cut partitions, e.g., \((v_2, v_3)\) resides in both \( F_1^h \) and \( F_2^h \) 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 \( HP(n) = (F_1, \ldots, F_n) \) of a graph \( G \) divides \( G \) into \( n \) small fragments \( F_1, \ldots, F_n \) such that (a) \( V_i = (V_i) \), (b) \( V = \bigcup_{i=1}^n V_i \), and (c) \( E = \bigcup_{i=1}^n E_i \). We refer to \( HP(n) \) 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 \( \bar{E}^v \) (resp. \( \bar{E}^e \)) the set of edges incident to vertex \( v \) in \( G \) (resp. \( F_i \)).

1. A vertex \( v \) is \( v \)-cut in \( HP(n) \) if the set of edges incident to \( v \) is not “complete” at any \( F_i \), i.e., \( \bar{E}^v \cap \bar{E}^{v_i} \neq \emptyset \ (\forall i \in [1, n]) \).
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., \( \bar{E}^v = \bar{E}^{v_i} \). If there are multiple copies of \( v \) in \( HP(n) \), we refer to its copy in \( F_i \) as an \( e \)-cut node and the others as its mirrors.
3. Denote by \( F_i.O = \{ v \in V_i \mid 3j(v \in V_i \land i \neq j) \} \) the set of border nodes of \( F_i \). Intuitively, a border node is replicated among fragments. We denote \( F.O = \bigcup_{i=1}^n F_i.O \). For each vertex \( v \in F.O \), 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 \((v_4, v_4)\) 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 \( v_3 \) 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 \( F.O = \{ v_2, v_3, v_4 \} \). □

One can easily verify the following.

1. Partition \( HP(n) \) 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 \( HP(n) \) 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., \( \{ v_1, v_2, v_3 \} \cap \{ v_3, v_4 \} = \emptyset \). Here the \( e \)-cut nodes are colored in gray, and the other vertices are mirrors.
2. Partition \((F_1^e, F_2^e)\) of Figure 1(b) is a vertex-cut partition of \( G \); since the edge sets of \( F_1^e \) and \( F_2^e \) 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.

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 \( CN(u, v) \) 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 \( CN(\cdot, \cdot) \) for the same common neighbor on replicated edges, IncEval only increases \( CN(\cdot, \cdot) \) for the “new edges” from other fragments, i.e., IncEval incrementally updates \( CN(\cdot, \cdot) \).

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 \( HP(n) = \)
(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, Γ^-(v), Γ^+(v)) to maintain both incoming neighbors Γ^-(v) and outgoing neighbors Γ^+(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 step, PEval computes 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 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 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 the 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_agg 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 computes the incoming neighbors of each vertex using the received message M_i. More specifically, (i) for each (v, ΔΓ^+_i(v), ΔΓ^-_i(v)) \in M_i, IncEval extends the accumulated neighbors ΔΓ^-_i(v) of v (line 3). (ii) For each outgoing neighbor u of v in ΔΓ^+_i(v), IncEval extends the incoming neighbors ΔΓ^-_i(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 CN(u_1, u_2).

(b) Updates. Then for each received vertex v, IncEval incrementally updates CN(\cdot, \cdot) using “new” incoming neighbors u_1 of v (lines 7-9). More specifically, for each vertex u_1 ∈ ΔΓ^-_i(v) \ ΔΓ^-_i(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 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 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 A = (PEval, IncEval, Assemble) for a class Q of queries. To specify the conditions we use the following notations. (a) Partial results R^\_i consist of a set of status variables (see Section 2.1 for R^\_i). We assume the existence of a partial order \preceq on R^\_i; for each status variable y in fragment F_i, let y ≤ x be a partial order on the domain of y; then we say that R^\_i ≤ R^\_i' if y.val ≤ x, y.val' for each status variable y in fragment F_i, where y.val and y.val' denote the values of y in partial results R^\_i and R^\_i' in F_i, respectively.

(b) Denote by G_1 \subseteq G_2 if graph G_1 is a subgraph of G_2.

We say that an algorithm A is monotonic if for all queries Q ∈ Q and graphs G_1 and G_2, if G_1 \subseteq G_2 then A(Q, G_1) ≤ 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 A, we require both PEval and IncEval to be monotonic [22]. The function PEval (resp. IncEval) is monotonic if for all queries Q ∈ Q and graphs G_1 and G_2, if G_1 \subseteq G_2 (resp. R^\_i ≤ R^\_i') then PEval(Q, G_1) ≤ PEval(Q, G_2) (resp. R^\_i+1 ≤ R^\_i+1). Here R^\_i and R^\_i' denote partial results in (possibly different) runs of the PIE program A.

We say that an algorithm A is correct under vertex-cut if for all queries Q ∈ Q, all graphs G and any vertex-cut partition HP(n) of graph G, A(Q, HP(n)) = Q(G).

Example 6: We show that the PIE program A for CN given in Section 3 is monotonic and is correct under vertex-cut.

(1) To see that algorithm A is monotonic, we define partial order ≤ as follows. Denote by R^\_i (resp. R^\_i') the common neighbor counts for all vertex pairs of G computed in fragment F_1 (resp. F_2). We say that R^\_i ≤ R^\_i' if CN_\_i(u, v) ≤ CN_\_i(u, v) for each pair (u, v) in G. Here CN_\_i(u, v) and
Algorithm

1. Algorithm \( A \) is partition-transparent if \( A \) satisfies conditions P1 and P2.

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

Let \( HP(n) = (F_1, \ldots, F_m) \). We next construct partitions \( HP^a(n) \) and \( HP^b(n) \) and verify their properties one by one.

1. We start with the construction of \( HP^a(n) \) and show that \( Q(G) = A(Q, HP^a(n)) \). Note that in the hybrid partition \( HP(n) \) there may exist duplicated edges among fragments. Let \( HP^a(n) = (F_1', \ldots, F_m') \) be a vertex-cut partition of \( G \) obtained by removing these duplicated edges from \( HP(n) \). Since \( A \) is correct under vertex-cut (condition P2), we have that \( Q(G) = A(Q, HP^a(n)) \).

It remains to show that \( A(Q, HP^b(n)) \subseteq A(Q, HP(n)) \). Observe that by the construction, we have that \( F_i' \subseteq F_i \) for \( i \in [1, n] \), since we only remove edges from \( HP(n) \), where \( F_i' \) is the fragment in \( HP(n) \) that corresponds to fragment \( F_i \) in \( HP(n) \). By the monotonicity of \( A \) (condition P1), we have the following: (a) \( R_0' = PEval(Q, F_i') \leq PEval(Q, F_i) = R_0 \), and (b) \( R_{i+1}' \leq R_{i+1} \) for \( i \in [1, n] \) and \( t \geq 0 \). From these it follows that \( Q(G) = A(Q, HP(n)) \). \( \square \)

**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. \( \square \)

**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.

3. Algorithm \( A \) is correct when graphs are replicated.

Here algorithm \( A \) is correct when graphs are replicated if for all queries \( Q \in Q \) and all graphs \( G, A(Q, HP^a(n)) = Q(G) \), where \( HP^a(n) = (G, \ldots, G) \) that duplicates \( G \).

**Corollary 2:** A vertex-centric algorithm \( B \) of the GAS model is partition-transparent if \( B \) satisfies conditions P1, P2 and P3.

**Proof:** Similar to the proof of Theorem 1, we build partitions \( HP^a(n) \) and \( HP^b(n) \) to bound \( B(Q, HP(n)) \).

1. \( HP^a(n) = (F_1', \ldots, F_m') \) is a vertex-cut by removing duplicated edges from \( HP(n) \). It follows from conditions P1 and P2 that \( Q(G) = B(Q, HP(n)) \). By P3, \( B(Q, HP^a(n)) = Q(G) \), implying condition P1, we have that \( B(Q, HP(n)) \). \( \square \)

**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. \( \square \)

**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-parallel model, which can run sequential algorithms in the first superstep, and then vertex-parallel algorithms in the following steps. If an algorithm 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 B as a special PIE program when each fragment consists of a single vertex. That is, B essentially runs under the PIE model and is assured to satisfy condition P3. Instead, we adopt the GAS mode of [36] for vertex-parallel 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 $dist(u, v)$ 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 $dist(v_s, v)$ for all $v \in V$.

**Algorithm.** A PIE program for SSSP is outlined as follows. For each vertex $v$ in fragment $F_i$, PEval declares a status variable $v.x = dist(v_s, v)$, 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 $aggr$.

PEval is simply algorithm Dijkstra [19]. At the end of PEval, $f_{agg}$ is simply the incremental algorithm of [42].

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 $v.id$. 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 $cid(v)$ the id of the WCC in which $v$ is. Then WCC is as follows.

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

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

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

Given the new $cid(u)$ of a border node $u$. IncEval incrementally updates the local WCC that contains $u$, by updating $cid(v)$ for all vertices $v$ in the WCC and by using the links of the old root. The process proceeds until no $cid(\cdot)$ changes. Assemble is then triggered to return $cid(v)$ 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 $G_1 \subseteq G_2$ then the WCC $C_1(v)$ containing $v$ in $G_1$ has no more vertices than the WCC $C_2(v)$ 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 follows:

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

**Algorithm.** For each fragment $F_i$, the PIE program $A$ declares its update region $C_i$ as $F_i O$; and for each vertex $v$, the status variable $v.x$ is its PR score $rank(v)$, initialized as 0. The aggregate function $f_{agg}$ 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 $rank(v)$ using update function, and (b) for other vertices $u$, it computes partial ranking score:

$$rank(v) = d \sum_{(u,v) \in E} \frac{rank(u)}{d_G(v) ||(u,v)||} + (1 - d). \quad (1)$$

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, $f_{agg}$ aggregates partial scores of each border node with sum.

Upon receiving messages, IncEval first updates the score $rank(u)$ 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 \( \text{rank}(\cdot) \) is below a user-defined threshold \( \epsilon \). At this moment Assemble simply returns \( \text{rank}(v) \) for all vertices \( v \).

(2) **Transparency.** We cannot apply P1-P3 to verify the partition transparency of algorithm \( A \), since \( A \) 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 \( HP(G) \), we construct an edge-cut partition \( HP'(G) \) 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) \( A(Q, HP(G)) = A(Q, HP'(G)) \) and (b) \( A(Q, HP'(G)) \) correctly computes PR scores [51]. Hence the program correctly computes PR scores under hybrid partition \( HP(G) \).

**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 \( A \) 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 \( \text{prec}(v) \) and \( \text{post}(v) \) consisting of vertices that have paths to and from \( v \), respectively; it computes \( \text{prec}(v) \cap \text{post}(v) \) to find an SCC. It iteratively conducts these steps until all SCCs are identified. \( A \) is correct under vertex-cut, i.e., \( P2 \) holds, since \( \text{prec}(v) \) and \( \text{post}(v) \) 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 \( A \) 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 \( B_e \) and \( B_v \), where algorithm \( B_e \) (resp. \( B_v \)) is designated for edge-cut (resp. vertex-cut). We also implement a transparent version \( A \) and two tailored versions, \( B_e \) and \( B_v \), 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 \( A \) is partition-transparent, it also works under both edge-cut and vertex-cut.

We will use the notations (i) \( A_e \) and \( A_v \) to denote that \( A \) runs on edge-cut and vertex-cut partitions, respectively; and (ii) \( A_e^h \) and \( A_v^h \) to denote that \( A \) 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 \( A \) return the same results as their counterparts \( B \) 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 \( A \) (i.e., \( A_e^h \) and \( A_v^h \)) of these problems substantially outperform their counterparts \( B_e \) under edge-cut and \( B_v \) under vertex-cut (see detailed analysis in Exp-2), while \( B_e \) and \( B_v \) 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.
(3) When all algorithms run under edge-cut, the performance gap between transparent algorithms $A_e$ and its tailored versions $B_e$ 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 $A_h$ and tailored versions $B_v$ 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 $A_h^T$, $A_v^T$ under hybrid partitions, transparent $A_e$, $A_v$ and their counterparts $B_e$ and $B_v$ under edge-cut and vertex-cut of UKWeb, Twitter, liveJournal and movieLens, respectively. We find the following.

(a) The transparent algorithm $A_h^T$ (resp. $A_v^T$) under hybrid partitions beats its tailored counterparts $B_e$ and $B_v$ 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 $B_e$ (resp. $B_v$) for CN under edge-cut (resp. vertex-cut) is imbalanced, while the hybrid partitions balance workload for CN. Observe that $B_e$ and $B_v$ 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 $A_h^T$ and $A_v^T$ under hybrid partitions also works better than under edge-cut and vertex-cut ($A_e$ and $A_v$), 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 $A_h^T$ (resp. $A_v^T$) for TC outperforms its tailored versions $B_e$ and $B_v$ 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 $A_e$, $A_v$ and their tailored counterparts $B_e$ and $B_v$ is quite small under edge-cut and vertex-cut partitions (see Exp-1). That is, transparent TC algorithm ($A_h^T$ and $A_v^T$) under hybrid partitions also converges in much less time than under edge-cut and vertex-cut partitions ($A_e$ and $A_v$).
(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, \( A^e \) under hybrid partition outperforms baselines \( A \) and \( B \) under edge-cut by 1.6 and 1.5 times, respectively. In contrast, \( A^h \) performs comparably to \( A \) and \( B \) 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 \( B_e \) and \( B_c \) 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, \( B_e \) and \( B_c \) 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 \( B_e \) and \( B_c \) 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 \( B_e \) and \( B_c \), 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 \( A \) are on average 2.3 times faster than \( A_e \) and \( A_v \) 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 \( B_e \) and \( B_v \) 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**
