2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

A Partition-centric Distributed Algorithm for Identifying Euler Circuits in Large Graphs

Siddharth D Jaiswal and Yogesh Simmhan
Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India
Email: {siddharthj,simmhan}@IISc.ac.in

***Abstract*—Finding the Eulerian circuit in graphs is a classic problem, but inadequately explored for parallel computation. With such cycles finding use in neuroscience and Internet of Things for large graphs, designing a distributed algorithm for finding the Euler circuit is important. Existing parallel algorithms are impractical for commodity clusters and Clouds. We propose a novel partition-centric algorithm to find the Euler circuit, over large graphs partitioned across distributed machines and executed iteratively using a Bulk Synchronous Parallel (BSP) model. The algorithm finds partial paths and cycles within each partition, and refines these into longer paths by recursively merging the partitions. We describe the algorithm, analyze its complexity, and validate it on Apache Spark for large graphs. Our experiments show that memory pressure still limits weak and strong scaling, and we propose an enhanced design to address it.**

I. INTRODUCTION

Finding an *Euler circuit* through a graph is a classic problem in graph theory [1], and requires that starting at any vertex, we traverse every edge of the graph that is part of the connected component exactly once, and return back to the starting vertex. Finding such circuits is practically useful in transportation and logistics for route planning [2] and for efficient circuit design [3]. There are sequential algorithms to find the Euler Circuit in linear time of the edge count, $\mathcal{O}(|E|)$ [4]. As a result, there has not been substantial work on parallel algorithms [5], [6], [7], [8]. However, this is changing with the use of Euler circuits in large graphs, from biology for DNA fragment assembly and rendering [9], [10], and for routing of autonomous vehicles [11]. There has also been a proliferation of scalable Big Data platforms and abstractions for graph processing [12], [13]. This motivates the need for a parallel Euler circuit algorithm that can make effective use of commodity platforms.

In this paper, we propose a novel partition-centric distributed algorithm to find an Euler Circuit [14], [15]. We assume that the graph is Eulerian, i.e., each vertex has an even degree [4], and it has been pre-partitioned into connected components placed on different machines by a partitioner. Our algorithm then concurrently finds pairwise edge-disjoint partial paths locally in each partition, and then recursively merges the partial paths in pairs of partitions to coarsen it, and eventually constructs the final circuit. Intuitively, it attempts to reduce the memory and communication costs – finding local paths reduces the memory for a single partition on a machine by replacing many edges with a single path, while merging partition-pairs onto a single machine is a form of coarse-grained communication, while allowing paths to be linked together. To our knowledge, this is the first partition-centric distributed algorithm for finding an Euler Circuit.

We make the following specific contributions in this paper:

1) We design a partition-centric algorithm to find the Euler circuit for large graphs on commodity clusters. We describe the algorithm and its complexity measures, in Sec. III.
2) We implement the algorithm on Apache Spark [16], and present experimental results and analysis for large synthetic graphs, in Sec. IV.
3) We identify memory bottlenecks in our design and propose improvements, supported by an analytical study, in Sec. V.

Also, we offer background and review related literature in Sec. II, and present our conclusions and future work in Sec. VI.

II. BACKGROUND AND RELATED WORK

A. Distributed Graph Processing Platforms

Many graph-processing libraries and frameworks such as *Parallel BGL* [17], *CGMgraph* [18] and *Gunrock* [19] exist for shared-memory, High Performance Computing (HPC) and GPU clusters. However, such platforms are not well-suited for commodity clusters and Clouds that we target, which have networks with higher latency. Rather, *component-centric models of iterative computing* based on Google's Pregel [20] have grown popular for distributed systems. Graph components of different granularities – vertex, subgraph, partition – are the unit of data-parallel iterative computation on the partitioned and distributed graph [14], [15]. User logic is specified from the perspective of a single component, which can then update the local state, and pass messages that are delivered in bulk to other components at global barriers. This iterates as as a set of *supersteps* using a Bulk Synchronous Parallel (BSP) model. Using coarser units of parallelism, such as *partitions* [15] and *connected-components* [14], can make more progress in a single superstep by processing all entities within the component to local convergence, and reduce the messages and supersteps required for completion. We use a *partition-centric computing model* in our algorithm design. While *GraphX* [13] is a a vertex-centric platform based on *Apache Spark* [16], we extend Spark to implement our partition-centric algorithm.

B. Parallel Eulerian Circuit Algorithms

Hierholzer's algorithm [4] to to find an Euler circuit, with a time complexity of $\mathcal{O}(|E|)$, is the most common sequential

978-1-5386-5555-9/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPSW.2019.00085

IEEE computer society

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 18,2022 at 06:55:45 UTC from IEEE Xplore. Restrictions apply.

one. It selects an initial source vertex in the graph and traverses out along an unvisited edge, repeatedly until it returns to this source and there are no unvisited edges out from it. It runs again from another source vertex with unvisited edges, and this repeats until all edges are visited. Each run identifies an *edge-disjoint cycle*, or a circuit, in the graph, and they each intersect with at least one other circuit at one or more vertices. *Merging* the cycles at these vertices gives a full Euler circuit.

PRAM algorithms to find the Euler circuit have been examined in detail [5], [6], and run in $\mathcal{O}(\log |V|)$ time with $|V| + |E|$ processors having access to the whole graph. It first transforms the original graph to an auxiliary bipartite graph, constructs a spanning tree through it, identifies an Euler digraph, followed by an Euler circuit, which is then mapped back to the original graph. However, the theoretical speedup of PRAM algorithms do not match reality on contemporary distributed computing clusters [21].

Makki [7] employs a vertex-centric adaptation of Hierholzer's algorithm and traverses the next unvisited edge from the current vertex. It backtracks when visiting a vertex which has only one unvisited edge, and constructs a single walk. However, executing it in a distributed vertex or partition centric model takes $\mathcal{O}(|E|)$ or $\mathcal{O}(\text{edge cuts})$ supersteps, which is a high *coordination cost*. Further, all but one machine hosting the partition with the active vertex are idle at a time, causing *poor resource utilization*. We address both these limitations.

Caceres, et al. [22] identify the Euler circuit on a *tree* using the BSP model. On a system with p processors, it requires $\mathcal{O}(\frac{|V|}{p})$ memory per processor, and takes $\mathcal{O}(\log p)$ supersteps and $\mathcal{O}(\frac{|V|}{p})$ computations per superstep. *Yan, et al.* [8] take a similar vertex-centric approach to find the Euler circuit for a tree. It satisfies their Balanced Practical Pregel Algorithm (BPPA) constraints, and takes 2 supersteps to complete. However, neither generalize to a graph. Lastly, *external-memory algorithms* to idenitfy Euler circuits on trees [23] have been proposed, but these have significant I/O bottlenecks.

In summary, current literature on finding Euler circuits are sequential algorithms, parallel PRAM algorithms limited to theoretical use, have high complexity for large graphs, or work only on trees and not graphs. We address this gap with a distributed algorithm that uses a partition-centric approach, and offers better complexity metrics and is suitable for execution on commodity clusters.

III. Partition-centric Euler Circuit

The starting point for our distributed algorithm is an *undirected graph* that has been partitioned into p parts, each present on a different machine. The expectation is that the partitioner has reduced edge cuts between the partitions, and load-balanced the number of vertices in the partitions, using tools like ParHIP [24]. The impact of partitioning strategies on component algorithms has been studied earlier [25], and as we see in the results, the number of edge cuts impacts our algorithm. Each partition tends to contain one or more large connected components. The graph must be *Eulerian*, with all vertices having an even number of (local or remote) edges.

Such an undirected graph, partitioned across P_1–P_4, is illustrated in Fig. 1a. *Internal vertices* in each partition (white, e.g., v_4, v_5 in P_2) are connected exclusively through *local edges* (red solid lines, e.g., $e_{3,4}, e_{4,5}$) to other vertices in the same partition. *Boundary vertices* (yellow, e.g., v_3 in P_2) have at least one *remote edge* (blue dashed lines) to vertices in other partitions (e.g., between v_3 in P_2 and v_{13} in P_4). All vertices are given unique labels, and the edges are uniquely labeled by the pair of vertex IDs that they are incident upon (e.g., $e_{3,13}$ between v_3 and v_{13}). As the graph is undirected, $e_{i,j}$ and $e_{j,i}$ are equivalent. One *Euler circuit* in this graph is:

$$(e_{1,2})_{P_1} \rightarrow e_{2,3} \rightarrow (e_{3,4} \rightarrow e_{4,5} \rightarrow e_{3,5})_{P_2} \rightarrow e_{3,13} \rightarrow (e_{12,13} \rightarrow e_{11,12})_{P_4} \rightarrow e_{6,11} \rightarrow (e_{6,7} \rightarrow e_{7,8} \rightarrow e_{8,9})_{P_3} \rightarrow e_{9,10} \rightarrow (e_{10,12} \rightarrow e_{12,14})_{P_4} \rightarrow e_{1,14}$$

For convenience, we have used parenthesis to group local edges within a partition, with the partition ID in the subscript. The blue edges not within parenthesis are remote edges.

In the rest of this section, we formally define the partitioned-graph and the Euler circuit problem, followed by our proposed algorithm, its formal properties, and a complexity analysis.

A. Preliminaries

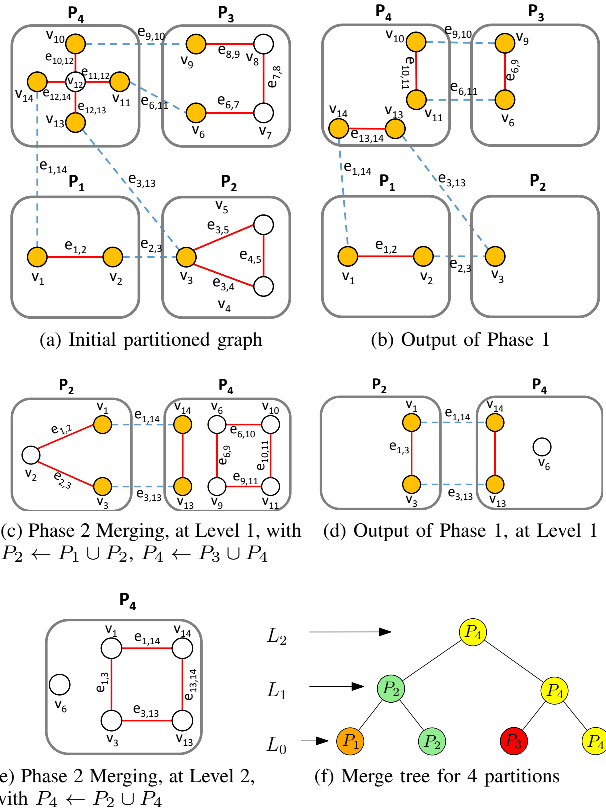
An *undirected graph* is defined as $G = \langle V, E \rangle$, where $v_i \in V$ is the set of vertices and $e_{i,j} \in E \subseteq V \times V$ is the set of edges, where $e_{i,j}$ connects from vertex v_i to v_j. We represent an undirected edge with a pair of directed edges, i.e., $e_{i,j} \in E \Leftrightarrow e_{j,i} \in E$. The *Euler Circuit* on this graph is a *path* $\pi = [e_{i,j}, e_{j,k}, ..., e_{p,q}, e_{q,i}] \mid \forall e_{i,j} \in E \; \exists \; e_{i,j} \in \pi$, *exactly once*. The path starts *and* ends at the same vertex $v_i \in V$, the sink vertex of an edge is the same as the source vertex of its successor edge, and every edge in E appears exactly once in the path.

Let $\delta(v_i)$ be the *undirected edge degree* of vertex v_i. It has been proved that a graph is *Eulerian*, i.e., has an Euler circuit, if and only if $\forall v_i \in V, \; \delta(v_i) \mod 2 = 0$, i.e., all vertices have an even degree [1].

A graph G partitioned into n parts is given as $\mathbb{G} = \{P_1, ..., P_n\}$, where $P_i = \langle I_i, B_i, L_i, R_i \rangle \mid I_i, B_i \subseteq V, \; L_i, R_i \subseteq E$. Here, I and B are the set of *internal* and *boundary vertices* for the partition, while L and R are its set of *local* and *remote edges*. Local edges connect vertices within a partition while remote edges connect vertices in different partitions: $l_{p,q} \in L_i \implies v_p, v_q \in (I_i \cup B_i)$ and $r_{p,q} \in R_i \implies v_p \in B_i, \; v_q \in \bigcup_{j \in n \setminus i} B_j$. Each vertex in the original graph appears exactly in one of the partitions, and similarly each local edge. Formally, for vertices: $\bigcup_{i \in n}(I_i \cup B_i) = V$ and $I_i \cap B_i = \varnothing$, $\forall i, j \in n, \; i \neq j, \; (I_i \cup B_i) \cap (I_j \cup B_j) = \varnothing$, and likewise for edges. WLOG we treat a partition as a *connected component* to simplify the algorithm design, but it can be generalized to partitions or even graphs with multiple components.

We define the *meta-graph* formed from the partitioned graph as $\overline{G} = \langle \overline{V}, \overline{E} \rangle$, where the *meta-vertices* $\overline{V} = \{P_1, ..., P_n\}$ are the set of partitions, and the *meta-edges* $\overline{E} \subseteq \overline{V} \times \overline{V} = \{m_{i,j} \mid \exists \; e_{k,l} \in E, v_k \in B_i, \; v_l \in B_j\}$ denote the existence of at least

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 18,2022 at 06:55:45 UTC from IEEE Xplore. Restrictions apply.



(a) Initial partitioned graph
(b) Output of Phase 1
(c) Phase 2 Merging, at Level 1, with $P_2 \leftarrow P_1 \cup P_2$, $P_4 \leftarrow P_3 \cup P_4$
(d) Output of Phase 1, at Level 1
(e) Phase 2 Merging, at Level 2, with $P_4 \leftarrow P_2 \cup P_4$
(f) Merge tree for 4 partitions

Figure 1: Incremental phases of finding a partition-centric Euler circuit on a sample graph.

one edge between their boundary vertices. The *weight of a meta-edge* $m_{i,j}$ is given as $\omega(m_{i,j})$, the count of the edges between all the boundary vertices of those two partitions.

Further, we classify the boundary vertices for a partition into one of two types: *odd degree boundary vertices (OB)* and *even degree boundary vertex (EB)* [1], such that $OB_i \cup EB_i = B_i$ and $OB_i \cap EB_i = \varnothing$. We further have,

$$o_i \in OB \subseteq B \implies \delta_L(o_i) \mod 2 = 1$$
$$u_i \in EB \subseteq B \implies \delta_L(u_i) \mod 2 = 0$$

where δ_L and δ_R give the *local edge degree* and *remote edge degree* for that boundary vertex, respectively. Since we have an Eulerian graph, $\delta_L(v) + \delta_R(v) \mod 2 = 0$. This implies that these odd vertices have an odd numbered remote edge degree as well, and even vertices have an even numbered remote edge degree. In Figure 1a, the boundary vertices are in yellow, of which v_3 is an EB with two remote edges, while the rest are OBs. Also, each partition will have an even number of odd vertices, as can be proved using the *Handshaking Lemma* [1].

B. Approach

We first present the high-level idea behind the partition-centric Eulerian circuit, which is calculated in 3 phases.

[1]For brevity, we contract these terms and alternatively refer to them as odd vertices (or OB) and even vertices (or EB).

Phase 1. Within each partition, we identify *edge-disjoint maximal local paths* that start and end at an OB through local edges. We then find *edge-disjoint maximal local cycles* using the remaining local edges, which start and end at EBs or internal vertices. This may include trivial tours that start and end at the same even degree boundary vertex, with no edges. This phase is done concurrently on all partitions. The following assertion holds (see [26] for details):

Assertion 1. *It is necessary to identify maximal local paths starting and ending at odd degree boundary vertices, and only then the cycles starting and returning at even degree boundary vertices and at internal vertices, within a partition.*

E.g., in Fig. 1a, if we first initiate a traversal at the internal vertex v_8 of P_3, it will terminate at one of the two odd vertices, v_9 or v_6, forming a path and not a cycle.

After this, all local edges L are part of exactly one path or cycle, and all internal vertices I and boundary vertices B are part of one or more paths or cycles. Each path effectively forms a new *coarser* edge (which we call an *OB-pair*) that encodes the edges in the path. Likewise, the source/sink boundary or internal vertex captures the edges in the cycle. The cycles on internal vertices are appended to some OB path or EB cycle. After Phase 1, the paths and cycles consume and replace all the local edges and internal vertices that are part of them, and *persist them to disk*. This allows the sets L and I to be removed to conserve memory, and only retain the OB-pairs, the boundary vertices B, and the remote edges R in memory.

E.g., In Fig. 1a, the local path between v_6 and v_9 in P_3 is $e_{6,7} \rightarrow e_{7,8} \rightarrow e_{8,9}$, and this is replaced by the OB-pair edge $e_{6,9}$ in P_4, as shown in Fig. 1b. Similarly, the local cycle starting at v_3 in P_2, $e_{3,4} \rightarrow e_{4,5} \rightarrow e_{3,5}$, is consumed and only the boundary vertex v_3 is retained in P_2.

Phase 2. After Phase 1 completes, we identify pairs of partitions that will be *merged* together. This is analogous to coarsening strategies prevalent in graph processing. This causes the remote edges between a partition-pair to become local edges of the merged partition, and the corresponding boundary vertices for those edges to become internal vertices (if they have no remote edges to any other partition), or remain boundary vertices. Then, we *rerun Phase 1* on each merged partition in parallel to identify additional local paths and cycles that consume these new local edges and internal vertices. This forms level 0 of the *merge tree*, and is repeated recursively (Fig. 1f). We use a greedy strategy that prioritizes partition pairs with *high meta-edge weight* ω between them. At the end, only one merged partition remains.

E.g., Fig. 1c shows the first level of Phase 2 merging performed on the output of Phase 1 in Fig. 1b, with P_1 and P_2 merged into P_2, and P_3 and P_4 merged into P_4. The remote edge $e_{2,3}$ between P_1 and P_2 has become a local edge. On running Phase 1 on the merged P_2, it would identify $e_{1,2} \rightarrow e_{2,3}$ as a local path between the odd vertices, v_1 and v_3, which is replaced by the OB-pair, $e_{1,3}$, as seen in Fig. 1d.

As we can see, Phase 1 reduces the memory usage by the local edges and internal vertices, while Phase 2 increases the memory usage by merging two partitions into one, with the

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 18,2022 at 06:55:45 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Phase 1: Identifying local paths and cycles

```
1: procedure doPhase1(Partition P_i)
2:     for v ∈ B_i do                      ▷ Init vertices and edges
3:         v.visited = false
4:     for e ∈ L_i do
5:         e.visited = false
6:     pathMap[ ] = ∅
7:     while ∃v.type == OB && v.visited == false do
8:         pathMap[ ].add(findEulerPath(v))
9:     for ∀v.type == EB do
10:        pathMap[ ].add(findEulerPath(v))
11:    while ∃e.visited == false do         ▷ e.src ∈ I_i
12:        ivCycle = findEulerPath(e.src)
13:        mergeInto(pathMap, ivCycle)
       return pathMap
```

Algorithm 2 Phase 2: Creating merge tree from meta-graph

```
1: procedure generateMergeTree(Ḡ)
2:     mergeTree = ∅
3:     l = 0            ▷ Current level
4:     Ḡ_l ← Ḡ             ▷ Initialize meta-graph at level 0
5:     while |V̄_l| > 1 do
6:         edges = maximalMatching(Ē_l)
7:         mergeTree.add(l, edges)   ▷ Add partitions incident
   on each edge as siblings in the tree at level l
8:         l ← l + 1          ▷ Increment level
9:         Ḡ_l = rebuildMetaGraph(Ḡ_{l−1}, edges)
       return mergeTree
```

expectation that the memory usage at any point in time is within the limits on a single machine.

Phase 3. At the root partition of the merge tree, we will have a set of internal vertices and cycles (Fig. 1e). We then recursively unroll this coarse-grained information to incrementally reconstruct the full Euler circuit, using the details persisted to disk. This path can be pushed to disk as it is created. Starting at any vertex in this root partition, we unroll the edges of a local cycle that it initiates until we reach a *pivot vertex*, which is a vertex that is present on *more than one* odd degree path, even degree cycle or an internal cycle. We then switch to recursively unrolling edges of a different path or cycle passing through this pivot vertex and created at a lower level, until we return to the pivot vertex, and resume our earlier unrolling. This continues until all edges are emitted. This combines the merge tree with the partial paths and cycles identified at different levels into the final circuit, in a single pass.

C. Algorithms

We describe the algorithms for the first two key phases here. We omit a detailed algorithm for Phase 3, for brevity.

1) Phase 1: Alg. 1 will be executed concurrently in a partition-centric manner on all the partitions of the initial graph, and on each merged partition after Phase 2. We initialize all boundary vertices and local edges to be unvisited in lines 2–5, and create an empty $pathMap$ structure in line 6, which will have the local paths and cycles found in this partition. We then call FINDEULERPATH for all unvisited odd degree boundary vertices, in lines 7–8. This procedure navigates along unvisited edges, marking them and their incident vertices as visited, and returns a maximal path [4]. We now state the following lemmas (see [26] for proof).

Lemma 1. *A maximal local path which starts at an odd degree boundary vertex will always end at another odd degree boundary vertex.*

Thus, if we have $2n$ OB vertices, this will find exactly n paths.

Once all OB paths are exhausted, we find cycles starting at EBs using FINDEULERPATH. Here, we traverse from every EB exactly once, as shown in lines 9–10, and include even a trivial singleton path with zero edges.

Lemma 2. *A local traversal that starts at an even degree boundary vertex or an internal vertex will always end at the same vertex, thus completing a cycle.*

i.e., a maximal path will result in a cycle, and if we have m EB vertices, we will find exactly m cycles or singletons.

Lastly, if there still remain any unvisited edges, then either of their incident vertices must be an internal vertex. We initiate a maximal traversal from this vertex, in lines 11–13, which will form a cycle, $ivCycle$ (Lemma 2).

Lemma 3. *At least one vertex in any cycle starting at an internal vertex must also be part of a previous odd vertex path or even vertex cycle in the same partition.*

MERGEINTO combines the internal vertex cycle into one of these prior paths/cycles, at a (pivot) vertex at which they intersect, thus extending the prior ones. The $pathMap$ assigns a unique *Path ID* to each cycle or path added to it, its *type* (OB path or EB cycle), the *source* and optionally (if it is a path) the *sink boundary vertex ID*, and the *remote edges* for the boundary vertices. The actual vertices and edges in the path/cycle can be persisted to disk. The $pathMap$ data structures is adequate to proceed to the next phase, and other in-memory states of the partition can be removed.

2) Phase 2: This phase first identifies the pairs of partitions that will be merged recursively, starting with the initial partitions of the graph, until just a single merged partition remains. Alg. 2 builds this *merge tree* statically on 1 machine, using just the *meta-graph* – the list of partition IDs as meta-vertices, their meta-edge connectivity, and their edge weights – which is small in size, $\mathcal{O}(n(n-1))$, with n partitions and a maximum of $n(n-1)$ meta-edges between them.

Alg. 2 uses a greedy approach, with the intuition that partitions with the most edges between them should be merged first as it allows for the consumption of more local edges in the next level's execution of Phase 1. In lines 2–4, we initialize an empty $mergeTree$, set the level as 0, and set the meta-graph at this level as the original meta-graph with all partitions. In lines 5–9, we build one level of the merge tree for each iteration, until only one partition remains. For each level, we find a *maximal matching* of meta-edges over the meta-graph such that the sum of the meta-edge weights is the highest and a meta-vertex appears exactly once, in one of the selected meta-edges. If one or more of the meta-vertices cannot be chosen, they are skipped. MAXIMALMATCHING sorts the meta-edges in descending order of their weights and greedily selects them while ensuring none share a common meta-vertex.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 18,2022 at 06:55:45 UTC from IEEE Xplore. Restrictions apply.

The pairs of meta-vertices (partitions) on each selected edge form siblings in the tree at level l, with the parent being one of the two partitions (e.g., the one with a larger partition ID) that is merged into at the next level. We then rebuild the meta-graph for the next level using the merged partitions as the meta-vertices, and repeat this process.

The merge tree for the graph in Fig. 1a is shown in Fig. 1f. Here, the edge weights between P_3 and P_4 is the highest, and they are selected for merging at level 0. That leaves P_1 and P_2 as the remaining partitions to be merged. We pick P_2 and P_4 as the parents. In the next level 1, these two merged partitions are merged again, and into a single partition P_4.

While the merge tree is constructed at the start, the actual merging of two partitions is done after executing Phase 1 on them at each level. This involves transferring the $pathMap$ from the child partitions to the parent, converting the remote edges between the children to local edges, and introducing relevant local and boundary vertices. Once merged, we invoke Phase 1 on each merged partition, in parallel, for this level.

D. Complexity Measures

There are three costs associated with our distributed partition-centric approach. The *coordination cost* is the total number of iterations (supersteps) that the BSP execution takes. Phase 1 executes on all partitions at a level in *one superstep, in parallel,* and Phase 2 constructs a *full* binary merge tree. So the coordination complexity is the height of the tree, given by $\lceil log(n) \rceil + 1$, for n partitions in the original graph.

The *computation complexity* is dominated by the Phase 1 algorithm. The complexity of Alg. 1 executing on each partition is the time to initiate traversals from each boundary vertex and, in the worst case, every internal vertex, and the time to visit every local edge once. This is $\mathcal{O}(|B_i|+|I_i|+|L_i|)$, for a partition P_i. This applies to leaf and merged partitions. For each level, all partitions execute concurrently, and so this is bound by the largest partition in a level, and for all levels.

Lastly, the *communication cost* is contributed by the merging of pairs of partitions as part of Phase 2, at each level. The $pathMap$ output from Phase 1 for a partition is sent to the other partition it merges into. The odd and even boundary vertices, and their remote edges form a bulk of this. So the communication complexity is $\mathcal{O}(|B_i| + |R_i|)$, for partition P_i merging into another. Since data transfers happen in parallel at a level, this is bound by the largest partition in a level.

IV. EXPERIMENTS

A. Implementation

We implement the Phase 1 and 2 algorithms on *Apache Spark 2.2* [16] with Java 8, using a partition-centric approach [2]. The implementation of Phase 3, which can be done sequentially, is left to future work. Since Spark does not naturally expose a partition-centric model, we model all partitions in one level as a single *Resilient Distributed Dataset (RDD)*, with each of our partitions mapping to a Spark partition in the RDD.

[2]https://github.com/dream-lab/euler-circuit

Table I: Characteristics of Input Eulerian graphs used

Graph	$\lvert V \rvert$	$\lvert E \rvert$	$\sum_{i \in n} \lvert B_i \rvert$	**Parts** (n)	$\sum \frac{\lvert R_i \rvert}{\lvert E \rvert}$%	$\lvert V_i \rvert$ **Imbal.%**
G20/P2	20M	212M	13M	2	38%	19%
G30/P3	30M	318M	22M	3	49%	48%
G40/P4	40M	423M	23M	4	59%	46%
G40/P8	40M	423M	33M	8	70%	41%
G50/P8	49M	529M	40M	8	70%	63%

Phase 1 at a level is designed as a set of *transformations* on the RDD to form a $pathMap$ per partition, and Phase 2 generates a new RDD for the next level by merging partition pairs, based on the merge tree computed offline. This repeats iteratively till an RDD with just one partition remains. We choose Spark over other popular graph processing systems like Giraph [12] or GraphX [13] because of the ease of abstraction in exposing a partition-centric model.

B. Experiment Setup

We run the application on a Spark cluster deployed on 8 Microsoft Azure `E8s v3` Virtual Machines (VMs), with each having 8 cores of Intel Xeon E5-2673 at 2.3 GHz, 64 GB of RAM, and 128 GB of local storage. Each Spark *executor* runs on a separate VM, has access to 45 GB of RAM, and *one executor is assigned to each partition* in the input graph.

The input we expect is an Eulerian graph, i.e., each vertex has an even edge degree. But there are no well-known tools for generating such graphs. So we first generate a *powerlaw graph* using the *parallel RMAT tool* [27], and develop a custom tool to add additional edges between vertices that have an odd degree, to make the graph Eulerian. The tool ensures that the edge degree distribution of the modified graph closely matches the original graph. In practice, the extra edges added is $\approx 5\%$. Subsequently, we use the *ParHip tool* [24] to partition these graphs, with the time taken to generate these partitioned graphs ranging from *6 mins* to *18 mins*. We use a parallel partitioner instead of a centralized one as the gains in terms of scalability, overhead and performance are higher compared to a shared memory centralized partitioning tool like METIS. Table I lists the graphs that we generate, and counts of their vertices, edges, boundary vertices, partition, edge-cut fraction, and peak vertex imbalance across partitions, given as $\max_i \left| \frac{|V|-n\times|V_i|}{|V|} \right|$. The bi-directed edge counts listed are twice the number of undirected edges. We run the parallel RMAT tool with its default settings, and an average undirected edge degree of 5.

C. Results

We examine the **total time** taken to execute our Eulerian circuit Spark application, along with the **user compute time** taken strictly within our code logic for the 5 candidate graphs, as reported in the Spark logs in Table II. The difference between the two is the platform's overhead for data transfers, coordination, scheduling tasks, etc. We observe that the *weak-scaling* is inefficient. The graphs G20/P2, G30/P3 and G40/P4 have the same ratio of input load per unit compute resource ($\approx 10M$ vertices per VM). While the ideal weak-scaling times for these should be constant, instead they linearly increase.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 18,2022 at 06:55:45 UTC from IEEE Xplore. Restrictions apply.

Table II: *Total* and *user compute times* for each graph

Metric	G20/P2	G30/P3	G40/P4	G40/P8	G50/P8
Total Time (min)	10	16.4	28.5	21.6	30.5
Compute Time (min)	4.3	7.4	12.9	10.8	16.1

Doubling the resources between G40/P4 to G40/P8 does not halve the time taken, indicating that *strong-scaling* is poor as well. As partition-count increases, the height of the merge-tree grows logarithmically. This increases the coordination costs for the partition-centric model, taking $2, 3, 3, 4$ supersteps for $2, 3, 4, 8$ partitions. So there is a trade-off between using less compute time against more cumulative data transfers and coordination costs.

We do see that the user compute time is just half of the total time, and also grows at a slower rate. This indicates that Spark's distributed data transfer and I/O as part of its *shuffle*, and the coordination costs for on-demand scheduling of tasks (e.g., 44 tasks are scheduled during the lifetime of G50/P8) and barrier synchronization at each application stage (e.g., 15 stages for G50/P8) dominate. They degenerate as the the graph size increases and the machines to transfer data across the commodity network increases.

We further examine the **user compute time splits** for our application. Fig. 2a shows a stacked bar plot of various user times, spent in the Phase 1 computation (pink), and the Phase 2 partition merging (rest), for the *G50/P8 graph*, across different partitions at different levels of the merge tree. In level 0 with 8 partitions, $P0 - P7$, much of the time is spent in constructing the graph object from its storage format (*Create partition object*) with only $\approx 33\%$ of the time, on average, spent on the Phase 1 computation. Here, the graph sizes are large on disk (≈ 605 MB per partition for G50/P8) and the I/O and Java object costs dominate. This continues in level 1 as well, with the additional cost of serializing, deserializing and merging the partition object pairs. In levels 2 and 3, each merged partition grows large enough that the compute time for Phase 1 starts to dominate with fewer partitions to move and merge, taking $\approx 48\%$ in level 2 and $\approx 51\%$ in level 3. We also observe that as the levels increase, the overall compute time increases as well. This shows that the time taken to process a merged partition is higher as more partitions are merged into one.

These illustrate the challenges of scaling graph algorithms on distributed memory machines, even for algorithms like Euler circuits that have modest complexity. It also highlights the inefficiencies of Big Data platforms like Spark in handling graph data with irregular structure, with substantial time spent on data movement (albeit coarse-grained), and Java object management, compared to processing text or tuple data.

Next, we specifically look at the **expected and observed times taken by Phase 1**, where the bulk of our actual graph traversal and circuit identification take place. Fig. 2b shows a scatter plot of the *expected time complexity* of Phase 1 for each partition at each level, $\mathcal{O}(|B_i|+|I_i|+|L_i|)$, on the X axis, and the corresponding *observed time in seconds*, on the Y axis. There are 8, 4, 2 and 1 partitions in levels 0, 1, 2 and 3, for a total of 15 data points. We also show a dashed linear trend-line through these points, calculated by identifying the line that reduces the squared sum of perpendicular distances from the data points. We also show a solid line with the asymptotic complexity, i.e., all observed values lie below this, calculated by identifying a line with a slope higher than the maximum slope amongst lines passing through the data points.

We can clearly see that the expected time complexity closely matches the observed times. So the computational cost for the critical Phase 1 algorithm is consistent with our design and analysis. We do see some outliers, e.g., in level 2. This is due to a skew in one of the two partitions at that level, as is seen in Fig. 2a for P3 at level 2. This is a consequence of the partition imbalances in level 0 caused by the external partitioner, accumulating at higher levels.

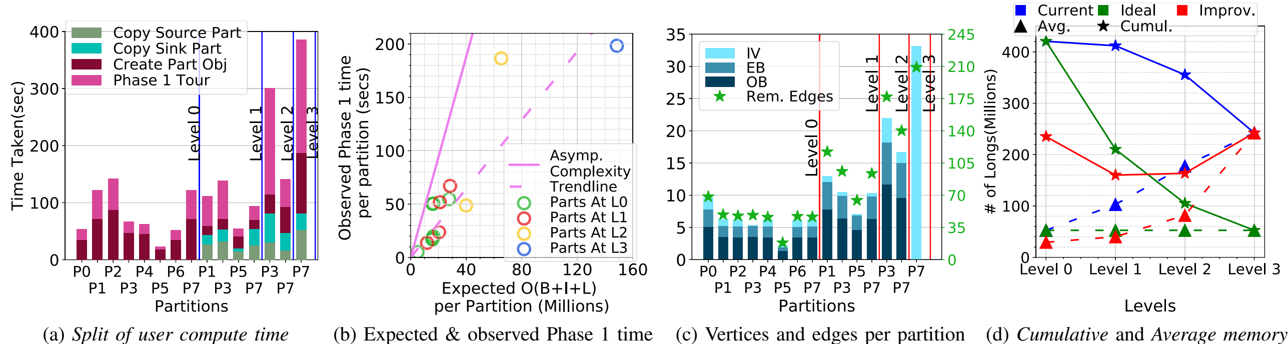
A key factor that limits our scaling is the **memory consumed** by the partitions at various levels. Each (merged) partition's state must fit in the memory of a single machine. Our algorithm design monotonically reduces the total in-memory state maintained across all partitions as we go up the level. This is done by replacing edge IDs of paths and cycles found in Phase 1 with a single path/cycle ID. But we also merge partitions up the level to allow remote edges to be included in the circuit, thus causing the partition sizes to increase. In an *ideal case*, the rate of drop in the state due to Phase 1 and the rate of increase in the state due to merging partitions should be identical, thus requiring a constant memory usage for partitions at any level. We study this behavior of the *memory state* as the algorithm progresses.

Fig. 2d plots the *cumulative (solid line) and average (dashed) number of* `Int64` (i.e., 8-byte `Long`) values that are maintained as part of the partitions' state at different levels. This is reported from within our user code, and is a platform-independent metric of the algorithm's memory use, compared to reporting the raw GB of RAM used as it is affected by the Java object structure, garbage collector, etc. We report this for our *current algorithm* as blue lines. The cumulative is the distributed memory state at a level across all partitions, while the average is the per-machine (partition) size. We also report the *ideal case* in green as a synthetic metric. Here, the state size for a merged partition matches the initial state size of its child partitions – the average is constant across levels, while the cumulative is this times the number of partitions at a level.

We see that there is a *monotonic drop* in the cumulative state as the levels progress for our current algorithms, as expected. But the *rate of drop is inadequate*. This is seen by the growth in the average partition size (blue dashed). While we expect the average size to be at $\approx 50M$ Longs, as seen at level 0, for G50/P8, this instead grows to $\approx 200M$ at level 3. Level 3 has just 1 merged partition rather than the 8 at level 0. But its size is 50% of the cumulative size at level 0, rather than 12.5% we expect.

Also, the drop in cumulative size is more at higher levels rather than lower. In an ideal case, it should be exponential and drop more initially (concave green vs. convex blue solid lines). So the memory pressure is relieved only toward the last level. As a consequence, we are unable to run larger graphs

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 18,2022 at 06:55:45 UTC from IEEE Xplore. Restrictions apply.



(a) *Split of user compute time* (b) Expected & observed Phase 1 time (c) Vertices and edges per partition (d) *Cumulative* and *Average memory*

Figure 2: Results on *User Compute time*, *Phase 1 time*, *Memory state size* and, *Vertices and edges* on G50/P8

with weak-scaling of memory as the required average memory increases with more levels.

In Fig. 2c (left Y axis), we examine the **number of vertices of different types** per-partition at *the start of* Phase 1, for different levels of G50/P8, to analyze the drop rate. Boundary vertices have local and remote edges. Since only the local edges are consumed by Phase 1, the remote vertices are just carried over to the merged partition (right Y axis), with only some becoming local edges. The plots show that the number of boundary vertices and remote edges keep growing as we merge partitions. In particular, the remote edge count is $\approx 7\times$ the vertex count, and dominates the memory usage.

Lastly, we report that despite the use of efficient Java data structures, the overhead between the minimal raw bytes taken by the Longs and the actual memory used by the application is $\approx 10\times$. E.g., a partition that is $950MB$ on disk has a memory footprint of $12.8GB$ within the Spark application. At the root of the merge tree, the G50/P8 partition consumes almost all of the JVM's allocated $45GB$ heap space.

V. Proposed Improvements to Partition Merging

The analysis in the previous section shows that our proposed distributed algorithm for finding the Euler circuit is unable to weakly scale due to memory pressure. The drop rate of the partition sizes does not keep up with the growth rate due to the merging. This is primarily due to the remote edges in the merged partitions, as they accumulate up the levels of the merge tree.

In this section, we propose heuristics to reduce the number of remote edges that are maintained in memory to enable the algorithm to scale. We describe these techniques and analyze their potential benefits, but defer an implementation and empirical evaluation to future work.

Avoid Remote Edge Duplication. Our partitioned graph models undirected edges as a pair of directed edges, as is common, to simplify traversals [12]. However, this doubles the memory usage for maintaining edges. As we see from Table I, large graphs with many partitions have 70% of their edges remote, and this duplication worsens the memory pressure.

Intuitively, if two partitions are merging, it is sufficient for only one of them to hold a *directed remote edge* between them, and convert it to a *pair of directed internal edges* after they merge. This allows the other partition to drop its remote edges to this partition. E.g., in Fig. 1a, rather than P1 and P2 maintain the remote edges $e_{2,3}$ and $e_{3,2}$ to each other, respectively, only one needs to retain it, say $e_{2,3}$ in P1. When they merge into P2 in the next level, we convert the remote edge $e_{2,3}$ into a pair of internal edges $e_{2,3}$ and $e_{3,2}$ before Phase 1.

The merge tree gives the partitions that will be merged at different levels, and this helps us decide at graph loading time which partition should retain the remote edges among a pair that will (eventually) merge. Here, we select the partition that is *heavier* among the pair, i.e., has more number of cumulative remote edges, as the one to drop its remote edges. This relieves its memory pressure, and also reduces the amount of state transfered from one level to another. Using this, the cumulative remote edge count kept in-memory will *halve* at each level.

Defer Transfer of Remote Edges. In our current strategy, the entire $pathMap$ from a partition is transfered to a parent partition when they merge. This includes remote edges between the two partitions, and remote edges between the source partition to other partitions with which it will merge higher up the tree. E.g., in Fig. 1f, when P1 and P2 merge into P2, P1 sends it its list of remote edges to P2, P3 and P4. While the edges to P2 become internal edges, the ones to P3 and P4 will only be used in level 2 but are in P2's memory in level 1.

In addition, we reduce the number of machines that are used to hold the merged partitions as we go up the merge-tree levels. So, while the total distributed memory on all machines is constant, the available distributed memory is halved at each level, even as the required memory does not drop as fast.

Combining these two intuitions, we propose to delay copying of the remote edges from a child to an ancestor partition, until we are at the level at which the ancestor is being merged. This reuses the machines on which the *inactive leaf partitions* are present to hold the deferred remote edges, and sends the relevant remote edges to the *active ancestor partition* at a higher level just before the latter's Phase 1 executes.

In Fig. 1f, the partitions P1–P4 retain their remote edges on their host machines' memory even after they complete level 0. Before level 1, P1 sends only its remote edges to P2 when it merges with it, but retains its remote edges to P3 and P4. Then,

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 18,2022 at 06:55:45 UTC from IEEE Xplore. Restrictions apply.

before level 2, P1 sends it remote edges to P3 and P4 to the merged ancestor P4, where they become internal edges are are used in P4's Phase 1. As a result, we use memory available on machines hosting the leaf partitions, reduce memory pressure on the ancestors, and minimize the number of transfers of the the remote edges up the levels.

Analysis. We analytically model the impact of these two strategies on the memory usage of G50/8P in Fig. 2d, based on the previous experiments' traces. The dashed and solid red lines indicate the average and cumulative memory state expected. We see that the total memory state (solid red) in level 0 is 43% smaller than the current approach (solid blue) due to avoiding holding the duplicate remote edges. As we saw in Fig. 2c, the remote edge counts outstrip the vertex counts, and hence this steep drop. As the levels progress, we see the additional benefits from the deferred transfers. The total memory state drops more sharply from level 0 to levels 1 and 2, compared to the current approach. This is on top of the reduced base. The average state maintained in the active partitions (dashed red) grows slower in levels 1 and 2, and is 50–75% smaller than the current approach (dashed blue). We are close to the ideal static value (dashed green); the reason the "ideal" is larger than the proposed for levels 0 and 1 is due to the avoiding remote edge duplication. Lastly, we note that our heuristics do not reduce the memory usage in the last level as there are no remote edges in this single merged partition. Mitigating this bottleneck is left to future work.

VI. CONCLUSIONS

We have designed a partition-centric distributed algorithm for finding the Euler circuit over large graphs. Our algorithm finds local paths and cycles within each partition, and recursively merges them to form larger paths and cycles. These progressively reduce the memory used by replacing multiple edges with the path or cycle IDs they appear in. We analyze the complexity metrics, and evaluate its performance using Apache Spark. While the time complexity for Phase 1 is as expected, the overheads of data transfer in Spark's shuffle stage and object creation in Java reduce the benefits. Also, the memory pressure grows with the levels of merging due to remote edges that accumulate. We address this using heuristics to mitigate the growth in remote edge state and data transfers, and our analysis shows that these can reduce the memory usage by 50–75%. Its empirical validation is left to *future work*, as is optimizing the last merged partition and evaluating the scalability for larger graphs. We will also consider generalizing this to non Eulerian graphs, by allowing edge revisits as well as to time-dependent dynamic graphs.

REFERENCES

[1] L. Eulero, "Solutio problematis ad geometriam situs pertinentis," *Commentarii Academiae scientiarum imperialis Petropolitanae*, vol. 8, pp. 128–140, 1741.

Acknowledgment. This work was supported by a Microsoft Data Science Fellowship provided to the first author, a Microsoft Azure grant for access to cloud resources, and a research grant from VMWare Inc.

[2] "Districting for salt spreading operations," *European Journal of Operational Research*, vol. 139, no. 3, pp. 521 – 532, 2002.

[3] K. Roy, "Optimum gate ordering of cmos logic gates using euler path approach: Some insights and explanations," *Journal of computing and information technology*, vol. 15, no. 1, pp. 85–92, 2007.

[4] C. Hierholzer and C. Wiener, "Über die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren," *Mathematische Annalen*, vol. 6, no. 1, pp. 30–32, 1873.

[5] M. Atallah and U. Vishkin, "Finding euler tours in parallel," *Journal of Computer and System Sciences*, vol. 29, no. 3, pp. 330–337, 1984.

[6] B. Awerbuch, A. Israeli, and Y. Shiloach, "Finding euler circuits in logarithmic parallel time," in *ACM SoTC*, 1984.

[7] S. Makki, "A distributed algorithm for constructing an eulerian tour," in *IEEE IPCCC*, 1997.

[8] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *PVLDB*, vol. 7, no. 14, 2014.

[9] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to dna fragment assembly," *PNAS*, vol. 98, no. 17, pp. 9748–9753, 2001.

[10] E. Benson, A. Mohammed, J. Gardell, S. Masich, E. Czeizler, P. Orponen, and B. Högberg, "Dna rendering of polyhedral meshes at the nanoscale," *Nature*, vol. 523, no. 7561, p. 441, 2015.

[11] A. Yazici, G. Kirlik, O. Parlaktuna, and A. Sipahioglu, "A dynamic path planning approach for multirobot sensor-based coverage considering energy constraints," *IEEE Trans. on Cybernetics*, vol. 44, no. 3, 2014.

[12] "Apache giraph," https://giraph.apache.org/.

[13] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework." in *OSDI*, vol. 14, 2014, pp. 599–613.

[14] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "GoFFish: A sub-graph centric framework for large-scale graph analytics," in *EuroPar*, 2014.

[15] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *PVLDB*, vol. 7, 2013.

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, 2010.

[17] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *POOSC*, vol. 2, pp. 1–18, 2005.

[18] A. Chan, F. Dehne, and R. Taylor, "Cgmgraph/cgmlib: Implementing and testing cgm graph algorithms on pc clusters and shared memory machines," *IJHPCA*, vol. 19, no. 1, pp. 81–97, 2005.

[19] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *ACM PPoPP*, 2016.

[20] 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 *ACM SIGMOD*, 2010, pp. 135–146.

[21] R. J. Anderson and L. Snyder, "A comparison of shared and nonshared memory models of parallel computation," *Proc. of IEEE*, vol. 79, 1991.

[22] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song, "Efficient parallel graph algorithms for coarse grained multicomputers and bsp," in *International Colloquium on Automata, Languages, and Programming*, 1997.

[23] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter, "External-memory graph algorithms." in *SODA*, 1995.

[24] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel Graph Partitioning for Complex Networks," vol. 28, no. 9, 2017, pp. 2625–2638.

[25] R. Dindokar, N. Choudhury, and Y. Simmhan, "A meta-graph approach to analyze subgraph-centric distributed programming models," in *IEEE International Conference on Big Data (Big Data)*, 2016.

[26] S. Jaiswal and Y. Simmhan, "A partition-centric distributed algorithm for identifying euler circuits in large graphs," *arXiv preprint arXiv:1903.06950*, 2019.

[27] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *IEEE/ACM PACT*, 2015.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on October 18,2022 at 06:55:45 UTC from IEEE Xplore. Restrictions apply.