A Single Machine System for Querying Big Graphs with PRAM

Yang Liu[1], Wenfei Fan[2,1,3], Shuhao Liu[2], Xiaoke Zhu[1], Jianxin Li[1]
Beihang University[1], China Shenzhen Institute of Computing Sciences[2], China University of Edinburgh[3], UK
ly_act@buaa.edu.cn,wenfei@inf.ed.ac.uk,shuhao@sics.ac.cn,zhuxk@buaa.edu.cn,lijx@buaa.edu.cn

ABSTRACT

This paper develops Planar (Plug and play PRAM), a single-machine system for graph analytics by reusing existing PRAM algorithms, without the need for designing new parallel algorithms. Planar supports both out-of-core and in-memory analytics. When a graph is too big to fit into the memory of a machine, Planar adapts PRAM to limited resources by extending a fixpoint model with multi-core parallelism, using disk as memory extension. For an in-memory task, it dedicates all available CPU cores to the task, and allows parallelly scalable PRAM algorithms to retain the property, *i.e.,* the more cores are available, the less runtime is taken. We develop a graph partitioning and work scheduling strategy to accommodate subgraph I/O, balance memory usage and reduce runtime, beyond traditional partitioners for multi-machine systems. Using real-life graphs, we empirically verify that Planar outperforms SOTA in-memory and out-of-core systems in efficiency and scalability.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SICS-Fundamental-Research-Center/Planar.

1 INTRODUCTION

A host of single-machine systems have been developed for graph analytics via multi-core parallelism, *e.g.,* [6, 35, 57, 64, 65, 72, 78, 83, 89, 94, 99, 100, 105]. These systems typically adopt a *vertex-centric* (VC) or *edge-centric* (EC) parallel model. A VC (resp. EC) program pivots computation around each vertex (resp. edge); it may only directly access its local data and adjacent edges (resp. endpoints), but it has to exchange information with "remote" entities via message passing. VC/EC is often inefficient in programming and execution [6, 29, 40, 86]. It is nontrivial to program for problems that are constrained by "joint" conditions on multiple vertices, *e.g.,* subgraph isomorphism [32]. Moreover, the local scope of VC/EC operations often incurs redundant computation [86], and its message-passing model introduces extra synchronization complexity [5]. These overheads are often excessive, leading to limited scalability and high COST [68] of a graph system under a shared-memory architecture.

On the other hand, parallel models have been studied for shared-memory architectures for decades, notably PRAM (Parallel Random Access Machine) [30, 36, 88]. PRAM allows multiple processors to work in parallel via *single-instruction-multiple-data* (*SIMD*), and synchronize via shared memory. A large number of PRAM algorithms are already in place, and many of them are provably work-time optimal [44] or parallelly scalable, *i.e.,* they guarantee that the more processors are used, the less parallel runtime is taken [56].

Is it possible to develop a single-machine graph system in which one can plug existing PRAM algorithms, and the system executes the algorithms to make the most of multi-core parallelism and the shared memory of the machine? This way, the users do not have to think like a vertex/edge and develop new parallel algorithms starting from scratch; instead, they can simply leverage the decades of work on PRAM and make effective use of the well-developed PRAM algorithms, capitalizing on their scalability and efficiency.

It is, however, nontrivial to run PRAM algorithms in a single-machine system. The PRAM model assumes that the memory is large enough to load the entire dataset at once, and there are a polynomial number of processors [7, 38]. In contrast, a single-machine system has a fixed number of CPU cores, limited memory capacity and disk I/O bandwidth. One has to simulate a polynomial number of cores assumed by PRAM. Worse yet, for *out-of-core* processing of graphs that are too large to fit into the main memory of a machine at once, it needs to use disk as memory extension [6, 35, 57, 65, 78, 94, 105]. Such systems have to carefully partition graphs and schedule I/O and CPU operations so that their CPUs do not have to wait for long for subgraphs to be loaded into the memory.

Contributions & Organization. This paper develops Planar (Plug and play PRAM), a single-machine system for running PRAM algorithms for graph analytics. Underlying Planar are the following.

(1) Parallel model (Section 3). Planar proposes a unified parallel model for both in-memory and out-of-core tasks. For a query class Q, it takes as input an existing PRAM algorithm $\mathcal{A}$ and a graph G. When G fits into the memory of a single machine, it executes algorithm $\mathcal{A}$ on G with all available cores. When the graph is too big, it partitions G into subgraphs such that each subgraph can fit into the memory, and uses the secondary storage as memory extension. It loads the subgraphs into memory one by one, and runs SIMD for multi-core parallelism on each in-memory subgraph with all available cores. It iterates the computation over all subgraphs until it reaches a fixpoint, adapting the graph-centric model (GC) [29].

The parallel model makes the first effort to adapt PRAM to physical machines in the real world. It simplifies parallel graph programming by reusing existing PRAM algorithms, and retains their parallel scalability for in-memory tasks. For out-of-core tasks, it extends the data partitioning parallel model of GC [29] with multi-core parallelism and shared-memory synchronization.

(2) Partitioning and scheduling (Section 4). We study a new problem, which aims to overlap CPU and I/O operations, balance the use of limited memory and cope with the dynamic behaviors of iterative rounds; these new challenges are not encountered by graph partitioners for multi-machine systems. We show the intractability of the problem, and develop an effective joint partitioning and scheduling strategy. As preprocessing, we partition the graph into small blocks; at runtime, we adapt to available memory and system bottleneck dynamically via grouped block processing.

(3) Experimental evaluation (Section 5). Using real-life and synthetic graphs, we empirically find the following. For weakly connected components (WCC), single-source shortest path (SSSP), PageRank (PR), vertex coloring (Coloring), minimum spanning tree

(MST), and random walk (RW), (a) Planar outperforms the state-of-the-art (SOTA) out-of-core systems by 34.42× on average, up to 302.01×. (b) On average it is 5.62× faster than the SOTA in-memory system. For parallelly scalable PRAM algorithms, it beats the SOTA by 5.91–9.58×; with 6× cores, it speeds up by 3.36×. (c) Its adaptive partitioning and scheduling strategy improves performance by 1.87–2.12×. (d) It performs as well as the SOTA multi-machine systems that use 4–10 machines, saving the monetary cost by at least 81.7%.

We discuss PRAM in Section 2 and future work in Section 6. We defer proofs and Planar programs to [2] for the lack of space.

Related work. We categorize the related work as follows.

Parallel models. Several models are in place for graph analytics. (1) *Vertex-centric* (VC) [66, 72, 83] and *edge-centric* (EC) [65, 78, 105] models parallelize computation centered around graph neighborhoods, by programming from the perspective of a single vertex/edge. This makes some graph algorithms inefficient for, *e.g.,* graph simulation [26]. (2) *Graph-centric model (*GC*)* [24, 29] parallelizes sequential graph algorithms across subgraphs, for user to think like a graph. (3) *Hybrid model* [106] adopts the data partitioning parallelism of GC and operation-level parallelism of VC at a single machine.

PRAM [30, 36, 88] supports SIMD parallelism for general computation. It facilitates interprocessor communication and synchronization via shared memory. However, several practical difficulties arise in mapping PRAM algorithms onto real-life physical machines [7], *e.g.,* the fixed number of CPU cores and limited memory capacity. Some programming models, *e.g.,* ICE [34] and XMTC [58], allow users to write lockstep programs similar to PRAM algorithms. These models, however, do not support out-of-core computing.

Planar proposes a parallel model to fit single-machine shared-memory parallelism. As opposed to message passing-based VC/EC, it supports subgraph-based processing beyond neighborhood, and synchronizes via shared memory, reducing redundant work and I/O. Moreover, it simplifies parallel graph programming by reusing existing PRAM algorithms and retaining their parallel scalability.

Planar extends GC in the following. GC was designed for multi-machine systems that load all subgraphs just once to different machines and process the subgraphs simultaneously via message passing. It supports neither intra-subgraph parallelism nor out-of-core computation. In contrast, Planar targets multi-core parallelism at a single machine. It separates (a) intra-subgraph parallelism via SIMD parallelism and shared-memory synchronization of PRAM, from (b) inter-subgraph parallelism by simulating the fixpoint model of GC and partitioning/scheduling graphs for out-of-core tasks. It supports both in-memory and out-of-core computations.

Planar simplifies the VC programming and execution of hybrid [106] by reusing PRAM programs, retaining their parallel scalability, and demanding neither code revamp nor manual tuning of hybrid.

Single-machine systems. (1) *In-memory ones* [35, 64, 72, 83, 99, 100, 102] assume that a graph can be loaded entirely into memory. They adopt variants of VC/EC [72, 99, 100], and improve data locality via scheduling [99]. (2) *Semi-external* systems (Blaze [52] and [62, 101]) fit all vertex data in memory, and load (immutable) edge data from the secondary storage on-demand. They cannot handle graphs with a large number of vertices. (3) *Out-of-core systems* [6, 52, 57, 62, 65, 78, 89, 101, 105] use disk as memory extension, and focus on reducing the I/O cost of swapping data between the disk and memory. CLIP [6] adopts an asynchronous model to reduce redundant synchronization cost of EC, which may compromise the correctness. All the previous systems adopt VC/EC, except MiniGraph [106] that employs a hybrid model.

Planar supports a new parallel model to speed up in-memory and out-of-core graph computations, outperforming SOTA of both types (Section 5). For out-of-core execution in particular, it proposes an adaptive strategy for partitioning and scheduling, to cope with the dynamic runtime behavior, and reduce I/O and parallel runtime. These challenges are not encountered by in-memory systems.

Hardware-accelerated systems [21, 51, 65, 91, 97, 103] leverage GPUs or FPGAs to speed up graph computation. These systems propose various optimizations to leverage hardware features, *e.g.,* a memory hierarchy with fine-grained accesses, low-level task optimization, and massive hardware parallelism, which are not accessible in CPU-based systems like Planar. Despite the significant speedups, they come with several trade-offs, including a high cost, increased programming complexity, and reduced flexibility in practical use. These factors are at odds with our design objectives in Planar, which aims to provide a cost-effective and general-purpose solution.

Multi-machine systems [22–24, 29, 40, 63, 71, 86, 90, 95, 104] support big graph analytics by scaling out. Such systems adopt a shared-nothing architecture: they partition the input graph, and load the fragments to the machines at once; all workers process their local fragments in-memory in parallel, and synchronize via message passing. The communication cost and workload balancing among workers are thus two vital issues to performance. Rather than scaling out with multiple machines, Planar seeks cost-effective scaling up. It meets new challenges, *e.g.,* limited memory and excessive I/O.

Graph partitioning. For multi-machine systems, the topic has been well studied (see [14, 17] for surveys). (1) Edge-cut [8, 47, 48, 55, 84] partitions vertices into disjoint sets and cuts edges. It promotes locality but may lead to imbalanced fragments [37]. (2) Vertex-cut [21, 43, 67, 74, 75, 98, 103] partitions edges into disjoint sets and allows mirrored vertices. It balances partitions at the cost of locality [18]. Recent out-of-core systems, *e.g.,* [65, 105], employ a 2D partitioner, which enables fast indexing with massive border vertices. (3) Hybrid [9, 18, 20, 25, 60, 104] strikes a balance by combining the two. Representative heuristics include MDBGP [9] and an application-driven partitioner [25], designed for VC and GC, respectively.

The conventional partitioners mostly aim to reduce the replication factor and the balancing ratio. As will be seen in Section 4, these are not of primary concerns for out-of-core systems; in contrast, Planar tackles a unique joint partitioning and scheduling problem. (1) It develops a partitioner by advocating connectivity among subgraphs and locality within a subgraph, not the balancing ratio. (2) It conducts subgraph grouping and scheduling decisions adaptively at runtime, a mechanism not considered by prior partitioners.

2 PRELIMINARIES

This section reviews basic notations and PRAM algorithms.

Graphs. Assume a countably infinite alphabet Ω for labels. Consider graph $G = (V, E, L)$, directed or undirected, where V is a finite set of vertices; $E \subseteq V \times \Omega \times V$ is a finite set of edges, such that

Algorithm 1: Algorithm $\mathcal{A}$ for WCC (Shiloach *et al.* [82]).

Status Declaration: $S_V = \{\bar{p}\}$ where $p(v) = v$ for each $v \in V$;
Input: Graph $G = (V, E, L)$. /*vertex represented by numeric ID. */
Output: The number of weakly connected components in G.
1 **while** E_i not empty **do**
2 **parallel for each** $e = \langle u, v\rangle \in E$ **do** Graft($\langle u, v\rangle$);
3 **parallel for each** $v \in V$ **do** PointerJump(v);
4 **parallel for each** $e = \langle u, v\rangle \in E$ **do** Contract($\langle u, v\rangle$);
5 **return** the number of distinct values in $\bar{p}$;
Procedure Graft ($\langle u, v\rangle$):
6 **if** $p(u) \neq p(v)$ **then**
7 swap u and v if $p(u) > p(v)$; /* ensure $p(u) \leq p(v)$. */
8 $p(p(v)) := p(u)$; /* graft the pseudo-tree of v to that of u. */
Procedure PointerJump (v):
9 **repeat** $u := p(v)$; $p(v) := p(u)$;
10 **until** $p(u) = u$; /* halt if u is the root of its pseudo-tree. */
Procedure Contract ($\langle u, v\rangle$):
11 **if** $p(u) = p(v)$ **then** remove edge $\langle u, v\rangle$;

each edge is labeled with a label in Ω; moreover, each vertex v in V carries a label $L(v) \in \Omega$ to represent properties.

Partition strategies. Given a graph G and a number m, a graph partitioner $\mathcal{P}$ partitions G into *fragments* $\mathcal{F} = (F_1, \ldots, F_m)$ such that each $F_i = (V_i, E_i, L_i)$ is a subgraph of G, $E = \bigcup_{i\in[1,m]} E_i$ and $V = \bigcup_{i\in[1,m]} V_i$. We use $F_i.B$ to denote the set of *border entities* (vertices and edges) that are shared by at least two subgraphs.

PRAM. PRAM is a theoretical model that simplifies the design and analysis of parallel algorithms, particularly in a shared-memory environment. It abstracts the complexities of hardware, assuming the availability of a large number of processors and unlimited shared memory. It supports *SIMD* parallelism: all processors execute the same instruction concurrently on different pieces of data. A key feature is that PRAM allows any processor to access any memory location in constant time, hence the name "random access".

Decades of research have developed a rich set of PRAM graph algorithms. Compared to algorithms designed for message-passing models, they are often provably scalable and more efficient, and make a better fit to multi-core parallelism with shared memory of a single machine. For a class $\mathcal{Q}$ of graph queries, a PRAM algorithm $\mathcal{A}$ takes as input a query $Q \in \mathcal{Q}$ and graph G. It is specified by the following for data, parallelism and computation logic [30, 36, 88].

(1) Status declaration. Given G as a set of data arrays for (V, E, L), algorithm $\mathcal{A}$ declares and initializes a set of *status variables*, which are auxiliary data structures used by $\mathcal{A}$ that represent the intermediate states of G. These include (a) variables associated to individual graph elements, *i.e.,* a set of *vertex* (resp. *edge*) *status*, denoted by S_V (resp. S_E), one for each vertex (resp. edge) in G; and (b) variables that represent the overall state of G, *i.e.,* a set S_G of *global status*, which is particularly important for coordinating the overall control flow, *e.g.,* maintaining counters or flags that influence algorithm termination. We use $S(G)$ to denote tuple (S_V, S_E, S_G), referred to as the *status* of G *w.r.t.* Q, keeping track of the computation.

(2) Processor allocation. With intermediate state $R(G) = (G, S(G))$ in shared memory, $\mathcal{A}$ conceptually assigns each processor to a unique memory location (*e.g.,* a vertex or an edge) for SIMD parallelism. The assignment allows any processor to access any part of $R(G)$ to perform read, compute, or write operations. Note that $\mathcal{A}$ assumes n processors where n is a polynomial in $|V|$ and $|E|$.

(3) Lockstep. Algorithm $\mathcal{A}$ specifies its logic of computation in a sequence of operations for SIMD parallelism, possibly with conditionals and loops. Each operation is called a *lockstep*, where all processors execute the same instruction simultaneously on different pieces of data in the shared memory. The synchronization is enforced by a barrier at the end of each lockstep, ensuring all processors are ready before proceeding to the next instruction. The locksteps update $R(G)$ in place; they produce the final state $R'(G) = \mathcal{A}(Q, G)$ at the end of the sequence, which yield $Q(G)$.

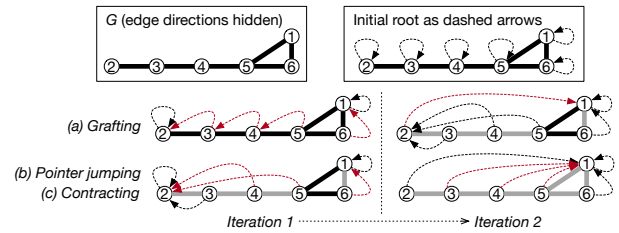


Figure 1: WCC computation over sample graph G with Algorithm 1.

Example 1: Consider WCC. Given a graph $G = (V, E, L)$, it counts the number of maximum subgraphs of G in which all vertices are connected to each other via a path, regardless of the edge direction.

The PRAM algorithm $\mathcal{A}$ of [82] is shown Algorithm 1. Using $|E| + |V|$ processors, it computes WCC of G in $O(\log |V|)$ time. It maintains a disjoint set of *pseudo-trees*. A pseudo-tree rooted at vertex r, denoted by $\Lambda(r)$, is a tree-like structure where each vertex v has a parent $p(v)$ that points to another vertex in $\Lambda(v)$, except that the root r is its own parent, *i.e.,* $r = p(r)$. Algorithm $\mathcal{A}$ iteratively merges these pseudo-trees based on edge connectivity, maintaining the invariant that all vertices in the same pseudo-tree are weakly connected. On graph G of Figure 1, it works as follows.

(1) Status. Algorithm $\mathcal{A}$ declares a *parent pointer* $p(v)$ for each $v \in V$, initialized to itself and represented as dashed arrows in Figure 1. The pointers are stored as an array $\bar{p}$ within the vertex status S_V.

(2) Processors. It virtually allocates a processor to each edge in E and each vertex in V, allowing all to be processed in parallel.

(3) Locksteps. $\mathcal{A}$ maintains and updates parent pointers $\bar{p}(v)$ in a loop that comprises three lockstep operations. Figure 1 illustrates changing parent pointers as the red dashed arrows in each lockstep, and removed edges in light colors. (a) *Graft* (Line 2). All edges $e = \langle u, v\rangle$ are checked in parallel. If u and v belong to different pseudo-trees $\Lambda(r)$ and $\Lambda(r')$, respectively, a merge is performed by grafting $\Lambda(r)$ onto $\Lambda(r')$, thus forming a larger pseudo-tree. In Figure 1, Iteration 1 of grafting results in two pseudo-trees rooted at 1 and 2, while Iteration 2 further merges them into one. (b) *Pointer jump* (Line 3). The parent pointers for all vertices are updated in parallel. For each vertex v in a pseudo-tree $\Lambda(r)$, the parent pointer $p(v)$ is updated so that $p(v) = r$. This ensures that all vertices in a pseudo-tree point directly to the root (see Figure 1). (c) *Contract* (Line 4). All edges are checked in parallel, to remove ones internal to a pseudo-tree.

Algorithm $\mathcal{A}$ continues to iterate through these locksteps until no edges remain in G (Line 1). At this point, it returns the number of distinct pseudo-tree roots as WCC of G (Line 5). □

As shown above, compared to VC/EC programs, PRAM algorithms (1) support beyond-neighborhood direct memory accesses, not restricting the operations within the neighborhood of each vertex; (2) exploit shared memory for synchronization, not via message passing; and (3) feature inherent load balancing among processors, without skewed workloads over power-law graphs [37]. Moreover,

Algorithm 2: Planar program for WCC.

Status Declaration: $S_V = \{\bar{p}\}$ where $p(v) = v$ for each $v \in V$;
StatusAggr: $(p(v), p'(v)) \Rightarrow \min\{p(v), p'(v)\}$.

Function PEval (subgraph $F_i = (V_i, E_i, L_i)$) :
1 **while** E_i not empty **do**
2 EApply($(\forall e \in E_i) \Rightarrow$ Graft(e));
3 VApply($(\forall v \in V_i) \Rightarrow$ PointerJump(v));
4 EApply($(\forall e \in E_i) \Rightarrow$ Contract(e));
5 **return** state R_i on F_i;

Function IncEval (partial result R_i, updates $\Psi[F_i]$) :
6 VApply($(\forall v \in \Psi[F_i](\bar{p})) \Rightarrow p(p(v)) := \Psi[F_i](p(v))$);
7 VApply($(\forall v \in V_i) \Rightarrow$ PointerJump(v));
8 **return** updated partial state R_i;

Function Assemble (partial results $R_1, R_2, \ldots, R_m$) :
9 **return** the number of distinct values in $\bar{p}$;

unlike VC/EC that may require nontrivial efforts in algorithm development, PRAM is backed by a rich set of existing algorithms, which simplifies both programming and debugging.

3 A PARALLEL COMPUTATION MODEL

This section introduces the parallel model of Planar, including how to program (Section 3.1) and execute (Section 3.2) a graph algorithm. We also discuss its benefits over prior parallel models (Section 3.3).

3.1 Programming with Planar

We aim to "plug" existing PRAM algorithms in a single-machine system. However, this is nontrivial. PRAM assumes (a) unlimited memory and a unit access cost, and (b) a polynomial number of cores such that one can process all edges with different cores in parallel. These are beyond the reach of a real-life machine; *e.g.*, when graphs are too large to fit into its memory, we have to use secondary storage as memory extension; with this comes I/O cost.

To close this gap, we propose a parallel model to make practical use of PRAM graph algorithms. We (1) decompose out-of-core computation into in-memory tasks; and (2) run a PRAM algorithm on each in-memory task to leverage multiple cores and shared memory. This is enabled by a simple, high-level programming abstraction.

For a query class Q, the user needs to provide three functions: (1) PEval, an existing batch PRAM algorithm that evaluates queries $Q \in Q$ on a subgraph; (2) IncEval, an existing incremental PRAM algorithm that refines partial results with border updates; and (3) Assemble, an aggregator for final results. To further simplify programming, each of these functions can be implemented using high-level primitives specialized for PRAM, *i.e.*, concurrent data structures and PRAM operators. The handling of partitioned graphs and synchronization is mostly hidden from users; the only addition is the definition of functions to resolve conflicts in border updates.

PEval. Batch function PEval takes as input a query $Q \in Q$ and a subgraph F_i of G ($i \in [1, m]$); it computes the partial result $Q(F_i)$ at *state* $R_i = \mathcal{A}(Q, F_i)$ *on* F_i by PRAM algorithm $\mathcal{A}$. More specifically, PEval initializes *partial status* $S(F_i) = (S_{V_i}, S_{E_i}, S_G)$, where S_{V_i} (resp. S_{E_i}) is the status variables associated with vertices (resp. edges) in F_i. The partial evaluation process evaluates Q over F_i; it returns updated status $S(F_i)$ as part of state R_i, keeping track of the computation. It also extracts *round updates*, which consist of changes to the border vertices/edges and their status. As PEval concludes on subgraph F_i, round updates are aggregated into a global cache Ψ, which resolves the conflicting updates to border entities.

PEval may implement an existing batch PRAM algorithm $\mathcal{A}$ for Q. One only needs to extend it with the following.

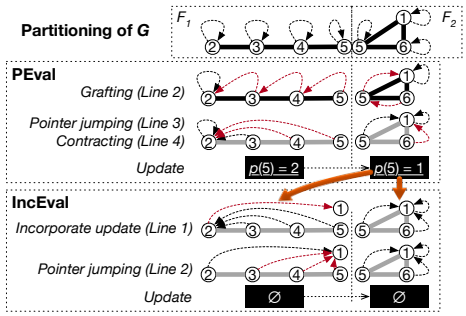


Figure 2: WCC computation over partitioned G with Planar.

(1) Declare status in $\mathcal{A}$. Function PEval declares the status $S(G)$ of G, by making use of concurrent data types. Status $S(G)$ includes

- vertex status S_V (resp. edge status S_E) in an array of length $|V|$ (resp. $|E|$), indexed by vertex (resp. edge) identifiers; and
- global status S_G as variables that are globally accessible.

Given subgraph F_i, PEval initializes *partial status* $S(F_i) = (S_{V_i}, S_{E_i}, S_G)$, where S_{V_i} (resp. S_{E_i}) is the subset of variables in S_V (resp. S_E) that are associated with vertices/edges in F_i. It maintains the *partial state* $R_i = (F_i, S(F_i))$, which is persisted onto secondary storage upon task completion to keep track of the computation.

(2) Specify aggregators. PEval on different subgraphs may make conflicting updates to status variables of border entities. To resolve the conflicts, PEval specifies two aggregation functions: (a) StatusAggr for status variables, and (b) MutateAggr for edge mutations. At the end of PEval, Planar aggregates border updates by applying both.

(3) Implement function body. Function PEval is essentially PRAM algorithm $\mathcal{A}$. It applies a sequence of synchronized parallel operations over F_i and initial status $S(F_i)$ to produce state R_i, where direct and concurrent memory accesses are granted for each operation.

The lockstep operations of $\mathcal{A}$ are directly streamlined as if users were programming sequentially, using PRAM operators:

(a) VApply(f_V): vertex parallel operator, which applies a function f_V to a set of vertices in parallel. Here f_V is the procedure of $\mathcal{A}$ for processing each vertex v. It can access any variables relevant to v in partial status $S(F_i)$, and may mutate the graph (see below).
(b) EApply(f_E): edge parallel operator, similar to VApply.

Both parallel operators are synchronized; we place an implicit synchronization barrier after each invocation of VApply and EApply. Within each parallel operator, all reads to R_i *precede* any write.

To support PRAM algorithms that transform the topological structure, functions f_V and f_E may invoke an additional primitive:

(c) Mutate(e, e'): replace an existing edge e with a new edge e'.

Example 2: Now we show the PEval program for $\mathcal{A}$ of Example 1. As shown in Algorithm 2, PEval (1) declares status along the same lines; (2) defines a StatusAggr function for S_V; and (3) implements the three locksteps using VApply and EApply (Lines 1–4).

Figure 2 depicts the execution of PEval over G. Assume that G is partitioned into subgraphs F_1 and F_2 via vertex-cut (by "cutting" through vertex v_5), such that either can be processed completely in memory. Planar first processes F_1, produces a single pseudo-tree rooted at v_2, and generates an update in a set Ψ indicating v_5's parent to be v_2; it then processes F_2 along the same line, and aggregates updates to $p(v_5)$ by taking the minimum. It finishes with $\Psi = \{p_\Psi(v_5) : v_1\}$ and subgraphs as pseudo-trees of height 1 (indicated by $\bar{p}$); function IncEval will later take these as input. □

IncEval. The incremental function takes as input query Q, subgraph F_i, stale partial state R_i and relevant subset $\Psi[F_i]$ of updates. It updates the partial state incrementally in-place via $R_i = \mathcal{A}(Q, R_i \oplus \Psi[F_i])$, where $R_i \oplus \Psi[F_i]$ denotes integration of $\Psi[F_i]$ with R_i; it adopts incremental evaluation so as to make maximum reuse of the last-round computation. In the end, Planar resets stale values in Ψ to a clean slate for the next round of IncEval execution.

Function IncEval implements an incremental PRAM algorithm $\mathcal{A}_\Delta$ for query class $\boldsymbol{Q}$, sharing the status declaration and aggregators of PEval. We may deduce $\mathcal{A}_\Delta$ from $\mathcal{A}$ following [28] such that $\mathcal{A}_\Delta$ guarantees correctness and minimal incrementalization cost.

Example 3: Continuing with Example 2, IncEval implements an incrementalized $\mathcal{A}$ for Planar, where each subgraph in the partial results of the last round consists of only a set of pseudo-trees with height 1. As shown in Algorithm 2, IncEval works on F_i as follows: (1) it incorporates aggregated updates in $\Psi[F_i]$ in parallel, such that for each border vertex v in F_i, its aggregated status update $p_\Psi(v)$ overrides $p(p(v))$, the parent of v's parent (Line 6); and (2) it conducts another round of parallel pointer jumping (Line 7), such that the resulting subgraphs remain a set of pseudo-trees with height 1.

Figure 2 also illustrates the execution of IncEval. Working over F_1, update $\{p_\Psi(v_5) : v_1\}$ sets $p(v_2) = v_1$; vertex v_1 then becomes a border vertex shared by F_1 and F_2. Then, pointer jumping changes the parent of all vertices in F_1 to v_1. IncEval generates no further update to S_V; thus, it triggers Assemble upon completion. □

Assemble takes partial state R_i ($i \in [1, m]$) and partitions $\mathcal{F}$ as input, and combines R_i to get the final answer $Q(G)$. It is triggered when IncEval makes changes to neither F_i nor $S(F_i)$ ($i \in [1, m]$).

Example 4: Assemble in Algorithm 2 simply counts distinct roots of all pseudo-trees in $\bar{p}$, following Line 5 of Algorithm 1. □

3.2 Parallel Model

Given a query $Q \in \boldsymbol{Q}$ and a graph G, the parallel model coordinates the execution of PEval, IncEval and Assemble, no matter whether the computation fits into the memory of a single machine or not.

Out-of-core computation. If graph G exceeds the memory capacity, Planar partitions G into m subgraphs $\mathcal{F} = (F_1, F_2, \ldots, F_m)$ such that each F_i and its status $S(F_i)$, are small enough to be processed in-memory. While Planar may use any graph partitioners $\mathcal{P}$ [8, 15, 25, 37, 49, 53], we will develop an "optimal" one in Section 4.

Planar executes algorithm $\mathcal{A}$ by decomposing computation over large G into manageable, in-memory PRAM tasks over subgraphs in $\mathcal{F}$. The tasks are organized in iterative rounds, synchronizing via the shared memory. Each PRAM task is executed one at a time without overlapping computation with others, using all available CPU cores at once. Following the principle of GC [29], the process of task decomposition and synchronization is made transparent.

In-memory computation. This is a special case under the parallel model. When G and its status $S(G)$ for $\mathcal{A}$ fit entirely into memory, Planar can work with partition $\mathcal{F} = (G)$ directly. It has a single PRAM task, by simulating $\mathcal{A}$ over G with all available cores.

Task decomposition. For out-of-core computation, Planar decomposes it into in-memory PRAM tasks over subgraphs, as follows.

Iterative evaluation. Given a query $Q \in \boldsymbol{Q}$, Planar works iteratively towards query result $Q(G)$, carrying out computation over each subgraph. To simplify the discussion, we adopt the BSP model [87], which separates computation in supersteps (rounds). A round starts with each subgraph being evaluated locally, and concludes with a global synchronization step, where border updates of all subgraphs are aggregated. For $t \geq 1$, a new round $t+1$ cannot start until round t has completed; the updates generated in round t are accessible only in round $t+1$. This ensures that the computation across the entire graph stays synchronized until a fixpoint is reached.

The iterative process has three phases: partial evaluation (PEval), incremental computation (IncEval), and termination (Assemble). Each phase takes a different PRAM function, as follows.

(1) Partial evaluation. The first round computes partial result $Q(F_i)$ for each subgraph $F_i \in \mathcal{F}$ by executing function PEval in parallel using all available cores. It also extracts *round updates*, which consist of changes to the border vertices/edges and their status.

(2) Incremental computation. Starting from the second round, Planar iteratively carries out one or more incremental evaluation rounds over each subgraph $F_i \in \mathcal{F}$, by IncEval. Intuitively, an IncEval round maintains the partial result at each subgraph, by refining it incrementally in response to border updates of the last round.

(3) Termination. If an IncEval round ends up with no change to status variables, Planar triggers function Assemble, which aggregates partial results of all subgraphs and returns query answer $Q(G)$.

Each round. In each round, Planar iterates through all subgraphs and executes one of the three functions, with all available cores. It processes one subgraph in memory at a time without overlapping computations of multiple subgraphs, by executing PRAM on a physical machine with p cores with a size-p thread pool. That is, a round involves (at most) m "independent" tasks to cope with limited memory and CPU cores. Moreover, Planar loads and processes each subgraph together with its associated status, overlapping computation with I/O to improve CPU and I/O bandwidth utilization. With a subgraph under processing, it preloads the next into memory for buffering and persists the previous one (with the partial state) onto disk (see [2] for more details). This effectively creates a checkpoint for the computation, providing some resilience for failure recovery.

Fixpoint. The out-of-core execution of Planar can be modeled as a fixpoint computation over partition $(F_1, F_2, \ldots, F_m)$ of G:

$$\begin{array}{lcll} R_i^0 & = & (F_i^0, S(F_i^0)) & = \ \mathsf{PEval}(Q, F_i^0), \\ R_i^{t+1} & = & (F_i^{t+1}, S(F_i^{t+1})) & = \ \mathsf{IncEval}(Q, R_i^t \oplus \Delta S(F_i^t)). \end{array}$$

For $i \in [1, m]$, t denotes a round; F_i^0 is subgraph F_i; F_i^t is subgraph F_i at the end of round t with status $S(F_i^{t+1})$; R_i^t denotes partial results from F_i^t in round t; and $\Delta S(F_i^t)$ denotes status updates to F_i generated in round t. The process iterates until it reaches round $\hat{t}$ such that $R_i^{\hat{t}+1} = R_i^{\hat{t}}$ for all $i \in [1, m]$. At this point, $\mathsf{Assemble}(R_1^{\hat{t}}, \ldots, R_m^{\hat{t}})$ computes and returns $Q(G)$. The fixpoint computation starts with PEval and takes IncEval as its intermediate consequence operator.

Similar to the proof of [29], one can verify that Planar supports all graph computations, which are covered by PRAM. Moreover, we show that the fixpoint computation guarantees to converge at $Q(G)$ if (a) PEval and IncEval are *contracting*, *i.e.,* the values of all status variables are from a finite active domain and updated monotonically following a partial order in each round; and (b) PEval, IncEval and Assemble are *correct* PRAM algorithms, *i.e.,* at round $t+1$ when partial state $R_i^{t+1} = R_i^{\hat{t}}$ for all $i \in [1, m]$, Assemble over

$R_1^t, \ldots, R_m^t$ produces $Q(G)$ deterministically. Intuitively, the former ensures the termination of the fixpoint computation, while the latter warrants the correctness of the final result. An assurance theorem and proof are deferred to [2]. It extends [29] by supporting multi-core parallelism for each in-memory subgraph via SIMD of PRAM.

3.3 Planar vs. VC/EC and GC

Taking WCC as an example, we compare the parallel model of Planar with prior models. We also develop Planar programs for SSSP, PR, Coloring, MST and RW; each implements a well-established prior PRAM algorithm for the query class, preserving the correctness and efficiency while requiring little modification. Their analyses are consistent (deferred to [2] for the lack of space).

Parallel scalability. We adapt the parallel scalability of [56] to characterize the effectiveness of parallel algorithms. Consider a sequential algorithm $\mathcal{A}'$ for a query class $\mathcal{Q}$, which takes $t_{\mathcal{A}'}(|Q|, |G|)$ time to answer a query $Q \in \mathcal{Q}$ over graph G; and a parallel algorithm $\mathcal{A}$ that takes $t_{\mathcal{A}}(|Q|, |G|, p)$ time using p cores. The *speedup* of $\mathcal{A}$ over sequential $\mathcal{A}'$ is $s(|Q|, |G|, p) = t_{\mathcal{A}'}(|Q|, |G|)/t_{\mathcal{A}}(|Q|, |G|, p)$. We say that $\mathcal{A}$ is *parallelly scalable relative to* $\mathcal{A}'$ if for any Q and G, $s(|Q|, |G|, p)/p \geq \epsilon$ for some constant $\epsilon > 0$. Intuitively, it guarantees speedup of $\mathcal{A}$ relative to a "yardstick" sequential $\mathcal{A}'$. In principle, such $\mathcal{A}$ is able to reduce the cost of $\mathcal{A}'$ with more cores.

Comparison with VC/EC. We start with VC/EC.

Example 5: A common VC/EC algorithm for WCC is HashMin [96]. Given graph $G = (V, E)$, it assigns each vertex a unique ID, and propagates the lowest ID across each connected component via iterative message passing through edges. It takes $O((|V| + |E|)D)$ time when G fits in memory, where D denotes the diameter of G.

In contrast, Planar does $O((|V|+|E|) \log D)$ amount of work [82] (see Examples 2–4). This is because $\mathcal{A}$ shrinks D by half after each round via topological mutations, reducing message propagation. Neither $\mathcal{A}$ nor HashMin is parallelly scalable relative to sequential BFS, as both incur polylog amount of redundant work. This said, Planar guarantees linear speedup for up to $|V| + |E|$ cores [82], but VC does not due to contention over high-degree "hubs" nodes.

For large G with partition $\mathcal{F} = (F_1, \ldots, F_m)$, Planar takes at most $\lceil \log \min\{m, D\} \rceil$ rounds, with beyond-neighborhood computation of GC and contracting subgraphs. In contrast, HashMin takes D rounds in the worst case, incurring much more I/O. □

Benefits. The parallel model of Planar makes a better fit to single-machine graph processing than VC/EC, as demonstrated by Example 5 and analyses with other common graph algorithms [2].

For in-memory workloads, Planar may do less work than VC/EC by taking a more efficient PRAM algorithm $\mathcal{A}$; moreover, if $\mathcal{A}$ has proven parallelly scalable, Planar retains the property. In contrast, VC/EC does not guarantee this due to the complexity of message passing: it aggregates messages at all vertices, where high-degree "hubs" become inevitable stragglers and reduce parallel speedup.

Moreover, Planar supports direct beyond-neighborhood data accesses, flexible control flows to skip unnecessary computation, and graph topology mutations. These reduce redundant disk I/O.

Comparison with GC. Designed for multi-machine systems, GC [29] itself is not a good fit for a single-machine system, because it (a) does not support intra-subgraph parallelism and (b) requires expensive message passing for synchronization. As remarked in Section 1, the parallel model of Planar extends GC to a shared-memory multi-core architecture by supporting (1) SIMD intra-subgraph parallelism, (2) out-of-core computation and (3) memory-based synchronization and graph partitioning/scheduling.

4 PARTITIONING AND SCHEDULING

This section develops a graph partitioning and scheduling strategy for Planar. We start with unique challenges introduced by single-machine systems (Section 4.1). We then formalize partitioning and scheduling as an optimization problem and show its intractability (Section 4.2), followed by our strategy for the problem (Section 4.3). We focus on out-of-core Planar computations in this section.

4.1 Challenges

To process a graph G that exceeds the memory capacity of a single machine, Planar partitions G into a set $\mathcal{F}$ of subgraphs. Most conventional partitioning strategies are designed for multi-machine systems. They strive to minimize the replication factor and balancing ratio [18, 25, 29, 37, 66], reducing the communication cost, redundant work and stragglers. On the contrary, Planar synchronizes via the shared memory for which the communication cost is negligible; moreover, it serializes subgraph processing such that workload skewness can hardly slow down computation. This said, single-machine systems introduce the following unique challenges.

Dynamic behavior. The runtime behavior of a Planar program may vary substantially across rounds. Recall that a PEval round loads each subgraph F_i from disk, executes PRAM algorithm $\mathcal{A}$ and produces the partial state R_i; an IncEval round reads R_i of the last round, runs incremental $\mathcal{A}_\Delta$, updates R_i and persists it on disk. For CPU computation, IncEval executes an algorithm different from PEval, whose cost depends heavily on the border updates from the last round. For I/O, IncEval loads partial result R_i, whose size depends on the mutated topological structure of F_i. Moreover, the runtime may change substantially in different IncEval rounds. Given this, how should we partition graph G for the best performance?

In contrast, this is not an issue for multi-machine systems.

Scheduling for CPU- vs. I/O-bound computation. Planar works out-of-core by repeatedly swapping subgraphs in and out of memory, and incurs disk I/O throughout the program execution. More specifically, *a round of execution* can be CPU-bound or I/O-bound. It is *CPU-bound* if its total computational cost over all subgraphs is higher than the total I/O cost; in this case, we should prevent I/O operations from blocking computation. Otherwise, it is an *I/O-bound* round, *i.e.,* the I/O dominates the execution cost, for which we should maximize the I/O bandwidth utilization. Both cases require that we schedule the subgraph processing to maximally overlap the CPU and I/O operations, whichever the bottleneck is.

Scheduling is not an issue for a multi-machine system, which (1) amortizes the I/O cost by loading each subgraph just once; and (2) processes all subgraphs at the same time with different machines.

Granularity. On the one hand, subgraphs should be large enough to improve locality, reduce synchronization and redundant computation. On the other hand, too large subgraphs could take too much memory for computation, leaving little buffering space for overlapped I/O operations. Can we strike a balance in granularity to min-

imize overhead under the constraint of limited memory capacity?

This tradeoff is not studied by prior partitioners, since multi-machine systems assume sufficient memory for each worker and partition the graph based on the number of available machines.

These challenges demand a joint optimization effort in both partitioning and scheduling, taking into account the overlapping of CPU and I/O operations as well as the memory constraint.

4.2 Partitioning and Scheduling Problem

Based on a general cost model for a Planar program, we formalize the partitioning and scheduling problem and show its intractability.

Cost model. Consider a partition $\mathcal{F}$ of G. We formulate the round cost in terms of the computational and I/O costs for each F_i.

Round cost. For round j, denote by $C_{\mathcal{A}_j}(F_i, p)$ the computational cost over F_i using p processors, and by $\mathsf{IO}(F_i)$ the I/O cost, which is proportional to the size of the partial state R_i, Assuming that Planar processes subgraphs $\mathcal{F} = (F_1, \ldots, F_m)$ in order and overlaps CPU and I/O whenever possible, the round cost is

$$C_j(\mathcal{F}) = \mathsf{IO}(F_1) + \Sigma_{i=2}^{m} \max\{\mathsf{IO}(F_i), C_{\mathcal{A}_j}(F_{i-1}, p)\} + C_{\mathcal{A}_j}(F_m, p). \quad (1)$$

Intuitively, this model accounts for the sequential processing of subgraphs and the overlapping of I/O and computation in a pipeline. While Planar computes on the current subgraph, it simultaneously loads the next one; the longer duration between the two determines the subgraph cost. The round cost is thus the sum of all subgraphs.

Peak memory usage. When processing the partial state R_i of subgraph F_i at runtime, let $M_{\mathcal{A}}(F_i)$ denote its peak memory usage. This can be estimated as a function $M_{\mathcal{A}}(F_i) = \mu_{\mathcal{A}}(|R_i|)$, where $\mu_{\mathcal{A}}$ is determined by the space complexity of algorithm $\mathcal{A}$.

Profiling. Once a Planar program is compiled, we feed in some graphs as profiling tests, to (1) train a binary classification profiler to determine *qualitatively* whether the PEval round is CPU- or I/O-bound on the given machine; and (2) train function $\mu_{\mathcal{A}}(|R_i|)$ as a regression model. The training samples include runtime parameters *e.g.,* subgraph sizes, degree skewness, and border updates.

The test inputs are small in size, of the same type as the real input. In our experiments we used 4 graphs from 0.14–30.14GB in size, and the entire profiling procedure takes 96s. The profiler can accurately predict the bottleneck in >95% cases.

Problem statement. Consider a Planar program $\mathcal{A}$. Given input graph G, p cores and memory capacity B, we formulate the joint partitioning and scheduling problem as an optimization problem to find a partition $\mathcal{F} = (F_1, \ldots, F_m)$ of G, such that if subgraphs are processed in order, the round cost is minimized and memory capacity B is never exhausted during execution. The objective is

$$\arg\min_{\mathcal{F}} C_j(\mathcal{F}),$$
$$\text{subject to} \quad M_{\mathcal{A}}(F_i) + M_{\mathcal{A}}(F_{i+1}) \le B, i \in [1, m-1].$$

Its decision problem, denoted by DPSP, is to decide, given Planar program $\mathcal{A}$, graph G, integer m, memory bound B, and cost threshold η, whether there exists a valid partition $\mathcal{F}$ of G such that if subgraphs are processed in order, the round cost is at most η.

Theorem 1: DPSP *is* NP*-hard.* □

Proof sketch: We show that DPSP is NP-hard for both I/O-bound and CPU-bound $\mathcal{A}$ (see [2]), by reduction from the NP-complete 3-partition problem (cf. [32]). Given a set A of positive integers, we construct a graph G, an integer m, two positive numbers B and η, algorithm $\mathcal{A}$ (with memory usage $M_A(F_i)$ and round cost $C_j(\mathcal{F})$), such that the set A can be partitioned into disjoint subsets A_1, A_2 and A_3 of equal sum iff there exists a partition $\mathcal{F}$ of G with m subgraphs that meets both the memory bound B and the cost threshold η. □

4.3 Partitioning and Scheduling Strategies

Theorem 1 suggests that even with accurate cost estimations, an optimal partitioning and scheduling strategy still remains intractable. Hence, we seek an efficient heuristic method that intuitively optimizes performance based on observed workload characteristics.

Overview. We adopt a joint partitioning and scheduling strategy. The idea is to (1) partition G speculatively into small "blocks" during preprocessing, (2) group blocks into subgraphs that fit in memory at runtime, and (3) schedule adaptively based on the bottleneck. In other words, by processing graphs in manageable blocks, Planar efficiently adjusts partitioning/scheduling based on real-time measurements, circumventing the need for precise cost estimations.

At preprocessing, Planar decomposes G into a collection of small blocks, to reduce the runtime decision space and the complexity of grouping. It uses a new locality-aware branching technique, which is optimized for connectivity among blocks and locality within each block. Intuitively, both metrics benefit subgraph-based processing by facilitating update propagation and reducing border status.

To group blocks at runtime, Planar follows the principle of a state-of-the-art partitioner to minimize the replication factor. It boosts the locality within each grouped subgraph by reducing border entities. As opposed to working with "static" subgraphs, Planar (1) makes block grouping decisions adaptively at runtime to cope with its dynamic behavior; (2) adopts different scheduling strategies for CPU- and I/O-bound rounds to accommodate the bottleneck; and (3) adjusts granularity based on runtime memory usage and strikes a balance between the "on-stage" and "off-stage" working memory.

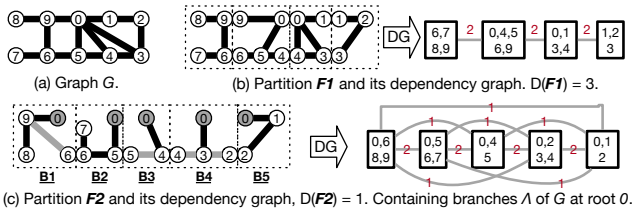
Speculative partitioning. We first develop a speculative partitioner that produces small blocks $\mathcal{F}$ of G. To simplify the discussion, we adopt vertex-cut; it can be adapted to edge-cut or hybrid.

Motivation. Based on Equation 1, we identify key design objectives:

- *Connectivity among subgraphs.* Stronger connectivity boosts update propagation across the entire graph, which leads to faster convergence and fewer rounds in fixpoint computation.
- *Locality within a subgraph.* Better locality can (a) reduce the total size of partial states and hence the I/O cost; and (b) lower the IncEval round complexity. This is in principle consistent with minimizing the replication factor for multi-machine partitioners.

To these ends, we develop a two-stage partitioning algorithm. In the first stage, we model the connectivity among blocks by introducing a notion of *dependency graph* DG. We propose a *branch decomposition* technique, which produces a collection $\mathcal{F}$ of blocks and minimizes the diameter of $\mathrm{DG}(\mathcal{F})$. The second stage performs *greedy adjustment*, which refines blocks by redistributing edges at the border and merges blocks that are too small in size.

Dependency graph models the connectivity among blocks (subgraphs). Such a graph *w.r.t.* $\mathcal{F} = (F_1, \ldots, F_m)$ is an undirected graph $\mathrm{DG}(\mathcal{F}) = (V_{\mathrm{DG}}, E_{\mathrm{DG}}, L_{\mathrm{DG}})$. It has m vertices, where $v_i \in V_{\mathrm{DG}}$ denotes F_i for each $i \in [1, m]$. An edge e_{ij} exists between v_i and v_j



(a) Graph G. (b) Partition $\mathcal{F}1$ and its dependency graph. $D(\mathcal{F}1) = 3$.

(c) Partition $\mathcal{F}2$ and its dependency graph, $D(\mathcal{F}2) = 1$. Containing branches Λ of G at root 0.

Figure 3: Partitions, branches and dependency graphs.

if F_i and F_j share some entities; its weight $L_{\text{DG}}(e_{ij})$ denotes the number of shared entities. We will see how m is determined shortly.

Denote by $D(\mathcal{F})$ the diameter of $\text{DG}(\mathcal{F})$. Intuitively, the smaller $D(\mathcal{F})$ is, the stronger the connectivity is. It takes fewer steps to propagate an update throughout a small-diameter graph, benefiting label-setting algorithms [50], *e.g.,* WCC, SSSP and Coloring.

Example 6: Figures 3b and 3c show two ways of partitioning graph G (Figure 3a). Partition $\mathcal{F}_1$ has a dependency graph whose diameter is 3; $\mathcal{F}_2$ has $D(\mathcal{F}_2) = 1$ despite that it has 1 more subgraph. □

(1) Branch decomposition employs a greedy algorithm, denoted by Decompose. It produces an arbitrary number of blocks. The idea is to cut the graph through a high-degree vertex r, boost the connectivity and minimize $D(\mathcal{F})$. It puts a few high-degree vertices on the border, and strives to keep others within a block. We will further reduce border entities via greedy adjustment and block grouping.

Consider *w.l.o.g.* connected $G = (V, E)$. As shown in Algorithm 3, after finding the highest-degree *root* vertex v_r in V, Decompose breaks $V \setminus \{v_r\}$ into a disjoint set Λ of branches with procedure SortBFSBranch (line 2). A branch λ in Λ is a set of vertices on the same branch in a BFS tree rooted at v, and can thus be reconstructed into a block via procedure Expand, by adding edges that are incident to its vertices. BFS traversal balances the height of each branch; this limits the diameter of each expanded block and reduces within-subgraph computation. The algorithm recurses if the expanded subgraph of a branch is too large to make a valid partition (lines 3–4).

Example 7: Continuing with Example 6, Decompose breaks G into five branches (Figure 3c). The edges in light color connect separate branches; they may be subject to adjustment later. □

(2) Greedy adjustment. We next adjust $\mathcal{F}$ by (a) redistributing border entities, and (b) merging blocks that are too small in size. It produces blocks whose number m is arbitrary yet more manageable.

With redistribution, our goal is to reduce border entities. More specifically, we find each edge e incident to a non-root border vertex v, and migrate e to another block with v tentatively. The changes are materialized if the total border entities are reduced.

Merging aims to (a) limit the number of blocks in $\mathcal{F}$ to reduce the complexity of grouping, and (b) avoid small, scattered I/O requests and improve disk bandwidth utilization. We will merge blocks whose estimated memory usage is below a threshold. The threshold is decided based on the storage type; by default over a SATA SSD, we set it to 256MB for full disk bandwidth utilization [85].

Remark. Speculative partitioning introduces a one-time preprocessing cost, higher than hash-based *e.g.,*VCut [98] and ECut [61]. Nevertheless, as we will show in Section 5, the cost can be amortized over a few rounds of computation. Moreover, we implement incremental partitioning [27] to cope with dynamic graphs.

Block grouping. Consider a speculative partition $\mathcal{F}$. Grouping selects a set of pending blocks in $\mathcal{F}$ at runtime and processes them as a single subgraph. Given a memory budget τ, we aim to find a grouping that (1) promotes locality within a group, and (2) requires memory usage at most τ. Note that the groupings are temporary; each subgraph is still persisted as a separate file after computation.

Algorithm 3: Function Decompose.

Input: A connected graph G, memory budget τ.
Output: Vertex-cut subgraphs $\mathcal{F}$ of G.

1 **if** $M_{\mathcal{A}}(G) \le B$ **then return** $\{G\}$; **else** init $\mathcal{F} := \emptyset$;
2 find max-degree vertex v_r in G; $\Lambda :=$ SortBFSBranch(G, v_r);
3 **while** Λ has non-empty head λ **and** $M_{\mathcal{A}}(\text{Expand}(\lambda)) > \tau$ **do**
4 $\quad \mathcal{F} := \mathcal{F} \cup \text{BranchGroup}(\text{Expand}(\lambda))$; remove λ from Λ;
5 **return** $\mathcal{F}$;

Procedure Expand (λ):
6 $\quad$ **return** $F = (V(\lambda), E_F)$, where $E_F = \{e \in G \mid e.\text{src} \in V(\lambda)\}$;

Procedure SortBFSBranch (G, v):
7 $\quad$ tree $T := \text{BFS}(G, v)$; get T's branches $\Lambda := \{\lambda_1, \ldots, \lambda_{|N(v)|}\}$;
8 $\quad$ **return** sorted Λ, decreasingly in $M_{\mathcal{A}}(\text{Expand}(\lambda)), \forall \lambda \in \Lambda$;

At a high level, this is equivalent to partitioning the dependency graph $\text{DG}(\mathcal{F})$ via edge-cut, so as to minimize the total weight of border edges. To this end, we propose an algorithm called Grouping. It adapts the *neighbor expansion* heuristic of [98] to weighted dependency graphs, which guarantees an optimal replication factor.

More specifically, Grouping selects a set $\tilde{F}$ of blocks, starting with a random vertex v_1 in $\text{DG}(\mathcal{F})$. It works iteratively until the memory budget τ is exhausted; each iteration adds one block (*i.e.,* a vertex in $\text{DG}(\mathcal{F})$) to $\tilde{F}$. In the i-th iteration, it finds v_i greedily based on

$$v_i = \arg \max_{v \in \mathcal{F} \setminus \tilde{F}} \Sigma_{j=1}^{i-1} L_{\text{DG}}(\langle v, v_j \rangle). \quad (2)$$

Intuitively, Grouping greedily adds blocks to $\tilde{F}$, to maximally hide border entities inside a group as "internal" entities. It is efficient. In our experiments over large graph `clueWeb` (see Table 1), $\mathcal{F}$ has 328 blocks, and each grouping iteration takes a negligible ≤30 ms.

Example 8: Suppose that Planar takes 5-block $\mathcal{F}_2$ (Figure 3c) as input, with a memory budget τ of two blocks. If we start with block B1, the grouping strategy will group it with B2 and process both as a single subgraph, since the two share the most border entities. □

Adaptive scheduling. To overlap CPU and I/O operations, we split the working memory into the off-stage area for buffering pending blocks, and the on-stage area for computation. With the strategies above, one question remains open: how can we balance the split, by setting a memory budget τ for off-stage area?

We use different strategies for CPU- and I/O-bound rounds. At each round, the profiler predicts the upcoming bottleneck utilizing the statistics of subgraphs and runtime parameters of previous rounds (if any). This prediction initializes the scheduling strategy, which is adapted to real-time measurements as the round proceeds.

CPU-bound rounds. The goal is to ensure that I/O operations do not block CPU computation. Thus, we attempt to keep CPUs busy at all times, and may allow some gaps in block loading.

To prevent CPUs from starving, we adopt greedy strategies. (1) With an empty off-stage area, we always load the smallest pending block. (2) When the current on-stage area concludes computation, we immediately group all blocks in the off-stage area and move them to the on-stage area for computation. (3) For the off-stage area, we reserve an *upper bound* τ as half of the memory capacity B, to allow sufficient buffering for the next group of blocks. Intuitively, to improve CPU utilization in early rounds, we start with fine-grained groupings; to speed up computation, we fine-tune the granularity.

Table 1: Graph datasets.

Name	Type	$\|V\|$	$\|E\|$	Mean Distance	Data Size (GB)
friendster [1]	social network	65.6M	1.8B	5.1	28.9
web-sk [77]	Web	50M	1.9B	13.7	32.0
datagen [42]	synthetic	29M	2.6B	12.5	80.7
clueWeb [77]	Web	1.7B	7.9B	65.7	140.6
hyper12 [4]	Web	273M	9B	42.8	143.0

Too small τ often leads to under-utilization of the working memory, while too large τ may result in oscillations in grouping sizes.

I/O-bound rounds. Our objective is to ensure computation not to block I/O. We strive to keep loading from the disk, and may tolerate idling CPUs. To this end, we attempt to load as many blocks as possible and process them as a group, until the aggregate memory usage of the group exceeds budget τ. In contrast to CPU-bound rounds, we set τ dynamically to maximize block grouping and memory utilization. More specifically, we start with a *lower bound* $\tau = B/2$, and gradually increase τ as long as the computation in the on-stage area concludes before the off-stage area is saturated.

Example 9: Continuing with Example 8, suppose that Planar has a working memory that can hold 4 blocks. If the round is CPU-bound, it will start computation with the smallest block B5, during which B3 and B4 will be buffered in the off-stage area and processed with grouping, so on and so forth. If the round is I/O-bound, it will start a group of two blocks before commencing computation. □

5 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we evaluated Planar for its (1) efficiency, (2) partitioning and scheduling strategy, (3) (parallel) scalability, and (4) performance vs. multi-machine systems.

Experimental setting. We start with the setups.

Datasets. We used five real-life and synthetic datasets, as described in Table 1. All graphs have been widely used in prior work, allowing us to have a comparison under similar conditions. Among these, datagen, clueWeb and hyper12 are among the few open-source graphs that cannot fit in the memory of our main testbed. Here datagen is a synthesized dataset in the LDBC [42] benchmark suite; hyper12 is a BFS sample [59] of hyperlink [4]. The two smaller graphs, friendster and web-sk, are similar in size but have different distributions; they can reveal interesting insights. Datasets were formatted and partitioned based on each system's requirements; no partitioning is required for in-memory workloads.

To study the impact of graph characteristics on system performance, we generated a series of synthetic graphs using gMark [10], of type bibliography and uniprot. Each graph has 100M vertices, with an average degree ranging from 5–15, *i.e.*, 0.5B–1.5B edges.

Baselines. We evaluated four out-of-core systems: VC-based GraphChi [57], EC-based GridGraph [105] and Blaze [52], and MiniGraph [106] with a hybrid model. We omitted Mosaic [65] for its now-discontinued Intel Xeon Phi coprocessor [70], and CLIP [6] because it produces inconsistent results for multiple graph queries.

We tested SOTA in-memory systems Galois [72] and Ligra [83] for (parallel) scalability, and omitted CoroGraph [102] because it cannot handle graph with >4.3B edges. We also tested GPU systems Subway [79] and CGGraph [19], FPGA system AccuGraph [97], and multi-machine systems Gluon [22] and GraphScope [24]. All systems were tested in default configurations.

We also tested four variants of Planar: (1) $\text{Planar}_{\text{static}}$, which disables block grouping (see Section 4.3); (2) $\text{Planar}_{\text{rand}}$, which groups blocks randomly, not based on the heuristic of Equation 2; (3) $\text{Planar}_{\text{par}}$, by allowing concurrent subgraphs processing; (4) $\text{Planar}_{\text{persist}}$, which persists border updates on disks at each round, and (5) $\text{Planar}_{\text{par+persist}}$, which enables both mechanism (3) and (4).

Table 2: Runtime statistics. Each round in Blaze is an EC superstep.

Query	Dataset	Metric	Planar	MiniGraph	Blaze	$\text{Planar}_{\text{static}}$	$\text{Planar}_{\text{rand}}$
WCC	web-sk	Time (s)	**56.9**	237.3 (4.17×)	66.0 (1.16×)	137.8 (2.42×)	67.9(1.19×)
		# Rounds	**2**	6	24	10	2
		I/O (GB)	**7.4**	104.0 (14.05×)	28.5 (3.85×)	55.1 (7.45×)	7.4 (1.00×)
	friend-ster	Time (s)	**54.7**	130.3 (2.38×)	91.0 (1.66×)	142.6 (2.60×)	58.4 (1.07×)
		# Rounds	**2**	3	16	3	2
		I/O (GB)	**7.0**	54.5 (7.79×)	27.0 (3.86×)	26.1 (3.73×)	7.0 (1.00×)
SSSP	web-sk	Time (s)	**19.2**	366.5 (19.09×)	107.7 (5.61×)	156.7 (8.16×)	27.6 (1.44×)
		# Rounds	**2**	18	58	17	3
		I/O (GB)	**11.0**	201.1 (18.28×)	40.0 (3.64×)	85.8 (7.80×)	20.0 (1.82×)
	friend-ster	Time (s)	**23.1**	183.8 (7.96×)	96.8 (4.19×)	99.8 (4.32×)	30.3 (1.31×)
		# Rounds	**2**	8	31	8	3
		I/O (GB)	**7.0**	86.0 (12.29×)	36.4 (5.20×)	100.2 (14.31×)	12.0 (1.71×)

We evaluated four alternative partitioning strategies (Exp-2): (1) VCut, a state-of-the-art vertex-cut heuristic [98]. (2) ECut of [61], the edge-cut used by MiniGraph [106]. (3) 2DVCut, a vertex-cut partitioner used by GridGraph [105], Mosaic [65], and GraphScale [21]. (4) 1DVCut, the vertex-cut used by HitGraph [103]. They produce the same number m of blocks as our speculative partitioner does.

Algorithms. We evaluated Planar programs for WCC, PR, Coloring, SSSP, MST and RW (see [2] for implementation details), common graph queries included in various benchmarks [3, 11]. Among these, PR and Coloring are representative algorithms cast from VC/EC; the others have the best known asymptotic complexity for the query. For baselines, we used their out-of-box implementations if available. Since GridGraph does not support Coloring or MST out-of-box, we implemented their VC algorithms [33, 73].

Moreover, we tested various subgraph queries over bibliography, counting k-stars (*i.e.*, the number of papers with at least k authors) and k-hop paths (for paper impact analysis), where $k \in \{3, 4, 5\}$.

For SSSP, we randomly picked 10 vertices and used them as sources for each input graph. For RW, we initiated a walker at every vertex; each walker takes a 5-step walk.

We have validated the correctness and consistency of system outputs. We present the average of each experiment over 5 repetitions. We report results over some graphs; the other results are consistent.

Testbeds. Our main testbed is a workstation with a consumer-grade CPU and limited memory. It is powered by an Intel Core i9-7900X@3.30GHz CPU, with 13.75MB LLC and 20 cores, and 64GB of DDR4-2666 memory. The graphs were loaded from a 1TB WD Blue WDS100T2B0A SATA SSD, which has an average sequential read throughput of 560MB/s. To further test the parallel scalability for in-memory workloads (Exp-3), we used an enterprise-grade server with 512GB of DDR4-2933 memory and 4× Intel Xeon Gold 5320@2.20GHz CPUs, each with 39MB LLC and 26 cores. Unless noted otherwise, all system were tested with default configurations.

We evaluated GPU systems on our server testbed with an NVIDIA V100 GPU with 32GB GPU memory. FPGA systems were evaluated on a Xilinx Alveo U50 card, with 32GB of HBM memory.

Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluated the efficiency and I/O of Planar versus out-of-core baselines for various queries. Over two small graphs, we imposed a 16GB memory budget using cgroups to study the out-of-core behavior. Table 2 reports the runtime statistics of some algorithms compared with MiniGraph and Blaze, the best

Table 3: Out-of-core system performance (in seconds). GraphChi could not finish within 5 hours for all queries over clueWeb and hyper12.

Query	datagen					clueWeb				hyper12			
	Planar	MiniGraph	Blaze	GridGraph	GraphChi	Planar	MiniGraph	Blaze	GridGraph	Planar	MiniGraph	Blaze	GridGraph
WCC	**60.9**	85.8 (1.41×)	81.6 (1.34×)	104.4 (1.71×)	2688.6 (44.15×)	**465.6**	4219.4 (9.06×)	670.9 (1.44×)	7755.3 (16.66×)	**203.2**	7825.0 (38.51×)	340.5 (1.68×)	>5h (>88.58×)
SSSP	**29.1**	42.3 (1.45×)	55.2 (1.90×)	146.7 (5.04×)	2483.4 (85.34×)	**229.7**	1062.9 (4.63×)	506.8 (2.21×)	1415.9 (6.16×)	**91.9**	2348.9 (25.56×)	115.1 (1.25×)	>5h (>195.87×)
PR	**61.0**	100.7 (1.65×)	200.6 (3.29×)	185.6 (3.04×)	646.6 (10.60×)	**529.3**	2131.8 (4.03×)	623.8 (1.18×)	2537.2 (4.79×)	**213.6**	1175.6 (5.50×)	406.6 (1.90×)	1820.2 (8.52×)
Coloring	**153.7**	620.9 (4.04×)	190.4 (1.24×)	2105.3 (13.70×)	4381.5 (28.51×)	**346.6**	1719.1 (4.96×)	576.6 (1.66×)	2238.5 (6.46×)	**218.6**	15470.5 (70.77×)	346.1 (1.58×)	>5h (>82.34×)
MST	**71.6**	147.2 (2.06×)	84.8 (1.18×)	118.9 (1.66×)	907.7 (12.68×)	**390.6**	>5h (>46.08×)	958.9 (2.45×)	>5h (>46.08×)	**252.4**	>5h (>71.32×)	407.0 (1.61×)	>5h (>71.32×)
RW	**19.6**	174.1 (8.88×)	99.8 (5.09×)	207.1 (10.57×)	3177.2 (162.10×)	**64.0**	1941.4 (30.33×)	297.4 (4.65×)	8391.5 (131.12×)	**59.6**	2067.2 (34.68×)	114.5 (1.92×)	>5h (>302.01×)

performing baselines supporting GC and VC/EC, respectively. Over large graphs, Table 3 reports the performance of all systems.

WCC. From Tables 2–3, we can see the following.

(1) On synthetic datagen (Table-3), Planar beats the four baselines by 1.34–44.15×. Over real-life clueWeb and hyper12, it outperforms MiniGraph by 9.06× and 38.51×, Blaze by 1.44× and 1.68× and GridGraph by 16.66× and >88.58×, respectively, while GraphChi cannot handle large graphs at this scale. Note that Planar is 2.29× faster on hyper12 than on clueWeb, a graph with fewer edges, since hyper12 has much fewer vertices (only 16.1% of clueWeb), allowing more graph contractions during PEval and faster IncEval rounds.

(2) Over two small graphs, Planar is 2.38–4.17× faster than MiniGraph, and 1.16–1.66× faster than Blaze (Table 2). Over web-sk, Planar only takes 2 rounds due to its beyond-neighborhood computation, and its partitioning strategy that promotes connectivity and hence reduces computation rounds. In contrast, MiniGraph and Blaze take 6 and 24 rounds (*i.e.,* supersteps), respectively. Despite the difference in the skeweness of two graphs, Planar has a relative consistent performance; other systems vary greatly.

Here we omit the results of GridGraph and GraphChi; they take at least 3.06× longer than Planar over various workloads.

(3) Planar substantially reduces I/O. On web-sk and friendster (Table 2), its disk read is 90.9% and 74.1% less than MiniGraph and Blaze on average, respectively. Besides taking fewer rounds, Planar reduces I/O in IncEval rounds (Figure 4a), because (a) it contracts the graph by removing most edges in PEval and incurring little disk read in the following rounds over clueWeb; and (b) it reduces I/O further by skipping "inactive" subgraphs. This justifies speculative partitioning, in which a small subgraph is more likely to become inactive.

SSSP. As shown in Tables 2–3, Planar outperforms the all four competitors for SSSP over all graphs. (1) On the two large Web graphs, it is 4.63–25.56×, 1.25–2.21× and 6.16–195.87× faster than MiniGraph, Blaze and GridGraph, respectively. It does the job within 4 min, while GraphChi does not finish in 5 hours. (2) On the synthetic datagen, it beats the four baselines 1.45×–85.34×. (3) Planar takes fewer rounds and 91.9% less disk read than MiniGraph on friendster, leading to a 7.96× speedup. The I/O reductions are not as significant as with WCC, since Planar does not contract the graph during SSSP computation. The speedup is greater (19.09×) on web-sk. These verify that Planar is more effective on large-diameter graphs, since its partitioner strives to improve subgraph connectivity. It leads to faster convergence for GC computation; Planar takes only 2 rounds on both graphs, while MiniGraph takes 8–18 rounds. (4) Over friendster (resp. web-sk), Blaze generates 5.20× (resp. 3.64×) of the disk I/O of Planar and takes 4.19× (resp. 5.61×) longer, even though it leverages on-demand, fine-grained (4KB) I/O to reduce disk reads. This highlights the effectiveness of beyond-neighborhood computation in Planar, which substantially reduces redundant computation, leading to 93.5–96.6% fewer rounds.

PR. For PR over all large graphs, Planar beats the baselines consistently despite that they all execute the same algorithm. As shown in Table 3, it outperforms MiniGraph, Blaze, GridGraph, and GraphChi by at least 1.65×, 1.18×, 3.04× and 10.60×, respectively. This is because Planar maximally overlaps computation and I/O, optimized for shared-memory, multi-core concurrent data accesses.

Coloring. As shown in Table 3 and Figure 4b, (1) Planar is over 1.24× faster, takes at least 97% fewer rounds, and generates 51.1% less disk read, compared to EC/VC-based Blaze and GridGraph. The improvements stem from its beyond-neighborhood computation, which reduces redundant coloring fixes. (2) Planar beats MiniGraph by ≥ 4.04×, even though they both follow the same principle of GC and execute the same algorithm. Figure 4b reveals two reasons for this: (a) Planar takes 69.2% less I/O in the first round, benefiting from the compact storage format (see Section D); and (b) it reduces more disk read in later rounds by skipping processing of more blocks.

MST and RW. As shown in Table 3, Planar consistently beats the four baselines for MST and RW. (1) For MST, Planar is at least 2.06×, 1.18×, 1.66× and 12.68× faster than MiniGraph, Blaze, GridGraph, and GraphChi, respectively. This is due mainly to its more efficient PRAM algorithm, which employs the graph-contraction mutations, like WCC. (2) For RW, Planar beats the best performing baseline by 1.92–5.09×, since Planar supports the random walk algorithm of [46], which is more efficient than that of VC/EC and Hybrid.

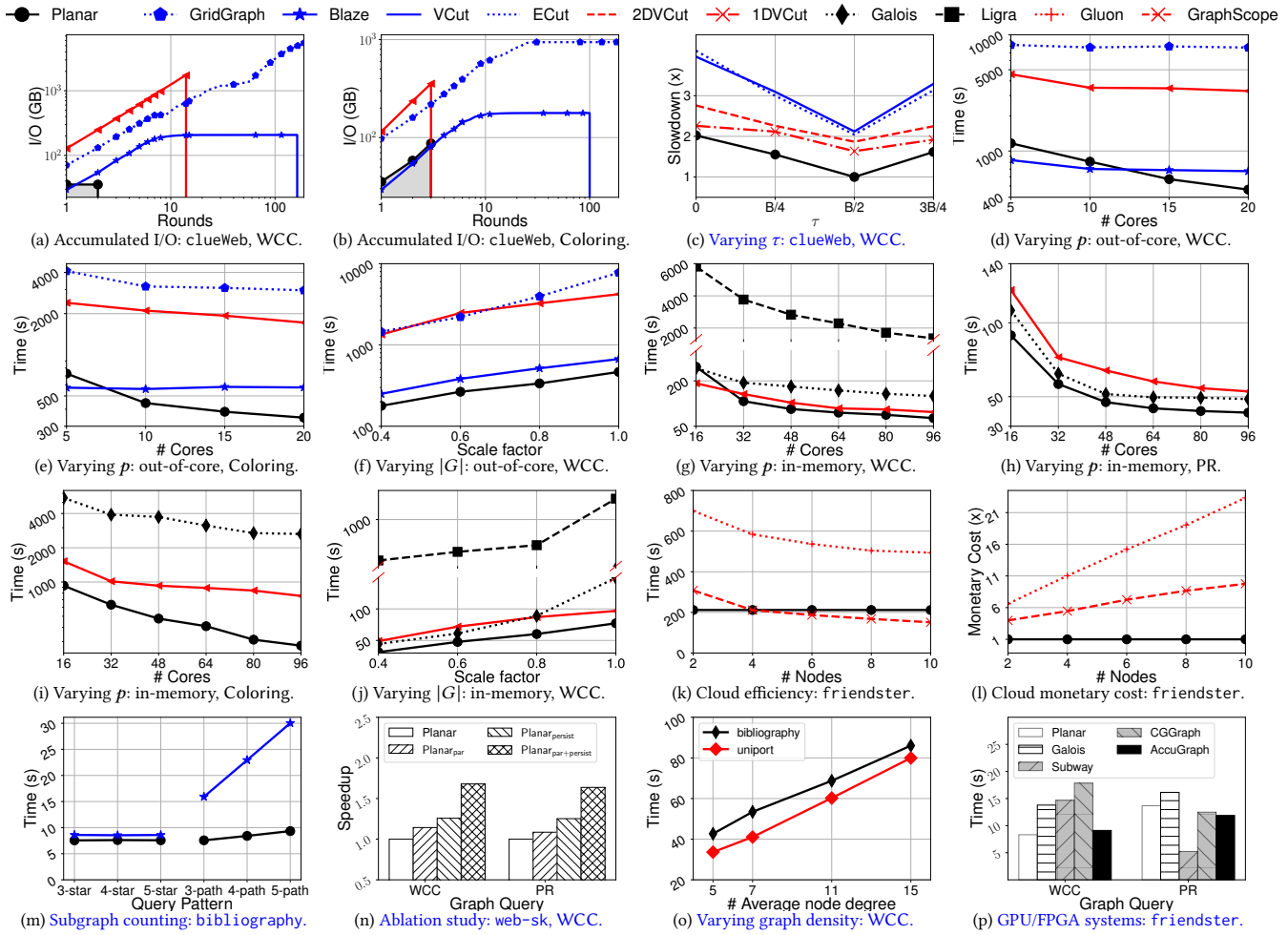
Subgraph counting. As shown in Figure 4m over bibliography, Planar answers all subgraph queries in 10s. For various star-counting queries, its performance is relatively consistent, beating Blaze by 1.13× on average. For path counting, Planar runs faster over simpler patterns, as expected. Its speedup over Blaze is 2.10–3.22×.

Exp-2: Ablation study. Next we tested the effectiveness of our partitioning and scheduling strategy, as well as other design choices.

Varying τ. We varied the memory τ reserved for I/O buffering. As shown in Figure 4c for WCC, setting $\tau = 0$ or $\tau = B$ effectively disables the overlapping of CPU and I/O operations, causing substantial slowdown. With $\tau = B/2$, Planar performs the best regardless of the partitioner, hence the V-shape of all lines. This justifies our adaptive scheduling strategy (Section 4.3).

Impact of partitioners. We tested Planar with different partitioners on clueWeb. As shown in Figure 4c for WCC with $\tau = B/2$, our partitioner beats the alternatives consistently. It speeds up VCut, ECut, 2DVCut, 1DVCut by 2.12×, 2.04×, 1.87× and 1.63×, respectively.

On the server testbed, Planar takes 51.9 min for preprocessing, while the other partitioners take 11.0–16.7 min. This is because speculative partitioning involves local graph traversals and greedy adjustments, which are more computationally intensive than the edge bucketing in other methods. This said, it is an one-time cost amortized over a few rounds of computation. Taking preprocessing into account, Planar beats MiniGraph after answering 1 WCC or 2 Coloring queries. This demonstrates that the partitioning strategy



(a) Accumulated I/O: clueWeb, WCC. (b) Accumulated I/O: clueWeb, Coloring. (c) Varying τ: clueWeb, WCC. (d) Varying p: out-of-core, WCC.

(e) Varying p: out-of-core, Coloring. (f) Varying $|G|$: out-of-core, WCC. (g) Varying p: in-memory, WCC. (h) Varying p: in-memory, PR.

(i) Varying p: in-memory, Coloring. (j) Varying $|G|$: in-memory, WCC. (k) Cloud efficiency: friendster. (l) Cloud monetary cost: friendster.

(m) Subgraph counting: bibliography. (n) Ablation study: web-sk, WCC. (o) Varying graph density: WCC. (p) GPU/FPGA systems: friendster.

Figure 4: Efficiency, scalability, partitioning and scheduling of Planar, and its performance vs. multi-machine systems.

of Planar is effective and shows a quick return on investment.

Effectiveness of scheduling. Both grouped and ordered processing of subgraphs make Planar faster. In our experiments, the trained profiler can accurately predict the round bottleneck in >95% cases; it is 100% accurate for WCC and SSSP. Consider SSSP in Table 2. (1) On friendster (resp. web-sk), $\text{Planar}_{\text{static}}$ took 4× (resp. 8.5×) more rounds, generated 14.31× (resp. 7.80×) more disk read and became 4.32× (resp. 8.16×) slower. This is because adaptive grouped processing elicits faster convergence, reducing redundant computation of repetitive border updates and allowing to skip some subgraphs in later rounds. (2) Planar beats $\text{Planar}_{\text{rand}}$ by 1.37× on average. This justifies our ordering on subgraphs, which promotes the locality for subgraph grouping. The results for the others are consistent.

Other techniques. We experimentally justified our design decisions to process subgraphs sequentially and cache border updates in-memory. Figure 4n shows that if we were to allow multiple subgraphs to be processed concurrently, Planar would take 1.09–1.14× longer; if we were to persist the updates to disk, it would be slowed by 1.26×. The two collectively can speedup Planar by 1.64–1.68×.

Exp-3: Scalability. For both out-of-core and in-memory computation, we evaluated the scalability of Planar with the number p of CPU cores and the graph size $|G|$. We report the results of WCC and Coloring; the results of the other queries are consistent.

Varying p: out-of-core. Scaling the number p of cores, we ran WCC (Figure 4d) and Coloring (Figure 4e) over clueWeb. (1) For WCC, Planar scales well with p. It gets a 2.50× speedup when p scales from 5 to 20 because its PEval round is CPU-bound. (2) It consistently beats MiniGraph and GridGraph, which barely scale with p due to the I/O-bound rounds. (3) It outperforms Blaze only when $p >$ 10, because its PRAM algorithm is more computation-heavy than HashMin [96] (by a constant factor, see Example 5). However, the latter has limited parallelism; Planar speeds up substantially with more cores, while Blaze barely improves when $p \geq 10$ since it cannot fully utilize the additional cores. (4) For Coloring, Planar is much faster than all baselines for $p > 5$ for similar reasons. It gets 2.10× faster with 4× cores, while other systems speedup <1.64×. Its I/O-efficient GC rounds can benefit more from higher parallelism.

Varying $|G|$: out-of-core. We sampled graphs G from large clueWeb using Edge Sampling [59], with a scale factor η for the fraction of edges to be sampled. As shown in Figure 4f when varying η from 0.4 to 1.0 for WCC, Planar scales well with $|G|$. It takes 2.61× longer, while it is 3.14× for MiniGraph, 2.68× for Blaze, and 5.33× for GridGraph. We omit the results of GraphChi, which is much slower.

Varying p: in-memory. Varying p from 16 to 96, Figures 4g–4i report the speedup of Planar and in-memory baselines. (1) For WCC, Planar is up to 1.95× and 28.39× faster than Galois and Ligra, re-

spectively. (2) It is 28.3% slower than MiniGraph when $p = 16$; yet it beats MiniGraph by 1.18×–1.26× with more cores. Consistent with the out-of-core tests (Figure 4d), Planar scales better than MiniGraph with cores. (3) Using 6× more cores for WCC (resp. PR), it gets faster by 3.20× (resp. 2.33×), while it is only 1.98× (resp. 2.28×) for MiniGraph, 1.61× (resp. 2.24×) for Galois. Ligra scales as well as Planar, yet it is slower by magnitudes (omitted in Figure 4h). (4) For parallelly scalable Coloring, Planar speeds up by 3.36× with 6× more cores, much better than MiniGraph (2.02×) and Galois (2.07×). This verifies that when the PRAM algorithm is parallelly scalable, it retains the property for in-memory computations.

Varying $|G|$: in-memory. Using all 104 cores, we further tested in-memory systems by varying the size $|G|$ of graph G. As shown in Figure 4j for WCC, by varying sampling factor η from 0.4 to 1.0 over `clueWeb`, Planar scales better with $|G|$ than VC-based Ligra and Galois; it takes 2.46× longer, while it is 3.37× for Galois and 4.59× for Ligra. This justifies the parallel model of Planar, which supports PRAM algorithms with better asymptotic complexity *w.r.t.* $|G|$. Its scalability is comparable to MiniGraph (1.98×), since both implement efficient algorithms and can scale well with the graph size.

Sensitivity to graph topology. Over synthetic graphs for WCC, Figure 4o shows the performance of Planar under varying graph densities and types. It takes longer for denser graphs, as expected, scaling almost linearly with the number of edges. Given a similar graph size, it performs better over uniprot than more skewed bibliography.

For Coloring and PR, Planar beats the other baselines in all the settings. The results are consistent and deferred to [2].

Exp-4: Planar vs. hardware-accelerated systems. As shown in Figure 4p for WCC over `friendster`, in-memory Planar is 1.10–2.14× faster than SOTA systems with GPU [19, 79] and FPGA [97] accelerators. However, it is slower by 1.09–2.61× for PR, an algorithm with a more regular data access pattern and thus can better utilize the massive hardware concurrency available to GPUs/FPGAs. Nevertheless, Planar is much more cost-effective, using a server testbed that costs less than half of a GPU/FPGA accelerator alone.

Moreover, GPU/FPGA cannot natively handle large graphs that exceed the on-chip memory capacity. We show that the parallel model of Planar can help address such a limitation. By extending CGGraph [19] to use Planar for coordinating computation over subgraphs, it can process 140GB `clueWeb` using a 32GB-memory GPU. It becomes 1.13× and 2.44× slower than Planar for WCC and PR, respectively, due to the excessive host–GPU data transfers.

Exp-5: Planar vs. multi-machine systems. We also evaluated the performance and cost effectiveness of Planar versus multi-machine systems GraphScope and Gluon. We deployed all systems in the cloud. More specifically, we ran Planar on a single 8-vCPU 32GB-memory instance. Gluon used instances of the same configuration; GraphScope used multiple 8-vCPU 64GB-memory instances because it requires more than 32 GB in all cases of our experiment.

Resource demands. Single-node Planar supports WCC computation over large graphs; in contrast, multi-machine systems easily run out-of-memory. Over `clueWeb`, for example, GraphScope (resp. Gluon) required at least eight 64GB-memory (resp. four 32GB-memory) nodes. This verifies that Planar lowers the bar of big graph analytics, where large memory capacities are no longer a necessity.

Execution time. As shown in Figure 4k for WCC on `friendster`, (1) using a single instance, Planar outperforms a 10-node Gluon cluster by 2.33×, and performs comparably to a 4-node GraphScope cluster. While GraphScope beats Planar when using 6+ instances (each with 2× memory capacity), it requires an additional 200+s for preprocessing, which is not counted in Figure 4k. (2) None of the multi-machine system scales well. Scaling from 4 to 10 machines (using 2.5× cores), Gluon and GraphScope run 1.2× and 1.4× faster, respectively, as the communication cost gradually dominates. In contrast, the more cores are available, the better Planar works.

Cost effectiveness. For WCC on `friendster`, Figure 4l shows the monetary cost of Planar and multi-machine systems, calculated as the real cloud expense. (1) Although GraphScope and Gluon run faster with more resources, they do not scale cost-effectively, as also observed by [68]. (2) The cost of GraphScope is 5.45× that of Planar for a similar performance; Gluon, running much slower, spends at least 6.61× more. This further justifies the need for a single-machine system to make graph analytics accessible and affordable.

Application. Planar boosts real-world graph analytics for its cost effectiveness. Consider a navigation system for vehicles, to find the shortest paths between user-specified start and destination in a large route graph. With limited in-vehicle computing resources, Planar offers an ideal solution, by speeding up route planning and improving user experience without requiring high-end hardware.

Summary. We find the following. (1) Planar consistently outperforms the SOTA single-machine systems. It beats out-of-core MiniGraph, Blaze, GridGraph and GraphChi by up to 70.77×, 5.09×, 131.12× and 302.01×, respectively; it can handle workload over large graphs that exceed the capacity of all four baselines. It speeds up in-memory Galois and Ligra by up to 9.58× and 28.39×, respectively. (2) Over various out-of-core workloads, it reduces the I/O cost of the baselines by up to 94.5%. (3) Its partitioner beats prior ones by at least 1.87×, up to 2.12×, and its scheduling strategy consistently speeds up performance. (4) It scales well with large graphs that do not fit in memory. For in-memory computation, on average it is 3.36× faster when using 6× cores for parallelly scalable PRAM algorithms. (5) It requires less memory and is consistently faster than a 10-node Gluon cluster; it performs comparably to GraphScope with 4 machines, saving the monetary cost by 81.7%.

6 CONCLUSION

The novelty of Planar includes the following. (1) Planar is the first graph system that makes practical use of existing PRAM algorithms. (2) It proposes a parallel model that unifies in-memory and out-of-core graph computations, which goes beyond simple simulation of PRAM; it separates inter-subgraph data-partitioned parallelism from intra-subgraph SIMD parallelism to utilize multi-core parallelism, a novel combination. (3) It studies a new graph partitioning/scheduling problem, and develops a strategy that solves the unique challenges not met in multi-machine systems. Our experimental study has validated that Planar is promising in practice.

One topic for future work is to equip Planar with GPU to speed up analytics. Another topic is to fine-tune Planar for a designated task, *e.g.*, graph cleaning, for its best performance.

REFERENCES

[1] 2024. Friendster dataset. https://snap.stanford.edu/data/com-Friendster.html.

[2] 2024. Full version, with source code availability. https://shuhaoliu.github.io/assets/papers/planar-full.pdf.

[3] 2024. Graph500 benchmark specifications. https://graph500.org/?page_id=12#sec-3.

[4] 2024. Hyperlink. http://webdatacommons.org/hyperlinkgraph/.

[5] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. 2018. Passing Messages while Sharing Memory. In *PODC*. 51–60.

[6] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *USENIX ATC*.

[7] Helmut Alt, Torben Hagerup, Kurt Mehlhorn, and Franco P Preparata. 1987. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM J. Comput.* 16, 5 (1987), 808–835.

[8] Konstantin Andreev and Harald Racke. 2006. Balanced graph partitioning. *Theory of Computing Systems* 39, 6 (2006), 929–939.

[9] Dmitrii Avdiukhin, Sergey Pupyrev, and Grigory Yaroslavtsev. 2019. Multi-Dimensional Balanced Graph Partitioning via Projected Gradient Descent. *PVLDB* 12, 8 (2019), 906–919.

[10] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat. 2016. gMark: Schema-driven generation of graphs and queries. *TKDE* 29, 4 (2016), 856–869.

[11] Scott Beamer. 2016. *Understanding and improving graph algorithm performance*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

[12] Naama Ben-David, Guy E Blelloch, Yihan Sun, and Yuanhao Wei. 2019. Multiversion concurrency with bounded delay and precise garbage collection. In *SPAA*. 241–252.

[13] Claude Berge and Ghouila-Houri. 1965. *Programming, games and transportation networks*. John Wiley, New York.

[14] Charles-Edmond Bichot and Patrick Siarry. 2013. *Graph partitioning*. John Wiley & Sons.

[15] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *SIGKDD*.

[16] Sergey Brin and Lawrence Page. 2012. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks* 56, 18 (2012), 3825–3833.

[17] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. Recent Advances in Graph Partitioning. In *Algorithm Engineering - Selected Results and Surveys*. 117–158.

[18] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing* 5, 3 (2019), 13.

[19] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *PVLDB* 17, 6 (2024), 1405–1417.

[20] Dong Dai, Wei Zhang, and Yong Chen. 2017. IOGP: An Incremental Online Graph Partitioning Algorithm for Distributed Graph Databases. In *HPDC*. 219–230.

[21] Jonas Dann, Daniel Ritter, and Holger Fröning. 2024. GraphScale: Scalable Processing on FPGAs for HBM and Large Graphs. *ACM Trans. Reconfigurable Technol. Syst.* 17, 2, Article 22 (mar 2024), 23 pages. https://doi.org/10.1145/3616497

[22] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *PLDI*.

[23] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V. Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. 2019. Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics. In *PACT*. 15–28.

[24] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Kai Zeng, Kun Zhao, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: A Unified Engine For Big Graph Processing. *PVLDB* 14, 12 (2021), 2879–2892.

[25] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Xiaojian Luo, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2020. Application Driven Graph Partitioning. In *SIGMOD*.

[26] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph Pattern Matching: From Intractability to Polynomial Time. *PVLDB* 3, 1-2 (2010), 264–275.

[27] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of graph partitioning algorithms. *PVLDB* 13, 8 (2020), 1261–1274.

[28] Wenfei Fan, Chao Tian, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2021. Incrementalizing graph algorithms. In *SIGMOD*. 459–471.

[29] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, XiaoJian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing Sequential Graph Computations. *TODS* 43, 18 (2018).

[30] Steven Fortune and James Wyllie. 1978. Parallelism in random access machines. In *STOC*. 114–118.

[31] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* (1987), 596–615.

[32] Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

[33] Assefaw Hadish Gebremedhin and Fredrik Manne. 2000. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience* 12, 12 (2000), 1131–1146.

[34] Fady Ghanim, Uzi Vishkin, and Rajeev Barua. 2017. Easy PRAM-based high-performance parallel programming with ICE. *TPDS* 29, 2 (2017), 377–390.

[35] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *PVLDB* 13, 8 (2020), 1304–1318.

[36] Leslie M Goldschlager. 1978. A unified approach to models of synchronous parallel machines. In *STOC*. 89–94.

[37] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX OSDI*.

[38] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. 1995. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press.

[39] Elaine T. Hale, Wotao Yin, and Yin Zhang. 2008. Fixed-Point Continuation for ℓ_1-Minimization: Methodology and Convergence. *SIAM Journal on Optimization* 19, 3 (2008), 1107–1130.

[40] Minyang Han and Khuzaima Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *PVLDB* 8, 9 (2015), 950–961.

[41] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*. 1223–1231.

[42] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. 2016. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB* 9, 13 (2016), 1317–1328.

[43] Nilesh Jain, Guangdeng Liao, and Theodore L Willke. 2013. Graphbuilder: Scalable graph ETL framework. *Graph Data Management Experiences and Systems* (2013).

[44] Joseph JáJá. 1992. *An introduction to parallel algorithms*. Addison-Wesley.

[45] Lokesh N. Jaliminche, Chandranil Nil Chakraborttii, Changho Choi, and Heiner Litz. 2023. Enabling Multi-tenancy on SSDs with Accurate IO Interference Modeling. In *SoCC*.

[46] David R Karger, Noam Nisan, and Michal Parnas. 1992. Fast connected components algorithms for the EREW PRAM. In *SPAA*. 373–381.

[47] George Karypis and Vipin Kumar. 1995. *METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0*. Technical Report. University of Minnesota.

[48] George Karypis and Vipin Kumar. 1998. METIS: A software package for partitioning unstructured graphs. *Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version* 4 (1998).

[49] George Karypis and Vipin Kumar. 1998. Multilevel k-way partitioning scheme for irregular Graphs. *JPDC* 48, 1 (1998), 96–129.

[50] Arijit Khan. 2017. Vertex-Centric Graph Processing: Good, Bad, and the Ugly. In *EDBT*. 438–441.

[51] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/2600212.2600227

[52] Juno Kim and Steven Swanson. 2022. Blaze: Fast graph processing on fast SSDs. In *SC*. 1–15.

[53] Mijung Kim and K Selçuk Candan. 2012. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data&Knowledge Engineering* 72 (2012), 285–303.

[54] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. In *ICLR*.

[55] Robert Krauthgamer, Joseph Naor, and Roy Schwartz. 2009. Partitioning graphs into balanced components. In *SODA*.

[56] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. 1990. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science* 71, 1 (1990), 95–132.

[57] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *USENIX OSDI*.

[58] Bryant C. Lee, Uzi Vishkin, and George C. Caragea. 2007. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. In *Handbook of Parallel Computing - Models, Algorithms and Applications*. Chapman and Hall/CRC.

[59] Jure Leskovec and Christos Faloutsos. 2006. Sampling from large graphs. In

SIGKDD.

[60] Dongsheng Li, Yiming Zhang, Jinyan Wang, and Kian-Lee Tan. 2019. TopoX: Topology Refactorization for Efficient Graph Partitioning and Processing. *PVLDB* 12, 8 (2019), 891–905.

[61] Yifan Li. 2017. *Edge partitioning of large graphs.* Ph.D. Dissertation. Université Pierre et Marie Curie-Paris VI.

[62] Hang Liu and H Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *FAST*. 285–300.

[63] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB* 5, 8 (2012), 716–727.

[64] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication. In *USENIX ATC*.

[65] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *EuroSys*.

[66] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *SIGMOD*.

[67] Daniel W. Margo and Margo I. Seltzer. 2015. A Scalable Distributed Graph Partitioner. *PVLDB* 8, 12 (2015), 1478–1489.

[68] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*.

[69] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: A parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[70] Timothy Prickett Morgan. 2018. The End of Xeon Phi - It's Xeon and Maybe GPUs From Here. https://www.nextplatform.com/2018/07/27/end-of-the-line-for-xeon-phi-its-all-xeon-from-here/.

[71] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *SOSP*.

[72] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*.

[73] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. 2018. The distributed minimum spanning tree problem. *Bulletin of EATCS* 2, 125 (2018).

[74] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *CIKM*.

[75] Alex Pothen, Horst D Simon, and Kang-Pu Liou. 1990. Partitioning sparse matrices with eigenvectors of graphs. *SIMAX* 11, 3 (1990), 430–452.

[76] Robert Clay Prim. 1957. Shortest connection networks and some generalizations. *Bell System Technical Journal* 36, 6 (1957), 1389–1401.

[77] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. 4292–4293.

[78] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*.

[79] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Eurosys*.

[80] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. 2013. "All roads lead to Rome": Optimistic recovery for distributed iterative data processing. In *CIKM*.

[81] Tomer Shanny and Adam Morrison. 2022. Occualizer: Optimistic Concurrent Search Trees From Sequential Code. In *USENIX OSDI*.

[82] Yossi Shiloach and Uzi Vishkin. 1982. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms* 3, 1 (1982), 57–67.

[83] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *SIGPLAN*. 135–146.

[84] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2017. *PuLP/XtraPuLP: Partitioning Tools for Extreme-Scale Graphs.* Technical Report. Sandia National Laboratory.

[85] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo I Seltzer. 2011. Benchmarking File System Benchmarking: It IS Rocket Science.. In *HotOS*.

[86] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "think like a vertex" to "think like a graph". *PVLDB* 7, 3 (2013), 193–204.

[87] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *CACM* 33, 8 (1990), 103–111.

[88] Leslie G. Valiant. 1990. General Purpose Parallel Architectures. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. 943–972.

[89] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*.

[90] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. 2013. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*.

[91] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

[92] Dominic JA Welsh and Martin B Powell. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Comput. J.* 10, 1 (1967), 85–86.

[93] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. *TBD* 1, 2 (2015), 49–67.

[94] Xianghao Xu, Fang Wang, Hong Jiang, Yongli Cheng, Dan Feng, and Yongxuan Zhang. 2020. A Hybrid Update Strategy for I/O-Efficient Out-of-Core Graph Processing. *TPDS* 31, 8 (2020), 1767–1782.

[95] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *PVLDB* 7, 14 (2014), 1981–1992.

[96] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. *PVLDB* 7, 14 (2014), 1821–1832.

[97] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. 2018. An efficient graph accelerator with parallel data conflict management. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 12 pages. https://doi.org/10.1145/3243176.3243201

[98] Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. 2017. Graph Edge Partitioning via Neighborhood Heuristic. In *SIGKDD*.

[99] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *PPoPP*. 183–193.

[100] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph DSL. *OOPSLA* 2 (2018), 1–30.

[101] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing Billion-Node graphs on an array of commodity SSDs. In *FAST*. 45–58.

[102] Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, and Minqi Zhou. 2023. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. *PVLDB* 17, 4 (2023), 891–903.

[103] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput Graph Processing Framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2249–2264. https://doi.org/10.1109/TPDS.2019.2910068

[104] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *USENIX OSDI*.

[105] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*.

[106] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. 2023. MiniGraph: Querying big graphs with a single machine. *PVLDB* 16, 9 (2023), 2172–2185.

Table 4: Notations

Notation	Definition
$Q, \mathcal{Q}$	a class of graph queries, a query $Q \in \mathcal{Q}$
$G, \mathcal{A}, \mathcal{P}$	graph, PRAM graph algorithm, graph partitioner
$S(G), (S_V, S_E, S_G)$	status of $\mathcal{A}$ over graph G and its vertex/edge/global status
$R(G) = (G, S(G))$	state of algorithm $\mathcal{A}$ for graph G
F_i, V_i, E_i	the i-th fragment (subgraph) of graph G, its vertex set and edge set
$F_i.B$	border entities (vertices and edges) of F_i
$S(F_i), S(F_i.B)$	status associated with F_i and its border entities, respectively
$R_i, S(F_i)$	partial state of and status associated with F_i
$\Psi, \Psi[F_i]$	key-value store in UpdateCache and its subset *w.r.t.* F_i

A PROGRAMMING WITH PLANAR

We first identify conditions under which the PRAM algorithms plugged into Planar guarantee to converge at correct answers in execution. Then, we provide Planar programs for SSSP, PR, Coloring, MST and RW with corresponding PRAM algorithms.

The notations of this paper are summarized in Table 4.

A.1 Correctness

We first present convergence guarantee for the parallel model of Planar as a sufficient condition. In general, the correctness and the convergence conditions for GC computations [29] can be readily adapted to the corresponding Planar computations.

Recall the fixpoint model for out-of-core Planar computation (Section 3.2). The fixpoint operator of Planar differs from that of GC [29] in the following. (1) Rather than being applied over different subgraphs in parallel, it is evaluated over subgraphs one by one sequentially, where each subgraph is processed at most once in a round. (2) It supports intra-subgraph parallelism on each in-memory subgraph, specified by a sequence of SIMD operations.

Notations. We use the following notations.

(1) Planar program ρ *terminates* with partitioner $\mathcal{P}$ if for all queries $Q \in \mathcal{Q}$ and all graphs G, there always exists a positive integer $\hat{t}$ such that after round $\hat{t}$, $R_i^{\hat{t}} = R_i^{\hat{t}+1}$ for all $i \in [1, m]$.

(2) Planar program ρ with PEval, IncEval and Assemble is *correct for* $\mathcal{Q}$ w.r.t. $\mathcal{P}$, if for all queries $Q \in \mathcal{Q}$ and all graphs G, Assemble over partial states $R_1^t, \ldots, R_m^t$ produces $Q(G)$ at any round t when variables in partial state $R_i^t (i \in [1, m])$ have the same values as in R_t^{t-1}, where $Q(G)$ is the answer to query Q in graph G.

Planar *correctly executes* ρ with $\mathcal{P}$, if for all queries $Q \in \mathcal{Q}$ and all graphs G, ρ terminates with $\mathcal{P}$ and returns $Q(G)$.

(3) We say that PEval and IncEval satisfy the *monotonic condition w.r.t.* partitioner $\mathcal{P}$, if for graphs G and every status variable $x \in S(G)$, (a) the values of x are from a finite set computed from values in the active domain of G (*i.e.,* the constants in G); and (b) there exists a partial order $\leq_{p_x}$ on the values of x such that IncEval decreases the value of variable x in the order of p_x in the computation.

Intuitively, condition (a) above says that variable x draws values from a finite domain, and condition (b) says that x is updated "monotonically" following the partial order p_x in the iterative IncEval rounds. These ensure that Planar program ρ terminates under $\mathcal{P}$.

Below is a condition for the correctness of Planar execution.

Theorem 2: *Consider a* Planar *program ρ for graph query class $\mathcal{Q}$.* Planar *correctly executes ρ with partitioner $\mathcal{P}$ if (a)* PEval *and* IncEval *satisfy the monotonic condition* w.r.t. $\mathcal{P}$*, and (b) ρ with* PEval, IncEval *and* Assemble *is correct for $\mathcal{Q}$* w.r.t. $\mathcal{P}$. □

Table 5: Parallel model of Planar vs. VC/EC. Assuming power-law graph G, whose diameter is D and maximum vertex degree is d.

$\mathcal{Q}$	Model	In-memory		Out-of-core
		Work	Parallel Scalability	# I/O Rounds
WCC	Planar	$O((\lvert V\rvert + \lvert E\rvert) \log D)$ [82]	×	$\lceil \log \min\{m, D\} \rceil$
	VC/EC	$O((\lvert V\rvert + \lvert E\rvert)D)$ [96]	×	D
SSSP	Planar	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$ [69]	✓	$\min\{m, D\}$
	VC/EC	$O(\lvert E\rvert \log \lvert V\rvert)$ [66]	×	D
PR	Planar	$O(\lvert V\rvert + \lvert E\rvert)$ [16]	×	$O(\lvert V\rvert)$
	VC/EC	$O(\lvert V\rvert + \lvert E\rvert)$ [16]	×	$O(\lvert V\rvert)$
Coloring	Planar	$O(d\lvert V\rvert)$ [33]	✓	$\lvert V\rvert$
	VC/EC	$O(d\lvert V\rvert)$ [33]	×	$\lvert V\rvert$
MST	Planar	$O(\lvert V\rvert^2)$ [13]	×	$\lceil \log \min\{m, D\} \rceil$
	VC/EC	$O((\lvert V\rvert + \lvert E\rvert)D)$ [76]	×	D
RW (length-l walks)	Planar	$O(l\lvert V\rvert)$ [46]	✓	$\min\{m, l\}$
	VC/EC	$O(l\lvert V\rvert)$ [66]	✓	l

Here (1) under the monotonic condition, Planar program ρ guarantees to terminate, and (2) it converges at correct answer $Q(G)$ for all queries $Q \in \mathcal{Q}$ and all graphs G as long as the PRAM algorithms PEval, IncEval and Assemble of ρ are correct for query class $\mathcal{Q}$. In other words, condition (a) guarantees termination of ρ, and conditions (a) and (b) put together guarantee the correctness of ρ.

Note that Theorem 2 just exemplifies one set of conditions for the correctness; a variety of other convergence conditions have been studied, *e.g.,* [39, 41, 80, 93], which can also characterize and guarantee the correctness of Planar programs.

Proof. We show the correctness of Theorem 2 by analyzing the computations of a Planar program. Observe the following.

(1) Termination. Under the monotonic condition, we have that $\ldots \leq_{p_x} R_i^{t+1} \leq_{p_x} \ldots \leq_{p_x} R_i^1 \leq_{p_x} R_i^0$ for $i \in [1, m]$. Since the partial states draw values from a finite domain, there must exist t such that $R_i^{t+1} = R_i^t$ for all $i \in [1, m]$. Thus ρ terminates.

(2) Correctness. From the argument above it follows that the algorithm ρ must terminate at some round $\hat{t}$. By condition (b), Assemble$(R_1^{\hat{t}}, \ldots, R_m^{\hat{t}}) = Q(G)$, *i.e., ρ* computes $Q(G)$. □

A.2 Case Studies

We next show how to program with Planar for various graph query classes. We compare the parallel model of Planar with VC/EC *w.r.t.* the amount of work, parallel scalability and I/O. In addition to WCC (Examples 2–4), we show how to program with Planar for SSSP, PR, Coloring, MST, and RW as case studies. The key results are summarized in Table 5.

1. Single-Source Shortest Path. We start with the single-source shortest path (SSSP) problem. Consider a directed graph $G = (V, E, L)$, where label $L(e) \in [0, 1]$ denotes weight for $e \in E$. A path is $\langle u_1, u_2, \ldots, u_{k+1} \rangle$ where $\langle u_i, u_{i+1} \rangle$ is an edge $(1 \leq i \leq k)$; its length is $\Sigma_{i=0}^{k-1} L(\langle u_i, u_{i+1} \rangle)$. For a pair (s, v) of vertices, dist(s, v) denotes the length of the shortest path from s to v; in particular, dist$(s, v) = \infty$ if v is not reachable from s. Given G and a source vertex s, SSSP is to compute dist(s, v) for all vertices $v \in V$.

Outline. Planar takes the PRAM algorithm Δ-stepping of [69] as $\mathcal{A}_{\text{SSSP}}$, which implements a technique known as label correcting. For each vertex v, it iteratively refines the *tentative distance* tent(v) of v from source s via edge relaxations. The relaxations reduce

Algorithm 4: Δ-stepping [69] for SSSP in Planar.

Status Declaration: $S_V = \{\bar{\text{tent}}\}$: $\text{tent}(v) = \infty$, for each $v \in V$;
StatusCR: $(\text{tent}(v), \text{tent}'(v)) \Rightarrow \min\{\text{tent}(v), \text{tent}'(v)\}$.

Function PEval **(** source vertex s, subgraph F_i, parameter Δ **) :**
1 **if** $s \notin V_i$ **then return** $R_i = (F_i, S(F_i))$;
2 init bucket array B; Relax $(s, 0)$; $i := 0$;
3 **while** non-empty bucket exists in B **do**
4 $D := \emptyset$; /* records deleted vertices for the round. */
5 **while** B not empty **do**
6 $D := D \cup B[i]$; copy $C := B[i]$; clear bucket $B[i] := \emptyset$;
7 EApply$((\forall e \in \{e | e.\text{src} \in C, e \in E_i, L_i(e) \leq \Delta\}) \Rightarrow$ Relax$(e.\text{dst}, t(e.\text{src}) + L_i(e)))$; /* relax light edges. */
8 EApply$((\forall e \in \{e | e.\text{src} \in D, e \in E_i, L_i(e) > \Delta\}) \Rightarrow$ Relax$(e.\text{dst}, t(e.\text{src}) + L_i(e)))$; /* relax heavy edges. */
9 $i := i + 1$;
10 **return** $R_i = (F_i, S'(F_i))$, where $S'(F_i)$ has updated $\bar{\text{tent}}$;

Procedure Relax *(*v, tent'*)*:
11 **if** $\text{tent}' < \text{tent}(v)$ **then**
12 relocate v into bucket $B[\lfloor \text{tent}'/\Delta \rfloor]$; $\text{tent}(v) := \text{tent}'$;

$\text{tent}(v)$ when a shorter path from s to v is found.

As shown in Algorithm 4, PEval takes as input a vertex-cut subgraph F_i of G, source s and parameter Δ. It declares vertex status S_V to include $\text{tent}(v) = \infty$ for each v in V_i except $\text{tent}(s) = 0$. For conflicting $\text{tent}(v)$ of a border vertex v, it keeps the shortest in Ψ. It uses an array B of buckets to keep track of vertices that need to be relaxed, where each bucket represents distance range of size Δ from s. It starts with the immediate neighbors of s (line 2), and works iteratively to update $\text{tent}(v)$ of v in the first non-empty bucket (lines 3–9). It deals with "light" edges of weight at most Δ in parallel (lines 5–7), before processing the remainder "heavy" edges (line 8).

IncEval works incrementally by only dealing with vertices that are affected by updates in $\Psi[F_i]$ in the last round. It follows the same lines as PEval, except that it starts with edges outwards of updated border vertices, instead the source s.

When no more distance updates can be made, Assemble takes the latest $\text{tent}(v)$ values of all $v \in V$ as $\text{dist}(s, v)$.

Benefits over VC. Consider a popular VC algorithm [66] for SSSP. In a nutshell, it works by iteratively refining $\text{tent}(v)$ for each vertex v based on the minimum sum of each in-neighbor u's tentative distance and edge weight $L(\langle u, v \rangle)$. Observe the following.

(1) In-memory. On a power-law graph G, Planar does $O(|V| \log |V| + |E|)$ amount of work, lower than $O(|E| \log |V|)$ of VC. Moreover, $\mathcal{A}_{\text{SSSP}}$ is proven parallelly scalable [69] relative to sequential Dijkstra's algorithm [31], yet VC does not guarantee this, since $\mathcal{A}_{\text{SSSP}}$ reduces redundant edge relaxations with its flexible control flow.

(2) Out-of-core. As in the case for WCC, Planar reduces disk I/O by taking fewer rounds than VC when computing for SSSP, in light of its beyond-neighborhood computation.

2. PageRank. We next consider PageRank (PR) [16], a problem that fits VC. PR takes as input a directed graph $G = (V, E)$ (*e.g.,* hyperlinks among Web pages) and a transition probability $\epsilon \in (0, 1)$; it returns the ranking scores of all vertices in G. The ranking scores model the stationary distribution of a Markov process; for each vertex v in V, v's ranking can be computed following

$$r(v) = d \cdot \Sigma_{\{u | \langle u,v \rangle \in E\}} \frac{r(u)}{u.\text{degree}^+} + (1 - d), \quad (3)$$

Algorithm 5: PR in Planar.

Status Declaration: $S_V = \{\bar{r}\}$: $r(v) = \text{rand}()$, for each $v \in V$;
StatusCR: $(r(v), r'(v)) \Rightarrow \text{mean}(r(v), r'(v))$.

Function PEval **(** fragment F_i, parameter $d = 1 - \epsilon$ **) :**
1 $\Pi := \{(v.\text{id}, 0) \mid v \in V_i\}$;
2 EApply $((e \in E_i) \Rightarrow \text{Propagate}(e))$;
3 VApply $((v \in V_i) \Rightarrow \text{UpdateRank}(v))$;
4 **return** $R_i = (F_i, S'(F_i))$, where $S'(F_i)$ contains updated $\bar{r}$;

Function IncEval **(** partial result R_i, updates $\Psi[F_i]$, parameter d **) :**
5 $\Pi := \{(v.\text{id}, 0) | v \in V_i\}$;
6 VApply$((\forall v \in \Psi[F_i](\bar{r})) \Rightarrow r(v) := \Psi[F_i](r(v)))$;
7 repeat Line 2–3 of function PEval; **return** R_i;

Procedure Propagate *(*$\langle u, v \rangle$*)*: /* F_i, Π accessible in global storage. */:
8 $\Pi[v.\text{id}] := \Pi[v.\text{id}] + r(u)/u.\text{degree}^+$;

Procedure UpdateRank *(*u*)*: /* F_i, Π accessible in global storage. */:
9 $r(v) := d \cdot \Pi[v.\text{id}] + \epsilon$;

where $d = 1 - \epsilon$ is the damping factor and $u.\text{degree}^+$ denotes the out-degree of u. This update function is applied iteratively to refine the ranking scores, until the computation approximates a steady state, *i.e.,* the Euclidean distance between the ranks in two consecutive iterations is below a predefined threshold $\delta > 0$.

Outline. Planar takes the PRAM algorithm of [16] as $\mathcal{A}_{\text{PR}}$, as shown in Algorithm 5. In each iteration, algorithm $\mathcal{A}_{\text{PR}}$ recomputes the score of each vertex based on the scores of its neighbors. More specifically, PEval initializes the in-degree of each vertex v in the vertex status S_D. In PEval round, it scans all fragments and accumulates the in-degree of each vertex in parallel. For IncEval round, it initializes the score of each vertex v as $1.0/d(v)$, in the first IncEval round. Then it updates the score of each vertex based on the scores of its neighbors as Pull procedure does. It returns the updated scores of all vertices. For the later IncEval rounds, it updates the scores of border vertices first, and then does the same Pull operation.

(1) PEval and IncEval. As shown in Algorithm 5, Planar for PR declares vertex status S_V, which includes the intermediate ranking scores for all vertices. The ranking $r(v)$ for each vertex v in V is initialized as a random number in the range of $(0, 1)$.

PEval starts by initializing an auxiliary structure Π, an in-memory hashmap that indexes the aggregate ranks from neighbors for each vertex v, *i.e.,* calculating the total "intensity of incoming flows" for the round (Line 1). It then (1) uses EApply to propagate the ranks of vertices along edges (Line 2); and (2) updates $r(v)$ by combining the aggregate message $\Pi[v]$, for each v in V_i (Line 3).

Conflicts may occur in the intermediate ranking scores of a border vertex v; they are resolved by taking the mean ranks of the same vertex among different fragments. After incorporating conflict-resolved rank $r(v)$ for each border vertex v, IncEval follows the same lines of PEval to update all ranks in another iteration.

(2) Assemble commences when IncEval cannot make significant changes, suggesting that a fixpoint has been reached. It concludes the process by returning the ranks associated to all vertices.

Benefits over VC. Both models support algorithm $\mathcal{A}_{\text{PR}}$. However, as shown in Section 5, Planar is faster than VC due to its parallel execution, data organizations and patterns of memory accesses.

(1) In-memory. Planar does the same amount of work as VC does. Moreover, neither the parallel model of Planar nor VC is parallelly

Algorithm 6: Coloring in Planar.

Status Declaration: $S_V = \{\bar{c}\}$: $c(v) = 0$, for each $v \in V$;
StatusCR: $(c(v), c'(v)) \Rightarrow \min(c(v), c'(v))$.

Function PEval **(** fragment F_i **) :**
1 $U := \emptyset$; $\Pi := \{(v, \emptyset) \mid v \in V_i\}$;
2 EApply$((\langle u, v\rangle \in E_i) \Rightarrow U$.add$(\min_{i \in \{u,v\}} c(i))$ if $c(u) = c(v))$;
3 **if** U is empty **then return** R_i;
4 EApply $((\langle u, v\rangle \in E_i, u \in U) \Rightarrow \Pi[v]$.add$(c(u)))$;
5 VApply $((v \in U) \Rightarrow$ set $c(v)$ to the first color not in $\Pi[v])$;
6 **return** $R_i = (F_i, S'(F_i))$, where $S'(F_i)$ contains updated $\bar{c}$;

Function IncEval **(** partial result R_i, updates $\Psi[F_i]$ **) :**
7 VApply$((\forall v \in \Psi[F_i](\bar{c})) \Rightarrow c(v) := \Psi[F_i](c(v)))$;
8 repeat Line 1–6 of function PEval; **return** R_i;

scalable. This is because algorithm $\mathcal{A}_{\mathsf{PR}}$ is inherently defined by message passing and requires an aggregation operation at each vertex, which typically leads to stragglers. More specifically, the aggregation sums up all $d^-(v)$ incoming ranks at each vertex v, where $d^-(v)$ is the in-degree of v. Over a "hub" vertex v with a large $d^-(v)$ (*e.g.,* $d^-(v) \geq 1,000,000$ is common for real-world power-law graphs), the computation can bottleneck the entire PR iteration.

This issue is particularly serious for VC/EC, because the aggregation at a "hub" can only be carried out in a sequential manner. Planar, on the contrary, can further mitigate the stragglers by parallelizing the summation calculation, thanks to its more flexible organization of auxiliary data structures.

(2) Out-of-core. Since Planar and VC execute the same algorithm, they incur the same amount of disk I/O asymptotically. Nevertheless, Planar may still incur less disk I/O than VC, since it only loads "inactive" blocks for processing in each round.

3. Vertex Coloring (Coloring). Given an undirected graph $G = (V, E)$, Coloring is to assign a color to each vertex in V, such that no two neighboring vertices share the same color.

Outline. Planar takes the PRAM algorithm of [33] as $\mathcal{A}_{\mathsf{Color}}$, as shown in Algorithm 6. In each iteration, $\mathcal{A}_{\mathsf{Color}}$ greedily assigns colors to vertices in parallel [92]; it revisits and fixes invalid coloring, *i.e.,* conflicts, in the next iteration. More specifically, PEval executes $\mathcal{A}_{\mathsf{Color}}$ over each vertex-cut subgraph F_i of G. It declares vertex status S_V to include the same initial coloring for each vertex (Line 1). It maintains a vertex set U, which includes "active" vertices with conflicts, and a hashmap Π, which keeps the *forbidden colors* for each vertex. In each iteration, PEval (1) checks all edges in parallel, adding active vertices to U (Line 2); (2) aggregates the forbidden colors of vertices in U (Line 4); and (3) assigns each vertex in U the first color that is not forbidden (Line 5).

Function IncEval repeats PEval over active vertices triggered by the changes to border nodes, until no more conflicts can be found. At this point, Assemble returns the final coloring.

Benefits over VC. While parallel models support algorithm $\mathcal{A}_{\mathsf{Color}}$, they perform differently for the same reasons given above.

(1) In-memory. Planar does the same amount of work as VC does. However, Planar retains the parallel scalability of $\mathcal{A}_{\mathsf{Color}}$, while VC does not. This is because VC performs message aggregation operations over "hub" vertices, which often lead to stragglers. This further shows the benefit of "flat" data accesses of PRAM.

(2) Out-of-core. Since Planar and VC execute the same algorithm, they incur the same amount of disk I/O asymptotically.

Algorithm 7: MST in Planar.

Status Declaration: $S_V = \{\bar{p}, \bar{e}\}$ where $p(v) = v, e(v) =$ null for each $v \in V$, $S_G = \{E_T\}$, where E_T is initialized as an empty edge set;
StatusCR: $(p(v), p'(v)) \Rightarrow \min\{p(v), p'(v)\}$, $(e(v), e'(v)) \Rightarrow \arg\min_{x \in \{e(v), e'(v)\}} L(x)$.

Function PEval **(** fragment F_i **) :**
1 VApply$((\forall v \in V_i) \Rightarrow e(v) := \arg\min_{x \in \{\langle v,u\rangle \mid u \in Nbr(v)\}} L(x))$;
2 **return** $R_i = (F_i, S'(F_i))$, where $S'(F_i)$ contains updated $\bar{e}$;

Function IncEval **(** partial result R_i, updates $\Psi[F_i]$ **) :**
3 VApply$((\forall v \in V_i) \Rightarrow E_T := E_T \cup \{e(v)\}$; Graft$(e(v)))$;
4 VApply$((\forall v \in V_i) \Rightarrow$ PointerJump$(v))$;
5 EApply$((\forall e \in E_i) \Rightarrow$ Contract(e););
6 VApply$((\forall v \in V_i) \Rightarrow e(v) := \arg\min_{x \in \{\langle v,u\rangle \mid u \in Nbr(v)\}} L(x))$;
7 VApply$((\forall v \in \Psi[F_i](\bar{c})) \Rightarrow c(v) := \Psi[F_i](c(v)))$;
8 **return** $R_i = (F_i, S'(F_i))$;

4. Minimum Spanning Tree (MST). Consider a connected, undirected graph $G = (V, E, L)$, whose label $L(e) \in \mathbb{N}^+$ denotes the weight for edge $e \in E$. A *spanning tree* is a subgraph $T = (V, E_T, L)$ where $E_T \subseteq E$ of G such that T is a tree. The weight $w(T)$ of tree T is the sum of all edge weights, *i.e.,* $w(T) = \sum_{e \in E_T} L(e)$. A spanning tree with the minimum possible weight is called a *minimum spanning tree* of G. For such a graph G with multiple connected components, MST is to compute the minimum spanning trees of all connected components in the graph.

Outline. Planar takes a parallel version of Sollin's PRAM algorithm [13] as $\mathcal{A}_{\mathsf{MST}}$. As shown in Algorithm 7, $\mathcal{A}_{\mathsf{MST}}$ is based on an idea similar to that of the WCC algorithm $\mathcal{A}$ presented in Example 1. It starts from a pseudo-forest of G, and iteratively merges two pseudo trees $\Lambda(r)$ and $\Lambda(r')$ that are connected via a set $E_{r,r'}$ of edges, and adds the edge $e \in E_{r,r'}$ with the minimum weight to E_T.

(1) PEval declares the following in S_V: (a) parent $p(v)$ for each vertex v, initialized as itself; and (b) the smallest-weight edge $e(v)$ for each vertex v, initialized as null. The parent pointers in $\bar{p}$ constitute a pseudo-forest of G, a collection of pseudo-trees as used in WCC (see Example 1). PEval scans all vertices in parallel, and sets $e(v)$ for each vertex v such that $e(v)$ is the minimum-weight edge incident to v.

(2) IncEval. In each IncEval round, it first handles incoming messages for each border vertex v: (a) it updates local $p(v)$ if the message contains a smaller parent ID; and (b) it updates local $e(v)$ if the message contains a lower-weight edge. After these, IncEval then proceeds in four stages as follows. (1) Grafting: parallel for each vertex v in V_i, it adds $e(v)$ to E_T, and merges pseudo trees that are connected by e. (2) Pointer jumping: this step is similar to WCC. (3) Contracting: parallel for each edge, it removes the ones internal to a pseudo tree. (4) Maintenance: parallel for each vertex v, it updates $e(v)$ with the minimum-weight remaining edge incident to v.

(3) Assemble. It returns all selected edges as the minimum spanning tree T of graph G when no more changes can be made.

Benefits over VC. A common VC algorithm for MST is a parallelized version of Prim's algorithm [76], which works as an extension to HashMin (see Section 3.3). Given a graph $G = (V, E, L)$, during the label propagation stage, each vertex maintains the lightest incident edge from which a label-updating message is sent. All these edges form an MST once label propagation has reached a fixpoint.

Algorithm 8: RW in Planar.

Status Declaration: $S_V = \{W\}$ where each column of W (denoted by $\bar{w}(v), \forall v \in V$) is an l-dimension, all-zero array;
$S_G = \{\bar{p}, \bar{c}\}$, where $\bar{p}$ and $\bar{c}$ are both n-dimension arrays, indicating that walker i's position is $\bar{p}[i]$ after $\bar{c}[i]$ steps;

StatusCR: $(\bar{w}(v)[i], \bar{w}'(v)[i]) \Rightarrow \mathsf{rand}(\bar{w}(v)[i], \bar{w}'(v)[i])$;

Function PEval **(** fragment F_i **):**
1 VApply$((\forall v \in V_i) \Rightarrow \bar{w}(v)[k] := \mathsf{rand}(Nbr^+(k)), \forall k \leq l)$;
2 **return** $R_i = (V_i, S'(F_i))$, where $S'(F_i)$ contains updated W;

Function IncEval **(** partial result R_i, updates $\Psi[F_i]$ **):**
3 VApply$((\forall v_k = \bar{p}[k]) \Rightarrow$ GreedyAdvance$(k, V_i))$;
4 **return** $R_i = (V_i, S'(F_i))$, where $S'(F_i)$ contains updated $\bar{p}$;

Procedure GreedyAdvance (k, V_i)**:**
5 **while** $k \leq l$ and $\bar{p}[k] \in V_i$ **do**
6 $\bar{p}[k] := W(\bar{p}[k])[k]$;
7 $k := k + 1$;

(1) In-memory. Let D denote the diameter of graph G. Planar does at most $O(|V|^2)$ total amount of work; in contrast, VC does $O((|V| + |E|)D)$ amount of work. The reduced computation cost stems from the graph contracting operations, which shrink the graph size (possibly exponentially) in each iteration.

Neither $\mathcal{A}_{\mathsf{MST}}$ nor the VC algorithm is parallelly scalable relative to a sequential MST algorithm, which finishes the computation in $O(|E| \log |V|)$ time. Both $\mathcal{A}_{\mathsf{MST}}$ and the VC algorithm incur redundant work for parallelization. That said, both guarantee a linear speedup when using up to $|V|$ cores [44].

(2) Out-of-core. Given a partition $\mathcal{F} = (F_1, \ldots, F_m)$ of graph G, Planar computes MST in at most $\lceil \log \min\{m, D\} \rceil$ rounds, but VC can take as many as D rounds. Similar to the case of WCC, the parallel model of Planar supports beyond-neighborhood computation of GC and shrinking graph size.

5. Random Walk. We then study the graph (massive) random walk (RW) problem. Given a graph $G = (V, E)$, $n = |V|$ walkers and a number l of walk length, RW is to find the terminal positions of these n walkers, where each walker (1) is initially placed on a different vertex of G; (2) uniformly samples an out-neighbor of the current vertex randomly and jumps to that vertex for each step it takes; and (3) terminates its walk after l steps.

Outline. Planar takes the PRAM algorithm of [46] as $\mathcal{A}_{\mathsf{RW}}$. As shown in Algorithm 8, instead of moving each walker step by step, algorithm $\mathcal{A}_{\mathsf{RW}}$ performs batch sampling from each vertex. It first samples an $l \times |V|$ matrix W, where $W_{i,j}$ denotes the $(i+1)$-th stop for a walker stationed at vertex j after taking i steps. Then, the algorithm moves each walker based on pre-sampled W.

Function PEval declares an l-dimension array $\bar{w}(v)$ for each vertex v in the vertex status S_V. In other words, S_V stores matrix W by columns. Over a vertex-cut subgraph $F_i = (V_i, E_i)$, it samples in parallel to fill each element in $\bar{w}(v)$. For conflicting $\bar{w}(v)$ values of a border vertex v, it decides the winner of each element in $\bar{w}(v)$ randomly, with a probability weighted by the number of edges incident to v within fragment F_i. Function IncEval overrides $\bar{w}(v)$ with the conflict-free message for a border vertex v, and moves walkers initiated in F_i based on pre-sampled matrix W (*i.e.,* S_V). Finally, function Assemble returns the final stations of all n walkers.

Benefits over VC. A VC algorithm for RW simply moves each worker by exactly one step in a round of computation. Given a graph $G = (V, E)$ and n walkers, it completes random walk in l rounds, while in each round, a walker step is performed via on-demand sampling of an out-edge. Observe the following.

(1) In-memory. Both $\mathcal{A}_{\mathsf{RW}}$ and the VC algorithm advance all walkers to their terminal positions in exactly $O(l|V|)$ moves. Moreover, both are parallelly scalable in theory, having a linear speedup when using up to $|V|$ processors. This said, algorithm $\mathcal{A}_{\mathsf{RW}}$ can be executed more efficiently on modern architectures, since its workflow has much better locality via batch sampling.

(2) Out-of-core. Given partition $\mathcal{F} = (F_1, \ldots, F_m)$, Planar incurs much less disk I/O than its VC counterpart when the computation is out-of-core for the following two reasons. (a) It runs in fewer I/O rounds. Since $\mathcal{A}_{\mathsf{RW}}$ advances walkers within each subgraph greedily, its computation takes at most $\min\{m, l\}$ rounds; in contrast, VC takes exactly l rounds. (b) For each IncEval round, it only needs to load partial status $R_i = (V_i, S(F_i))$, without the edge data from the disk. This results in much less I/O demand for each round.

A.3 Comparison with MiniGraph

Continuing with WCC in Examples 2–4, we further compare the parallel model of Planar with the hybrid model of MiniGraph [106].

Example 10: MiniGraph parallelizes BFS for WCC. It solves intra-subgraph parallelism via VC, allowing concurrent neighbor visits at each vertex. This creates data dependencies, which limit parallelism and scalability. In contrast, Planar fully exploits hardware concurrency by leveraging the "flat" data accesses of PRAM. □

In contrast to the two-level hybrid model of MiniGraph [106], (1) Planar plugs in PRAM algorithms and utilizes multi-core SIMD parallelism, which fits intra-subgraph parallelism better than VC adopted by MiniGraph (see Section 5); (2) it directly uses decades of work on PRAM algorithms, not requiring a manual code revamp; and (3) it simplifies the execution model of MiniGraph (see Section 4), without asking for delicate tuning efforts.

B PROOF OF THEOREM 1

We study the decision problem of the partitioning problem of Section 4.2, denoted by DPSP and stated as follows.

- *Input*: Planar program $\mathcal{A}$, graph G, the round cost function C_j (Section 4) for round j, an integer m, a memory usage function $M_{\mathcal{A}}$, memory capacity B, and a cost threshold η.
- *Question*: Does there exist a valid m-partition $\mathcal{F} = (F_1, \ldots, F_m)$ such that its peak memory usage is at most B, *i.e.,* $M_{\mathcal{A}}(F_i) + M_{\mathcal{A}}(F_{i+1}) \leq B$ for all $i \in [0, m-1]$, and its round cost is bounded by η, *i.e.,* $C_j(F_i) \leq \eta$?

Proof of NP-hardness. We verify the NP-hardness of DPSP by reduction from 3-partition problem, which is NP-complete (cf. [32]). That is, DPSP is intractable when m is a fixed constant 3. The 3-partition problem is to decide, given a finite set A of positive integers whose element sum is $3S$, whether there exists a 3-partition such that each partition adds up to S. In other words, it is to decide whether there exist three disjoint subsets $A_1, A_2, A_3 \subseteq A$ of equal sum, such that their union is equal to A. More specifically, the requirements are (1) disjoint constraint, *i.e.,* $A_1 \cap A_2 = A_2 \cap A_3 = A_1 \cap A_3 = \emptyset$; (2) union constraint, *i.e.,* $A_1 \cup A_2 \cup A_3 = A$; and (3) equal sum constraint *i.e.,* $\sum_{a \in A_1} a = \sum_{b \in A_2} b = \sum_{c \in A_3} c = S$.

Given a set of n positive integers $A = \{a_1, \ldots, a_n\}$ with $\Sigma_{i=1}^{n} a_i = 3S$, we construct a graph G, a Planar program $\mathcal{A}$, a positive integer m, two positive numbers B and η, a memory usage function $M_A(F_i)$ and a round cost function $C_j(\mathcal{F})$ such that the set A can be partitioned into disjoint A_1, A_2 and A_3 of sum S each iff $C_j(\mathcal{F}) \leq \eta$ and $M_{\mathcal{A}}(F_i) + M_{\mathcal{A}}(F_{i+1}) \leq B$, for $i \in [1, m-1]$. We *w.l.o.g.* write $\sum_{i=1}^{x_1} a_i = \sum_{i=x_1+1}^{x_2} a_i = \sum_{i=x_2+1}^{n} a_i = S$, where $1 \leq x_1 \leq x_2 \leq n$.

Next we present our construction details.

(1) Graph. We build a directed, connected graph $G = (V, E, L)$ with $|V| = 3S$ vertices and $|E| = 3S$ edges. It consists of n *disjoint* directed cycles; the length of the i-th cycle is equal to a_i. More specifically, denote by K_x a length-x directed cycle. We consider a self loop as a length-1 cycle K_1. If both directed edges $\langle v, v' \rangle$ and $\langle v', v \rangle$ exist, it constitutes a length-2 cycle K_2; so on and so forth.

It is easy to see that graph G has the following properties. (a) For each $i \in [1, n]$, there exists a directed cycle K_{a_i}. Apparently, K_{a_i} is a subgraph of G. (b) For any pair $i, j \in [1, n]$ and $i \neq j$, their corresponding cycles K_{a_i} and K_{a_j} have no shared entities.

(2) Planar program $\mathcal{A}$. We consider an inference algorithm for Graph Convolutional Network (GCN) [54] over graph G. Here we start with a brief overview of such an algorithm.

Consider graph $G = (V, E, L)$, where $L \in \mathbb{R}^{l \times |V|}$ is the node feature matrix (*i.e.,* $L(v) \in \mathbb{R}^l$ is the feature vector of $v \in V$). A k-layer GCN is to compute the node embedding $h(v) = h^k(v)$ of each $v \in V$; its i-th layer embedding $h^i(v) \in \mathbb{R}^l$, such that

$$h^0(v) = L(v), \tag{4}$$

$$h^i(v) = \sigma\left(\frac{W_i}{|N(v)|} \sum_{u \in N(v)} h^{i-1}(u) + B_i h^{i-1}(v)\right), i \in \{1, \ldots, k\}. \tag{5}$$

Here $h^{i-1}(v)$ is the node embedding of v from the previous layer, $N(v)$ is the set of neighbors of v. The purpose of $\frac{1}{|N(v)|} \sum_{u \in N(v)} h^{i-1}(u)$ is to aggregate neighboring features of v from the previous layer. Moreover, σ is the activation function (e.g. ReLU) to introduce non-linearity; $W_i \in \mathbb{R}^{l \times l}$ and $B_i \in \mathbb{R}^{l \times l}$ are the trainable parameters at the i-the layer.

Program $\mathcal{A}$ is a round of *forward-pass* computation in GCN. It applies Equation 5 to each $v \in V$. Observe the following:

- the computational cost for the round using p processors is $C_{\mathcal{A}}(F_i, p) = l^3|V_i| + l^3|E_i|$;
- the size of partial state is $|R_i| = l|V_i| + |E_i|$;
- its peak memory usage is $M_{\mathcal{A}}(F_i) = l|V_i| + |E_i|$; and
- its I/O cost is $\mathrm{IO}(F_i) = l|V_i| + |E_i|$.

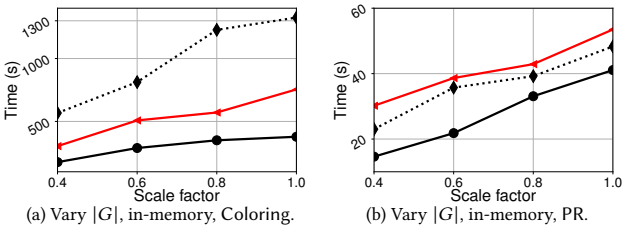
Note that a forward-pass round can be either CPU-bound or I/O bound, depending on the values of feature dimension l.

(a) For a large l such that a round is CPU-bound, the round cost is $C_j(\mathcal{F}) = l^3 \sum_{i=1}^{m} (|V_i| + |E_i|) + l|V_1| + |E_1|$.

(b) For a small l such that a round is I/O-bound, the round cost is $C_j(\mathcal{F}) = \sum_{i=1}^{m} (l|V_i| + |E_i|) + l^3(|V_m| + |E_m|)$.

(3) Other parameters. We set $m = 3$ and $B = 2(l+1)S$. We set η based on the system bottleneck. (a) For a CPU-bound round, $\eta = (6l^3 + l + 1)S$. (b) For an I/O-bound round, $\eta = (2l^3 + 3l + 3)S$.

Intuitively, these require that partition $\mathcal{F}$ includes three subgraphs F_1, F_2, F_3, where $M_{\mathcal{A}}(F_i) + M_{\mathcal{A}}(F_{i+1}) \leq B$ for $i \in \{1, 2\}$. This is, partition $\mathcal{F} = (F_1, F_2, F_3)$ does not break any cycle K_{a_i} in graph G; otherwise, it would produce duplicates (vertices and/or edges) and exceed the memory bound. In other words, a valid partition must distribute cycle K_{a_i} to one of fragments F_1, F_2 or F_3 ($i \in [1, n]$).



(a) Vary $|G|$, in-memory, Coloring. (b) Vary $|G|$, in-memory, PR.

Figure 5: Additional experimental results.

We next verify the correctness of our reduction.

($\Rightarrow$) Suppose that the set A can be partitioned into subsets A_1, A_2 and A_3 of equal sum. We have that $\sum_{i=1}^{x_1} a_i = \sum_{i=x_1+1}^{x_2} a_i = \sum_{i=x_2+1}^{n} a_i = S$. Thus, we construct partition $\mathcal{F}$ as

$$F_1 : \cup_{i=1}^{x_1} K_{a_i};$$
$$F_2 : \cup_{i=x_1+1}^{x_2} K_{a_i};$$
$$F_3 : \cup_{i=x_2+1}^{n} K_{a_i}.$$

One can verify that $\mathcal{F} = (F_1, F_2, F_3)$ is a valid partition that satisfies both the memory constraint and the cost threshold.

($\Leftarrow$) Conversely, suppose that partition $\mathcal{F} = (F_1, F_2, F_3)$ satisfies both the memory constraint and the cost threshold. This implies that $M_{\mathcal{A}}(F_1) + M_{\mathcal{A}}(F_2) = M_{\mathcal{A}}(F_2) + M_{\mathcal{A}}(F_3) = B$; therefore, we have that $|V_i| = |E_i| = S$ for $i \in \{1, 2, 3\}$. Consider *w.l.o.g.* that F_1 consists of k_1 cycles $K_{x_1}, K_{x_2}, \ldots, K_{x_{k_1}}$, F_2 consists of $(k_2 - k_1)$ cycles $K_{x_{k_1+1}}, K_{x_{k_1}+2}, \ldots, K_{x_{k_2}}$, and F_3 consists of $(n - k_2)$ cycles $K_{x_{k_2+1}}, K_{x_{k_2}+2}, \ldots, K_{x_n}$. Here we construct

$$A_1 = \{x_1, x_2, \ldots, x_{k_1}\},$$
$$A_2 = \{x_{k_1+1}, x_{k_1+2}, \ldots, x_{k_2}\},$$
$$A_3 = \{x_{k_2+1}, x_{k_2+2}, \ldots, x_n\}.$$

One can verify that each of A_1, A_2 and A_3 has a sum of S, which makes a valid 3-partition of A. □

C PRAM SIMULATION

Over an input graph $G = (V, E, L)$, a PRAM algorithm $\mathcal{A}$ typically assumes the availability of as many as $P(|V|, |E|)$ processors, where $P(|V|, |E|)$ is a polynomial of $|V|$ and $|E|$; this is beyond reach in practice. In light of this, we simulate PRAM by executing $\mathcal{A}$ using a constant p cores via multiplexing and dynamic load balancing.

A PRAM algorithm $\mathcal{A}$ specifies a sequence of lockstep operations, where each operation can be decomposed into at most $P(|V|, |E|)$ independent tasks. To simulate $\mathcal{A}$ at a machine with p of CPU cores, Planar runs lockstep operations sequentially, with implicitly placed barriers among consecutive ones; all independent tasks in the same lockstep are parallelizable and executed by a size-p thread pool.

More specifically, the simulation works as follows. (1) For each lockstep operation, its $P(|V|, |E|)$ independent tasks are multiplexed using p parallel threads. (2) As soon as all tasks within a parallel operation are concluded, all p threads in the thread pool synchronize

via a barrier, which is automatically placed by Planar via a conditional variable primitive. (3) The thread pool proceeds with the next operation in sequence, until all locksteps of $\mathcal{A}$ are completed.

For load balance within a lockstep operation, all independent, parallelizable tasks are initially placed in a task queue (whose size is at most $P(|V|, |E|)$). Threads in the size-p thread pool work asynchronously to consume this queue. Since each task is small in size, this mechanism ensures that all threads are mostly busy until completion. To minimize contention on the task queue, we implement it using a lock-free queue primitive (see Section D for how to support concurrent data accesses).

D SYSTEM IMPLEMENTATION

We next outline the implementation of Planar.

Architecture & key modules. Figure 6 depicts the architecture of system Planar, where the arrows indicate data flow among modules. It separates the data processing modules, which load the input graph and carry out computation over it, from the control modules, which manage and optimize the execution workflow.

Data processing modules. Planar implements a data pipeline that continuously reads from and writes to the secondary storage. As mentioned earlier, it partitions the memory into an off-stage area and an on-stage area to overlap CPU and I/O operations.

(1) Off-stage area. This area provides a buffering space for block loading and discharging. It has two modules. (a) BlockGrouper loads blocks into the memory and, when instructed by Scheduler, submits a block grouping to Executors. (b) BlockWriter persists the produced partial states at the disk, which includes both the subgraph F_i and its associated status $S(F_i)$. Using dedicated threads, they perform asynchronous I/O via io_uring to maximize throughput.

To make space for new blocks, off-stage area first reclaims memory from unchanged stale blocks to reduce disk writes; it persists and evicts updated status $S(F_i)$ only when necessary. Moreover, it adopts incremental writing; that is, it rewrites only the updated parts of a stale partial state. For example, when the topology of F_i remains unchanged, it persists only the differences of the updated status $S(F_i)$. Both strategies mitigate I/O interference [45] on SSDs.

(2) On-stage area. This area acts like shared memory for PRAM. Executors employ p threads to process a block grouping. Logically, they treat all blocks in a group as a single subgraph. For an entity shared among multiple blocks, we make changes to a master copy; the changes are synchronized to all copies at the end of the round.

UpdateCache is a global key-value store, which hosts updates Ψ in memory (see Section 3). With a bounded size, it caches as many ephemeral updates in memory as possible and uses secondary storage only when necessary. In our experiments over various graphs, a small portion (*e.g.,* 10%) of the memory capacity suffices.

Control modules. Planar has 3 control modules. (1) ConfigManager maintains system configs and the profiling results, *i.e.,* the PEval cost and model $M_{\mathcal{A}}$, making updates at background (see Section 4.2). (2) StateManager tracks "active" blocks that are pending processing within each round; at the end of each round, it checks updates in UpdateCache and decides whether to conclude computation by triggering Assemble. (3) Scheduler adjusts partitioning and scheduling at runtime (see Section 4.3), *i.e.,* block grouping and adaptive

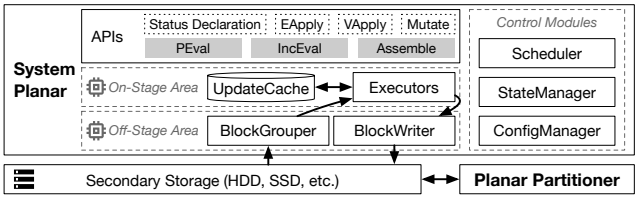


Figure 6: Planar architecture.

scheduling. It actively monitors runtime statistics, maintains the dependency graph *w.r.t.* graph blocks, makes scheduling decisions, and issues the decisions to data processing modules.

Partitioner. It implements the speculative partitioner of Section 4.3, storing subgraphs in an optimized Compressed Sparse Row format. The format is more compact than the ones used in prior systems [65, 106], since we have removed some data redundancies, by compressing the vertex ID offsets into shorter bit sequences.

PRAM simulation. The theoretical model of PRAM assumes a polynomial number of processors [7, 38], which is beyond the reach of a machine in practice. To bridge the gap, we simulate PRAM with a physical machine with p CPU cores as follows.

Synchronization. As mentioned earlier, to guarantee the correctness of simulation, we (1) place an implicit synchronization barrier after each parallel operator, *i.e.,* VApply or EApply; and (2) ensure that all read accesses *precede* any write within each parallel operator.

Planar implements these with versioned data accesses. More specifically, we consider *w.l.o.g.* a VApply operation that mutates vertex status S_V. Entering VApply, a copy S'_V of S_V is created. During VApply, Planar reads from S_V and writes to S'_V. Finally when VApply concludes, the stale status S_V is replaced with the new S'_V. To avoid duplication and reduce memory overhead, we create a new version of an element on-demand only when it updates.

Planar has two versions of updates Ψ in UpdateCache: Ψ_{pre} for the last round, and Ψ_{cur} for the current. A round starts by fetching from Ψ_{pre}, and ends with aggregating new border updates into Ψ_{cur}.

Load balancing. Each parallel operator can generate a number of independent, parallel tasks. To allocate these tasks to p processors with balanced workload, we employ a size-p thread pool in Executors. Initially, all generated tasks are placed in a task queue. Each thread then polls the task queue whenever it becomes idle, and executes the obtained task. To reduce contention on the task queue, we group multiple tasks into a package, and a thread consumes an entire task package for each poll. The packaging granularity is dynamically determined heuristically based on p and the task size.

Lock-free parallelism. To further optimize parallel performance, Planar exploits lock-free parallelism. Consider concurrent writes to data Φ. If Φ cannot be implemented using an atomic data structure, we adopt the copy-on-write technique. That is, whenever a thread attempts to modify Φ, it creates a thread-local copy Φ' of Φ, writes new value to Φ', and makes an atomic switch from the old to the new. Such fine-gained copy-on-write technique is known to reduce write contentions and improve parallel performance [12, 81].

Failure recovery. While single-machine systems inherently present a single point of failure, Planar offers a unique advantage in resilience compared to existing single-machine graph analytic systems. The previous systems typically maintain all intermediate

states in memory without any persistence mechanism and thus lack the ability to recover quickly after a failure.

More specifically, Planar has a built-in capability for faster recovery from failures. Unlike other systems, Planar persists both the status data and graph data to external storage at each round of computation (Section 3.2). This approach effectively creates checkpoints that capture the state of the computation at various stages. In the event of a machine failure, Planar can utilize these checkpoints to resume operations from the last saved state (*i.e.,* the last completed round), rather than restarting the computation from the beginning. The only lost progress would be the unfinished round at the time of failure, whose border updates are cached in-memory and cannot be recovered. This mechanism can significantly reduce redundant computations and minimize the recovery time required to restore the system to its previous state.

By maintaining these persistent checkpoints, Planar ensures that even in the event of a failure, the progress made is not entirely lost, thereby offering a more robust solution for large-scale graph analytics on single machines. It provides a meaningful improvement in reliability and operational continuity in scenarios where single-machine setups are preferred for their cost-effectiveness.

E ADDITIONAL EXPERIMENTAL RESULTS

Varying the size of in-memory graph G, Figure 5 shows the performance of Planar in comparison with MiniGraph and Galois. Planar consistently outperforms the baselines in both Coloring and PR, regardless of the graph size.