MiniGraph: Querying Big Graphs with a Single Machine

Xiaoke Zhu[1,2], Yang Liu[1,2], Shuhao Liu[2], Wenfei Fan[2,3,1]
[1]Beihang University [2]Shenzhen Institute of Computing Sciences [3]University of Edinburgh
{zhuxk@,liuyang@act.}buaa.edu.cn,shuhao@sics.ac.cn,wenfei@inf.ed.ac.uk

ABSTRACT

This paper presents MiniGraph, an out-of-core system for querying big graphs with a single machine. As opposed to previous single-machine graph systems, MiniGraph proposes a pipelined architecture to overlap I/O and CPU operations, and improves multi-core parallelism. It also introduces a hybrid model to support both vertex-centric and graph-centric parallel computations, to simplify parallel graph programming, speed up beyond-neighborhood computations, and parallelize computations within each subgraph. The model induces a two-level parallel execution model to explore both inter-subgraph and intra-subgraph parallelism. Moreover, MiniGraph develops new optimization techniques under its architecture. Using real-life graphs of different types, we show that MiniGraph is up to 76.1× faster than prior out-of-core systems, and performs better than some multi-machine systems that use up to 12 machines.

1 INTRODUCTION

Big graph analytics has mostly been a privilege of big companies by employing a cluster of machines. For instance, to compute connected components of a graph with billions of vertices and trillions of edges, Google employs a 1000-node cluster with 12000 processors and 128 TB of aggregated memory [66]. To mine 3-FSM with support = 25000 on a million-edge graph, DistGraph [68] uses 128 IBM BlueGene/Q supercomputers and 32,768 GB memory. A number of graph systems have been developed to explore multi-machine parallelism, *e.g.,* [3, 27, 33, 35, 52, 56, 69, 72, 76, 79, 82]. However, big graph analytics via multiple machines is often beyond reach of small companies, which cannot afford such enterprise clusters.

Moreover, many parallel graph systems "have either a surprisingly large COST, or simply underperform one thread" [53]. For example, single-source shortest path (SSSP) is "essentially not scalable with an increasing number of machines" [74]. This is because multi-machine systems typically adopt the shared-nothing architecture, and the more machines are used, the heavier their communication cost is incurred. In addition, machines in such a system are often under-utilized due to unbalanced workload.

To rectify the limitations of the multi-machine systems, single-machine systems have been studied to explore multi-core parallelism, notably out-of-core systems to process data that is too large to fit into the main memory of a single machine at once [11, 32, 45, 51, 62, 75, 83]. The systems are developed on a machine that has a number of CPU cores, but limited memory capacity and disk I/O bandwidth. To get over the bottleneck, these systems have mostly focused on how to optimize I/O and memory efficiency.

Despite the efforts, disk I/O remains the bottleneck of the single-machine systems. Consider Weakly Connected Components (WCC). Given a graph G, WCC is to compute the maximum subgraphs of G in which all vertices are connected to each other via a path, regardless of the direction of edges. Consider GridGraph [83], a state-of-the-art out-of-core system featuring optimal I/O. We run the

System	friendster (mean distance: 5.1)		web-sk (mean distance: 13.7)	
	# Supersteps	Disk Read	# Supersteps	Disk Read
GridGraph	21	135 GB	120	367 GB
MiniGraph	6	74 GB	9	82 GB

Table 1: WCC performance on GridGraph and MiniGraph.

out-of-box WCC implementation of HashMin [77] that comes with GridGraph as a benchmarking application. The test was conducted on a workstation powered with a 10-core CPU with hyperthreading and 16 GB DDR4 memory. Graph G is `friendster` or `web-sk` [61]; both have ~1.8 billion edges, and amount to ~31 GB, about twice the memory size. We measure its system I/O using `iostat`. As shown in Table 1, while the two graphs are similar in size, GridGraph incurs drastically different I/O costs. It induces 2.7× disk read on `web-sk` as on `friendster`, which is relatively denser.

An in-depth analysis reveals that the excessive I/O often stems from the vertex-centric model [33, 52, 83] adopted by GridGraph for parallel computation, referred to as VC. Under VC, an algorithm employs a user-defined function to process the immediate neighborhood of each vertex (or edge), and exchanges information between vertices via message passing. To send a message from a memory-resident vertex to a memory-absent one, it inevitably requires swapping their data in and out of the memory; as a consequence, VC generally incurs more disk I/O when G has a larger diameter.

Moreover, while VC is natural for graph algorithms such as PageRank [17] and HashMin [77] (for WCC), it is neither easy to write nor efficient to execute *e.g.,* an optimized WCC algorithm via Breath-First Search [37] and graph pattern matching algorithms with subgraph isomorphism or graph simulation [23] under VC.

Can we systematically reduce the I/O cost of an out-of-core graph system and improve multi-core parallelism? Given a computational problem, would other parallel models fit it better than VC?

MiniGraph. To answer these questions, we develop MiniGraph, an out-of-core system for graph computations with a single machine. It is the first single-machine system that extends the graph-centric model (GC) of [24, 27] from multiple machines to multiple cores. It shows that GC speeds up beyond-neighborhood computation and reduces I/O, and moreover, simplifies parallel programming by parallelizing existing sequential algorithms across cores.

As shown in Table 1, when computing WCC over `friendster` (resp. `web-sk`), the benefit of the beyond-neighborhood computation (GC) is evident: MiniGraph (a) takes 6 (resp. 9) supersteps to converge under the bulk asynchronous model (BSP) [70], as opposed to 21 (resp. 120) steps with GridGraph; (b) reads 74 GB (resp. 82 GB) of data in contrast to 135 GB (resp. 367 GB) of GridGraph; and (c) is less sensitive to the distribution of the input graphs.

However, it is nontrivial to migrate GC to a single-machine system. It introduces new challenges such as memory constraint and I/O cost. MiniGraph approaches the following challenges that are non-existent in prior GC systems: (a) out-of-core computations and in-memory synchronizations; (b) multi-core parallelism; and (c) resource scheduling. It has the following unique features.

(1) A pipelined architecture. MiniGraph proposes an architecture that pipelines access to disk for read and write, and CPU operations for query answering. The idea is to overlap I/O and CPU operations, so as to "cancel" the excessive I/O cost, and promote sequential accesses to the disk. Moreover, it employs a shared in-memory data structure for message passing and efficient synchronization. This architecture decouples computation from memory management and scheduling, giving rises to new opportunities for optimizations.

(2) A hybrid parallel model. MiniGraph proposes a hybrid model, supporting both the data-partitioned parallelism of GC and the operation-level parallelism of VC. Under the memory constraint, it promotes better utilization of the multi-core resources and avoids fragmentation of the input graph. Moreover, it exposes a unified interface such that the users can benefit from both and can choose one that fits their applications and graphs the best.

(3) Two-level parallelism. The hybrid model also enables two-level parallelism: inter-subgraph parallelism via high-level GC abstraction, and intra-subgraph parallelism for low-level VC operations. This presents new opportunities for improving multi-core parallelism. However, its relevant scheduling problem is NP-complete. This said, we develop an efficient heuristic to allocate resources, which is dynamically adapted based on resource availability.

(4) System optimizations. MiniGraph develops unique optimization strategies enabled by the hybrid parallel model. It employs a lightweight state machine to model the progress of cores working on different subgraphs, and tracks messages between cores. These allow it to explore shortcuts in the process to avoid redundant I/O.

Contributions & organization. After reviewing VC and GC parallel models (Section 2), we present MiniGraph as follows:

- its pipelined architecture and a system overview (Section 3);
- the hybrid parallel model (Section 4);
- the two-level parallel execution model, including resource scheduling (Section 5.1) and system optimizations (Section 5.2); and
- an experimental study (Section 6). Using real-life graphs, we find the following. (a) With various out-of-core workloads, MiniGraph outperforms prior out-of-core systems by 10.8× on average, up to 76.1×. (b) It is able to query a graph of size 10× of the memory capacity. (c) It outperforms the state-of-the-art multi-machine systems when they use up to 12 machines.

2 BACKGROUND AND MOTIVATION

In this section, we review parallel graph (programming) models.

Consider graph $G = (V, E, L)$, directed or undirected, where V is a finite set of vertices; $E \subseteq V \times V$ is a set of edges; each vertex v in V (resp. edge $e \in E$) carries label $L(v)$ (resp. $L(e)$) for properties.

In principle, a graph parallel model determines how users can program with the system and how the programs are executed in parallel. Two types of parallel models have been implemented in (multi-machine) graph systems, namely, VC and GC.

The vertex-centric model (VC). Initially proposed by Pregel [52], VC has been the *de facto* go-to model for parallel graph systems. Its principle is for users to think like a vertex: a vertex program is "pivoted" at a vertex; it may only directly access information at the current vertex and its adjacent edges [33, 52]; information is exchanged between "remote" vertices via message passing. Several variants of VC are in place. For example, the GAS model of PowerGraph [33] requires three user-specified functions: (1) Gather defines how a vertex v aggregates incoming messages from its neighbors; (2) Apply specifies how v updates its own attributes based on the message received; and (3) Scatter defines what messages at v are generated as outgoing messages to its neighbors.

It is natural to program with VC for a problem when its computation can be distributed to vertices and is centered at each vertex, such as PageRank. It is, however, nontrivial to write VC algorithms for problems that are constrained by "joint" conditions on multiple vertices, *e.g.,* graph simulation [54] and subgraph isomorphism [31]. While a variety of conventional sequential algorithms have been developed for such problems, to program with VC, one has to recast the existing algorithms into the VC model. Moreover, such VC programs are often inefficient since they incur heavy message passing. The issue is more staggering for out-of-core systems since we have to repeatedly load the adjacent list of a vertex from disk to memory when the recursive/induction process backtracks.

Example 1: As a common benchmarking algorithm for WCC, HashMin works as follows under VC: (1) initially, each vertex is assigned a distinct numeric label; (2) in every iteration, each vertex collects the labels from its neighbors (via bi-directional message passing along edges), and updates its own label with the minimum within the neighborhood; (3) when no more label updates can be made, it returns the total number of distinct labels in the graph.

On graph G of Figure 1a, HashMin takes 5 supersteps under BSP as shown in Figure 1b, where active vertices are in light color and inactive ones are in gray. We also count the number of message passed in each superstep, marked as "#Ops". As the computation progresses, more vertices become inactive, and less messages are passed.

Observe the following: (1) The labels of vertices in Subgraph D are overwritten repeatedly; many initial messages and label updates are redundant. In total 92 messages are passed, while G has 16 edges only. (2) It takes 5 supersteps. Every edge must be swapped in and out of the memory for 5 times if G cannot fit in the memory entirely. It is translated to a disk read demand that is 5× of the size of G. □

The graph-centric model (GC). Proposed by GRAPE [27], GC carries out data-partitioned parallelism. Given a big graph G, it partitions G into fragments (subgraphs) using existing graph partitioners (edge-cut [12, 42], vertex-cut [16, 33, 43] or hybrid [25]), and distributes the fragments to different workers. It parallelizes sequential graph algorithms. More specifically, for a graph computational problem Q, users may provide three (existing) sequential algorithms PEval, IncEval and Assemble, referred to as a PIE program:

- PEval is a conventional algorithm for Q such that all workers execute it on their local fragments in parallel, and produce partial results (*partial evaluation*); the values of border nodes (*e.g.,* vertices with edges to other fragments) are exchanged as messages;
- IncEval is a sequential incremental algorithm for Q; it is repeatedly executed to refine the partial results by treating messages from other workers as updates (*incremental computation*); and
- when no more messages are exchanged, Assemble aggregates partial results from all the workers and forms the final answer.

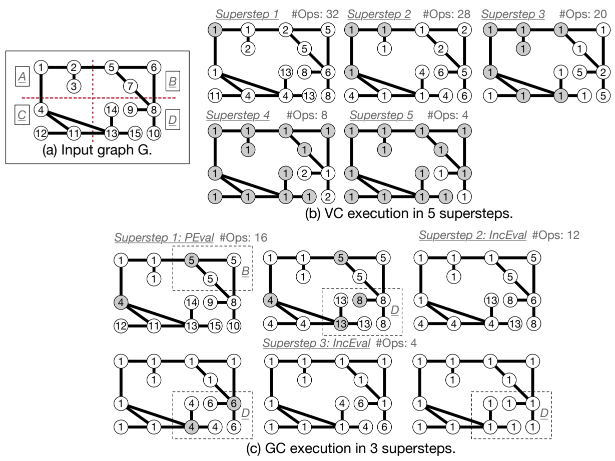

(a) Input graph G. (b) VC execution in 5 supersteps. (c) GC execution in 3 supersteps.

Figure 1: WCC computation on G with VC or GC.

The parallelized computation starts with PEval, and is followed by iterative IncEval until a fixpoint is reached. Under a generic condition, it guarantees to converge at correct answers as long as the sequential algorithms PEval, IncEval and Assemble are correct [28]. All three functions operate on subgraphs of graph G.

As opposed to VC, GC supports beyond-neighborhood computations. It simplifies parallel programming by parallelizing existing sequential algorithms; moreover, it reduces unnecessary recomputation via iterative IncEval. However, for problems such as PageRank, when graph partitions are not "well balanced" [25], the communication cost of a GC algorithm between workers may be higher than its VC counterpart. Moreover, the parallelism is explored at the subgraph level, not at the vertex/edge-level in each subgraphs.

Example 2: As opposed to HashMin, GC parallelizes a more efficient sequential WCC algorithm. After labeling all vertices of graph G with distinct integers, the algorithm proceeds on each fragment (subgraph) F_i of G with single-threaded worker W_i. (1) Starting with PEval, W_i initiates BFS from the vertex v that has the minimum label $L(v)$ among all the vertices in F_i. During the BFS, it overwrites their labels with $L(v)$. (2) As soon as no vertex remains active, W_i generates messages comprising the updated labels of border nodes of F_i. (3) With all updates received from other workers, if v's update carries a smaller label, W_i overwrites $L(v)$; it invokes IncEval to incrementally refine the partial results until no message is exchanged among workers. (4) Finally, the algorithm calls Assemble to count WCC as the total number of distinct labels from all workers.

Suppose that graph G is partitioned into 4 subgraphs, as depicted in Figure 1a. GC completes WCC in 3 supersteps under BSP (1 round of PEval and 2 rounds of IncEval), as illustrated in Figure 1c.

Observe the following: (1) GC reduces redundant computation and needs 3 supersteps only. (2) It passes 32 messages on G (-65% from VC), and requires loading each subgraph only three times (-40% from VC). This said, the computation within each subgraph is conducted *sequentially*, and there is room for improvement. □

To the best of our knowledge, no single-machine systems supports the GC model despite its efficiency and ease of programming.

3 MINIGRAPH: AN OVERVIEW

In this section, we present an overview of MiniGraph.

Challenges. GC is designed for multi-machine systems [24, 27]. Extending it to a single-machine system introduces new challenges.

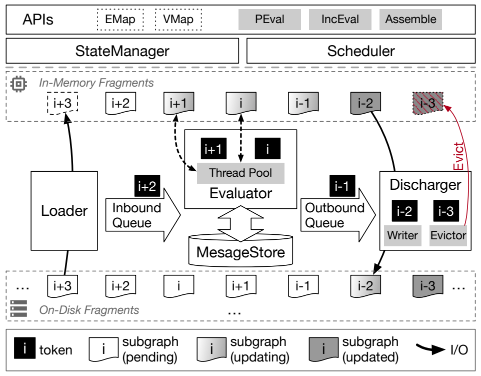


Figure 2: The pipelined architecture of MiniGraph.

- *Out-of-core computation.* A single-machine system has to resort to secondary storage as memory extension when an input graph exceeds its memory capacity. Managing the in-memory and the on-disk parts of an input graph is crucial to performance.
- *Synchronization.* GC passes messages among workers for synchronization. With a shared memory, can a single-machine system synchronize more efficiently via shared data structures?
- *Parallelism.* GC exploits data-partitioned parallelism only. With limited memory capacity, it would result in either underutilization of the CPU cores or excessive graph fragmentation. Systemically balancing the two factors is challenging.
- *Scheduling.* Work migration incurs low cost among cores in a single machine, due to its shared-memory architecture. Can we leverage flexible resource scheduling to improve performance?

In light of these, MiniGraph proposes a pipelined architecture, a hybrid parallel model of VC and GC, two-level parallelism, and unique optimizations on resource scheduling and I/O.

Graph partitioning. Previous single-machine systems organize an input graph as a large number of chunks (*a.k.a.* shards [45], blocks [83]), each consisting of a (possibly small) list of edges. This design serves the sole purpose of gathered I/O, while the system schedules and operates directly on edges. It provides fine-grained scheduling, yet may not work well with the block storage.

MiniGraph explores a conceptually different approach. It partitions a large graph G such that one or more fragments can fit into the memory. A fragment is essentially a *subgraph* of G. MiniGraph adopts a two-level abstraction: (1) at a high level, it sees a subgraph as the atomic unit for scheduling and I/O; and (2) when processing a subgraph that is in memory, it may carry out parallel computation on the edges and vertices. Our empirical study shows that the two-level abstraction improves multi-core parallelism.

MiniGraph may use any graph partitioners designed for multi-machine parallel systems *e.g.*, [12, 16, 25, 33, 42, 43, 59]. It works with any existing graph partitioners, by taking as input a *partitioned graph* and making scheduling/execution decisions accordingly. Deciding an optimal graph partitioner or the number of partitions is, however, highly non-trivial and beyond the scope of this paper.

A pipelined architecture. MiniGraph takes as input the subgraphs (fragments) $F_0, F_1, \ldots, F_{n-1}$ of a (possibly large) G; the subgraphs are initially stored on disk. MiniGraph iteratively processes the subgraphs in a pipelined architecture, as shown in Figure 2.

This architecture overlaps I/O and CPU operations, and conducts computations on in-memory subgraphs while loading pending ones

Algorithm 1: A conceptual implementation of EMap

```
Scope Variables: local subgraph $F_i$, intermediate data $\Pi_i$.
Input: active vertex set $\Delta \in 2^V$, update function $f_E : (V, V) \to 2^V$.
Output: updated active vertex set $\Delta' \in 2^V$.
1 $\Delta' := \emptyset$;
2 foreach $u$ in $\Delta$ do in parallel
3     foreach $v$ in $u$.neighbors do in parallel
4         $\Delta' := \Delta' \cup f_E(u, v)$        /* $F_i$ and $\Pi_i$ may be accessed. */
5 return $\Delta'$;
```

Algorithm 2: A conceptual implementation of VMap

```
Scope Variables: local subgraph $F_i$, intermediate data $\Pi_i$.
Input: active vertex set $\Delta \in 2^V$, update function $f_V : V \to 2^V$.
Output: updated active vertex set $\Delta' \in 2^V$.
1 $\Delta' := \emptyset$;
2 foreach $u$ in $\Delta$ do in parallel
3     $\Delta' := \Delta' \cup f_V(u)$        /* $F_i$ and $\Pi_i$ may be accessed. */
4 return $\Delta'$;
```

from the disk. It mitigates the excessive cost of I/O, improving CPU utilization by reducing idle waiting. While prior out-of-core systems implement a similar notion (*e.g.*, [45, 51]), their VC inevitably generates scattered I/O; in contrast, MiniGraph promotes subgraph-based I/O of GC, and consistently accesses the disk *sequentially*.

More specifically, MiniGraph breaks down the out-of-core processing of a subgraph F_i into three consecutive stages: (1) read F_i into the memory, (2) compute and update F_i, and (3) if necessary, write the updated F_i back to the secondary storage. It has three components: Loader, Evaluator, and Discharger, for the three stages, respectively. They work asynchronously in a pipeline, and are loosely coupled via two task queues InboundQueue and OutboundQueue.

Loader. Working in a dedicated thread, Loader continuously selects and reads a memory-absent subgraphs from disk into the memory, as long as system memory capacity is not exhausted. It operates regardless of the progress of other system components. This design makes full utilization of disk read bandwidth. Instead of many scattered reads, Loader issues a small number of bulk read requests, which can improve read throughput and reduce system interrupts.

Once a subgraph F_i is loaded into the memory, Loader pushes its token T_i into InboundQueue, where T_i includes metadata of F_i such as pointers to the data and its current state.

Evaluator. As the downstream component of Loader, Evaluator is responsible for efficient execution of an application. It pops token T_i from InboundQueue, locates F_i in memory, and initializes a virtual worker W_i with reserved thread allocations to execute the program on F_i. During the execution, W_i can update F_i in place, and may generate updates to other subgraphs. The update messages are cached in MessageStore, an array-like storage in memory.

To fully exploit the multi-core parallelism and cache locality, Evaluator manages a global thread pool, whose size is decided by the system hardware concurrency. Whenever threads become idle, it triggers Scheduler, which re-allocates the resources to workers (see below). Once a worker completes, Evaluator reclaims the reserved threads and pushes token T_i to OutboundQueue for discharging.

Discharger. As a counterpart of Loader, Discharger writes the subgraph data back to the disk. It consumes OutboundQueue continuously in a dedicated thread. For a popped token T_i, it (1) writes the updated F_i contiguously to disk, and (2) records F_i's latest state to StateManager for maintenance. After F_i is fully discharged, Discharger evicts it from memory and reclaims space for Loader.

Key modules. MiniGraph implements the following unique modules for programming, execution and optimization.

MessageStore. MiniGraph uses MessageStore, an in-memory data structure, for synchronization among parallel workers, by caching and managing pending messages. In contrast to worker-to-worker message passing in multi-machine systems [24, 27], MessageStore works more efficiently in a shared-memory environment.

We implement it as a compact variable-size array for space efficiency (details in Appendix B of [7]). Intuitively, its space complexity is closely related to the partitioning strategy: the more border vertices/edges we have, the larger MessageStore is. Yet, it still takes only a small portion in memory. For example, under various workloads over `web-sk`, it consumes 1.3% of the memory at the peak.

APIs for a hybrid parallel model. MiniGraph exposes PIE+, a unified interface that integrates VC with GC programming (see Section 4). Users can not only parallelize sequential graph algorithms under GC to simplify parallel programming, but also further explore intra-subgraph parallelism under VC via the new interfaces.

Scheduler. Scheduler tracks and allocates threads in ThreadPool in which each thread corresponds to a physical CPU core. It makes decisions to assign physical threads to virtual workers to carry out (parallel) computations on fragments. It also makes active adjustments to support two-level parallelism: when a thread is available, Scheduler allocates it to either a new worker by consuming InboundQueue for speeding up inter-subgraph parallelism, or a running worker for improving intra-subgraph parallelism.

StateManager. As computation progresses on the subgraphs of G, StateManager maintains a state machine to model their status. It is a low-cost method for convergence detection, and it helps identify chances to skip redundant I/O or computation. It is implemented as a lightweight data structure and maintains only a few states per subgraph, taking negligible space in memory.

We will present the details of the hybrid parallel model, two-level parallelism and state manager in Sections 4, 5.1 and 5.2, respectively.

4 HYBRID PARALLEL MODEL

In this section we introduce the hybrid parallel model of MiniGraph. It aims to improve multi-core parallelism by enriching inter-subgraph parallelism (GC) with intra-subgraph parallelism (VC).

Overview. Given a graph computational problem Q and a graph G, MiniGraph processes subgraphs $F_0, F_1, \ldots, F_{n-1}$ of G. It assigns virtual worker W_i to F_i ($i \in [0, n-1]$). To simplify the discussion, we adopt BSP [70] for synchronization. In each superstep, each fragment is processed exactly once. When F_i is loaded into the memory, W_i is activated and allocated CPU resources by Scheduler.

Recall from Section 2 that under GC, users provide a PIE program for Q, which consists of three sequential algorithms PEval, IncEval and Assemble. Each worker W_i executes PEval on its local subgraph F_i for partial evaluation; it then iteratively and incrementally refines the partial results via IncEval on F_i by treating messages between workers as updates, until it reaches a fixpoint, followed by Assemble to aggregate the partial results from all workers into the final answer.

Algorithm 3: PEval for WCC under PIE+.

```
Input: subgraph $F_i = (V_i, E_i, L_i)$.
Output: set $CC_i$ of connected components in $F_i$.
Message Preamble:          /* candidate set $C_i$ includes all border nodes. */
    VMap ($V_i$, function ($v$) { $v$.root := $v$; return $\emptyset$; }) ;
1  $CC_i := \emptyset; \Pi := V_i$;          /* $\Pi$ is the set of unvisited vertices. */
2  while $\Pi$ not empty do
3      find root vertex $v_r$, such that $L_i(v_r) = \min_{u \in \Pi} L_i(u)$;
4      $CC_i := CC_i \cup \{v_r\}$; remove $v_r$ from $\Pi$;
5      $\Delta := \{v_r\}$;
6      while $\Delta$ not empty do $\Delta$ := EMap ($\Delta$, BFSRecur);
7  return $CC_i$;
Message Segment: $M_i := \{(v, L_i(v.\text{root})) \mid v \in C_i\}$; $f_{\text{aggr}} := \min$;
8  Procedure BFSRecur ($u, v$) :
9      if $v$ not in $\Pi$ then return $\emptyset$;
10     remove $v$ from $\Pi$; $v$.root := $v_r$;
11     return $\{v\}$;
```

In principle, each W_i runs PEval and IncEval *sequentially*.

The key idea of the hybrid parallel model is two-level parallelism. At the top level, it adopts GC to benefit from data-partitioned inter-subgraph parallelism, parallelize existing sequential graph algorithms, and speed up beyond-neighborhood computations. Moreover, within each subgraph, it parallelizes the execution of PEval and IncEval by enforcing VC; that is, once a subgraph is in memory, it leverages the shared-memory architecture of a single machine and promotes intra-subgraph vertex/edge-level parallelism via VC.

Example 3: Continuing with Example 2, consider the execution of WCC under GC. On subgraph A (see Figure 1), it initiates BFS from root vertex with an initial label 1. It recurses as two independent BFSes starting from both neighbors of vertex 1, *i.e.,* vertices 2 and 4, which then proceed as BFSes from vertices 11 (subgraph C), 13 (subgraph B), and 5 (subgraph D), so on and so forth. If we execute each recursion with 2 parallel threads, we get a 2× speedup. □

PIE+. In light of this, we propose a new programming interface, called PIE+. Given a problem Q, MiniGraph also takes algorithms PEval, IncEval and Assemble for Q, and follows the same workflow of GC for inter-subgraph parallelism. Moreover, it extends PIE with two additional primitives, EMap and VMap, along the same lines as the VC interface of Ligra [63]. These primitives make PEval and IncEval parallel within each subgraph at the edge/vertex level.

EMap and VMap work on a set Δ of "active" vertices, which are to be further processed. Intuitively, EMap and VMap implement flatMap of functional programming, to execute an update function at each active vertex v in Δ and moreover, "flat map" v to a new set of vertices that will remain active or get activated after the operation.

As shown in Algorithm 1, EMap takes as input Δ and an edge update function f_E. For each active vertex u in Δ, EMap makes updates along each of u's outgoing edge by applying f_E, executed in parallel. Here f_E is a user-defined function. It processes an edge $e = \langle u, v\rangle$ from u, makes updates based on the data associated with e, u and v, and returns a new (possibly empty) set of vertices that remain active after the update. Moreover, f_E may access the local subgraph F_i and intermediate data structure Π_i of PEval and IncEval.

Similarly, VMap takes Δ and a vertex update function f_V as input.

Algorithm 4: IncEval for WCC under PIE+.

```
Input: subgraph $F_i = (V_i, E_i, L_i)$, $CC_i$, incoming messages $M_i$.
Output: refined set $CC_i$ of connected components in $F_i$.
1  $\Delta = \{v \mid v \in M_i\}$; VMap($\Delta$, UpdateRoot);
2  return refine($CC_i$);   /* merge roots with the same label in $CC_i$. */
Message Segment: $M_i := \{(v, L_i(v.\text{root})) \mid v \in C_i\}$;
3  Procedure UpdateRoot ($v$) :
4      $v_r := v$.root;
5      while $v_r$ not in $CC_i$ do $v_r := v_r$.root;
6      update $v$.root := $v_r$; $L_i(v_r) := \min(L_i(v_r), M_i[v])$;
7      return $\emptyset$;
```

As shown in Algorithm 2, it aims to parallelize operations that are centered at each active vertex in Δ. Instead of working along edges, f_V operates on each vertex $u \in \Delta$, processes and updates u by accessing F_i and Π_i, and returns an updated active vertex set.

Message passing. Similar to GC, PIE+ approaches inter-subgraph synchronization via message passing. At each subgraph F_i, it declares (a) *status variables* $\bar{x}$ for its *border nodes*, *i.e.,* vertices that have edges to other fragments (under edge-cut); and (b) an aggregate function f_{aggr}, *e.g.,* min and max. Intuitively, the status variables are candidates to be updated by the incremental step of IncEval, and f_{aggr} resolves conflicts when multiple workers assign different values to the same variable $v.\bar{x}$ for a border node v of F_i.

MiniGraph adopts a *push-pull* mechanism for workers to exchange messages through MessageStore. Right after worker W_i gets initialized, it checks MessageStore to *pull* a set M_i of messages, which consists of updated status of the border nodes in F_i. After concluding computation, W_i collects the latest values $v.\bar{x}$ for each border node v, and *pushes* them into MessageStore for aggregation. It inserts $\bar{x}$ into MessageStore$[v]$ if the entry is missing, and applies $f_{\text{aggr}}(\text{MessageStore}[v], \bar{x})$ to resolve the conflict.

Example 4: We present a PIE+ program for WCC. It is the same as the PIE program of [28] for WCC, except that it parallelizes BFSes in PEval and IncEval within each subgraph via EMap and VMap. We consider *w.l.o.g.* edge-cut partitions [12, 42], in which border vertices are those that have edges to another fragment.

(1) PEval. As shown in Algorithm 3, at worker W_i, PEval computes the connected components CC_i of its local subgraph F_i, the partial result at F_i. Here CC_i is the set of root vertices that identify connected components, referred to as *CRoots*. It declares (a) a link to its CRoot, and (b) min as f_{aggr}. PEval (a) initializes an empty CC_i and a set Π of unvisited vertices in F_i (Line 1); and (b) conducts iterative BFS until all vertices are visited, *i.e.,* Π is emptied (Line 2–6). The BFS finds its CRoot v_r at the vertex with the lowest label (Line 3), adds v_r to CC_i to identify a unique connected component (Line 4), and sets the label of each descendant of v_r to the label $L_i(v_r)$ of the CRoot (Line 5–6). MessageStore packs the smallest label for each border node, accessible by workers in the next superstep.

This PEval differs from its counterpart given in [28] only in the following, marked with dotted underlines. VMap is applied to parallelize the initialization of status variables (Message Preamble). EMap parallelizes its recursion (Line 6) to conduct a round of BFS. It works on a set Δ of active vertices, which serves as the recursion queue for BFS. Function BFSRecur is passed to EMap as the update function f_E. For each *unvisited* neighbor v of the working vertex u

(Line 9), BFSRecur overwrites v.root with u's CRoot (Line 10), and adds v to the recursion queue by returning $\{v\}$ (Line 11). During the process, EMap makes parallel visits to the neighboring vertices.

(2) IncEval. As shown in Algorithm 4, IncEval *incrementally* refines the partial result CC_i at F_i, based on messages (updates) from other subgraphs. IncEval (a) updates the label $L_i(v_r)$ of each CRoot v_r as the smallest of its descendants in the same connected component (Line 1); and (b) refines CC_i by merging CRoots ($u_r, v_r \in CC_i$) with the same label, and sets their root to the remaining CRoot (Line 2). The messages are aggregated via min just like in PEval.

This IncEval differs from its counterpart of [28] in its VMap-based implementation of local message propagation. Here function UpdateRoot is passed to VMap as its update function f_V. For each vertex v that receives messages, UpdateRoot works in parallel to (a) find the CRoot v_r of v (Line 4–5), and (b) update the label of v_r by incorporating the incoming messages (Line 6).

(3) When no messages are generated, Assemble collects CRoots from all W_i's and counts the number of CRoots of distinct labels. □

Partial results. Under GC, PEval computes partial result $\mathcal{R}_i$ on the local subgraph F_i while IncEval refines $\mathcal{R}_i$ incrementally based on messages. In a distributed system like GRAPE, $\mathcal{R}_i$ can be cached in memory. However, in MiniGraph, it might exceed the memory capacity to cache all partial results. Thus MiniGraph opts to persist $\mathcal{R}_i$ together with its subgraph F_i. We embed $\mathcal{R}_i$ as the metadata of F_i, which will be discharged and loaded together with F_i.

Writing PIE+ programs. Besides directly parallelizing an existing PIE program (*e.g.*, Example 4), one may cast a VC program into PIE+. Given, *e.g.*, a GAS program of PowerGraph [33], we may convert it into a PIE+ program as follows: (1) initialize MessageStore for caching incoming messages at all vertices; (2) recast Scatter with VMap to generate the outgoing message for each vertex; (3) recast Gather with EMap to perform the message passing via adjacent edges of each vertex, and caching/aggregating them in the corresponding entries of MessageStore; (4) recast Apply with VMap to update each vertex based on its own status and its aggregated messages in MessageStore; (5) embed the implementation above in PEval and IncEval; and (6) specify Assemble based on the output.

With PIE+, users may opt to use either VC or GC that fits their applications and graphs the best. In [7], we give PIE+ programs for single-source shortest path (SSSP), PageRank (PR), breadth-first search (BFS), random walk (RW), and graph simulation (Sim).

5 EXECUTION MODEL

In this section, we present the execution model of MiniGraph, with resource scheduling (Section 5.1) and optimizations (Section 5.2).

5.1 Resource Scheduling

Given a partitioned graph, MiniGraph schedules its work as follows.

Workflow. Taking subgraphs $F_0, F_1, \ldots, F_{n-1}$ of graph G as input, MiniGraph executes a PIE+ program $\mathcal{A}$ in BSP supersteps (rounds). Each round iterates over all subgraphs: the first is a PEval round, followed by IncEval rounds, on each subgraph. A new IncEval round cannot start until the last round completes, and messages generated in the current round are pulled when the next (IncEval) round starts.

Within a round, unprocessed on-disk subgraphs are loaded one after another, while in-memory subgraphs are processed by concurrent workers. To process subgraph F_i, its virtual worker W_i is allocated p_i physical threads from available ones in ThreadPool by Scheduler; W_i then pulls messages generated in the last round, conducts computations on F_i, and aggregates new messages; it releases resources back to Scheduler before it is deactivated.

The scheduling problem. Scheduler has to make two decisions at runtime: (1) when to load and process a subgraph, and (2) how to allocate resources to maximize two-level parallelism. Making optimum decisions for these is, however, highly nontrivial.

A cost model. Given a PIE+ program $\mathcal{A}$, we train a cost function $C_{\mathcal{A}}$ to estimate the execution time of worker W_i on subgraph F_i. We adapt the cost function $C_{\mathcal{A}_{\text{PIE}}}$ of [25] for PIE program $\mathcal{A}_{\text{PIE}}$:

$$C_{\mathcal{A}_{\text{PIE}}}(F_i) = \Sigma_{v\in F_i} h_{\mathcal{A}_{\text{PIE}}}(\bar{x}_i(v)), \qquad (1)$$

where for a vertex v in F_i, $h_{\mathcal{A}_{\text{PIE}}}$ is a learnable polynomial regression model defined over a vector $\bar{x}_i(v)$ of *metric variables*. Here $\bar{x}_i(v)$ takes into account the average in/out-degree of all vertices in G, and various structural information of v, *e.g.*, v's in/out-degree in F_i and G, and the number of v's mirror across all subgraphs of G.

Model $C_{\mathcal{A}_{\text{PIE}}}$ has proven accurate in predicting the computational cost of a PIE program, because polynomials can approximate a continuous function defined on a closed interval [73]. Extending $C_{\mathcal{A}_{\text{PIE}}}$, we define $C_{\mathcal{A}}$ for a PIE+ program $\mathcal{A}$ as

$$C_{\mathcal{A}}(F_i, p_i) = \sum_{v\in F_i} [h^{\text{seq}}_{\mathcal{A}}(\bar{x}_i(v)) + \frac{h^{\text{para}}_{\mathcal{A}}(\bar{x}_i(v))}{\min\{p_i, \lfloor d_i \rfloor\}}], \qquad (2)$$

where p_i denotes the number of allocated threads for worker W_i, d_i is the average degree of vertices in F_i, and $h^{\text{seq}}_{\mathcal{A}}$ and $h^{\text{para}}_{\mathcal{A}}$ are learnable models like $h_{\mathcal{A}_{\text{PIE}}}$. Intuitively, the cost of $\mathcal{A}$ is broken into $h^{\text{seq}}_{\mathcal{A}}$ for unparallelizable computations and $h^{\text{para}}_{\mathcal{A}}$ for parallelizable operations (those defined in EMap and VMap). When more cores p_i are allocated, $h^{\text{para}}_{\mathcal{A}}$ may get a linear speedup subject to maximum d_i.

Model $C_{\mathcal{A}}$ generalizes $C_{\mathcal{A}_{\text{PIE}}}$ by extending the data-partitioned parallelism of GC and adapting to the unique two-level parallel execution model of MiniGraph. It inherits two key properties of $C_{\mathcal{A}_{\text{PIE}}}$, which have been verified empirically in [25]: (1) $C_{\mathcal{A}}$ depends on individual algorithm $\mathcal{A}$, and (2) it remains quite accurate on graphs that are of the same type (*e.g.*, Web graphs) as training graphs, since its regression model assesses the impact of metric variables, and the variables "characterize" topological features of the type of graphs.

We train the polynomial regression models $h^{\text{seq}}_{\mathcal{A}}$ and $h^{\text{para}}_{\mathcal{A}}$ with training samples collected from historic running logs. The cost model of [25] is trained with stochastic gradient descent (SGD) [15], which requires a collection of running logs with $\mathcal{A}$ as a *priori*. In the lack of the training data, $C_{\mathcal{A}}$ may not be accurate. To cope with this, we develop a training procedure to handle cold starts. We follow incremental SGD [14], initializing the model with default weights and updating it as more running logs are collected.

(1) Initialization. Given $\mathcal{A}$ without historic running logs, we set

$$C_{\mathcal{A}}(F_i, p_i) = \sum_{v\in F_i} \frac{d^+_i(v)}{\min\{p_i, \lfloor d_i \rfloor\}},$$

where $d^+_i(v)$ denotes the out-degree of vertex v in subgraph F_i. We heuristically take this as pre-trained weights for incremental SGD. Intuitively, without prior knowledge about $\mathcal{A}$, we assume that the cost of $\mathcal{A}$ on F_i is proportional to the number of edges in F_i, and that operations in $\mathcal{A}$ are fully parallelized with EMap and VMap.

(2) Sample collection. We collect training data for each run of $\mathcal{A}$ over subgraph F_i. A sample includes: (a) the metric variables $x_i(v)$ of each active vertex v in F_i, which "characterize" their individual topological structures; (b) the thread allocation p_i; and (c) its measured computational cost t_i. After rounds of $\mathcal{A}$ execution, we batch the new samples and feed them into the model.

(3) Training. Using mean square error as the loss function, incremental training computes the gradients and updates the model accordingly. It iterates through the new training data only, and we do *not* shuffle the new with the previous seen samples. The model is continuously trained and updated at backend at a small cost.

Remark. We have experimented with $C_{\mathcal{A}}$ and two alternative cost models: (a) a polynomial regression model with fewer metric variables and (b) a multi-layer perceptron. We find that (1) once trained for algorithm $\mathcal{A}$, $C_{\mathcal{A}}$ works well on graphs of the same type, echoing the observation of [25]; and (2) compared to model (a) and (b), $C_{\mathcal{A}}$ strikes a better balance among accuracy, generalizability, and training cost. Due to lack of space, we present the results in [7].

Problem formulation. With $C_{\mathcal{A}}$, we model the scheduling problem as an optimization problem. Given subgraphs $F_0, F_1, \ldots, F_{n-1}$ of G and an m-core machine with memory size η, it is to find an optimal schedule $\mathcal{S} = (\bar{t}, \bar{p})$, where $t_i \in \bar{t}$ (resp. $p_i \in \bar{p}$) denotes the start time (resp. thread allocation of W_i) for $i \in [0, n)$. Its objective function is

$$\arg\min_{\mathcal{S}} \max_{i \in [0,n)} \{t_i + C_{\mathcal{A}}(F_i, p_i)\}. \quad (3)$$

It is to minimize the *makespan* of $\mathcal{S}$, *i.e.,* the overall completion time of all workers. At any point, a *valid* schedule is subject to three constraints: (1) the consumed memory cannot exceed η; (2) the total thread allocation cannot exceed m; and (3) the start time t_i cannot be set before F_i is fully loaded into the memory, for all i in $[0, n)$. Note that the final constraint takes into account the I/O cost under the memory capacity η. It models the behavior of Loader (see Section 3), which loads fragments continuously as long as η is not exhausted.

Its decision problem, denoted by DSP, is to decide, given the input and a deadline B, whether there exists a valid schedule $\mathcal{S}$ with makespan at most B. The problem is intractable; it subsumes the NP-complete problem of *malleable parallel task scheduling* [48].

Theorem 1: DSP *is NP-complete.* □

Proof sketch: DSP is in NP since one can guess a schedule and check in PTIME whether it is valid and its makespan is at most B. We show that it is NP-complete by reduction from the set partition problem, which is known NP-complete [31] (see [7] for a proof). □

Scheduling strategy. Due to the intractability, Scheduler adopts a lightweight strategy. Offline for each subgraph F_i, it collects its size information and assigns F_i a number $\hat{p}_i$. Then at runtime, it loads pending subgraphs and launches new workers greedily, such that at least $\hat{p}_i$ threads are allocated to W_i. If idle threads exist with no available worker to launch, they are reallocated to active workers.

Intuitively, we strike a balance between inter-subgraph parallelism and intra-subgraph parallelism. It ensures that each subgraph is allocated at least $\hat{p}_i$ threads. Then whenever a set T of threads gets freed, we prioritize pending subgraphs F_i if $|T| \geq \hat{p}_i$. We set $\hat{p}_i$ heuristically to estimate the optimal allocation for worker W_i,

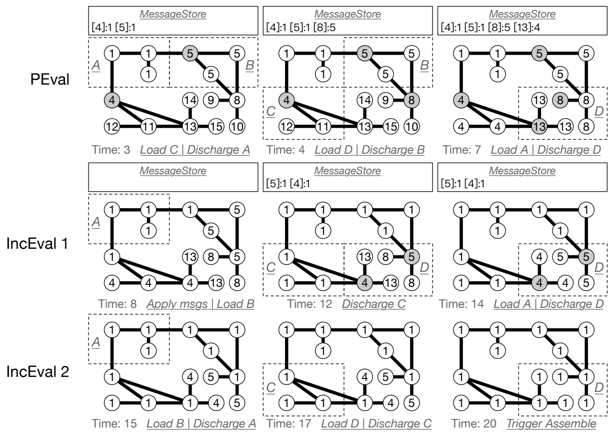

PEval
IncEval 1
IncEval 2
MessageStore
[4]:1 [5]:1
[4]:1 [5]:1 [8]:5
[4]:1 [5]:1 [8]:5 [13]:4
[5]:1 [4]:1
A
B
C
D
Time: 3 Load C | Discharge A
Time: 4 Load D | Discharge B
Time: 7 Load A | Discharge D
Time: 8 Apply msgs | Load B
Time: 12 Discharge C
Time: 14 Load A | Discharge D
Time: 15 Load B | Discharge A
Time: 17 Load D | Discharge C
Time: 20 Trigger Assemble

Figure 3: WCC on G with the two-level parallel execution model.

given the CPU and memory constraint. This strategy satisfies $\hat{p}_i$ for W_i greedily and thus approximates the optimal schedule.

Tentative resource allocation. Scheduler allocates resources based on the subgraph size and the memory size η. It sets $\hat{p}_i = \lceil \frac{s(F_i)}{\eta} m \rceil$ for worker W_i, where $s(F_i)$ is the binary size of F_i. Intuitively, $\hat{p}_i$ indicates the *minimum* number of threads for W_i to process F_i, to strike a balance between computing and memory resources. Following *multi-resource scheduling* [34], all threads are allocated *before* the memory gets exhausted, prioritizing CPU utilization.

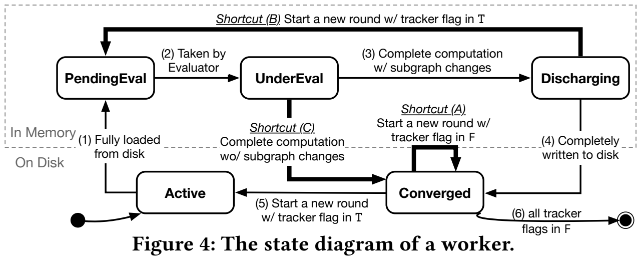
Greedy subgraph processing. At runtime, Scheduler keeps track of a list of pending subgraphs, sorted by $C_{\mathcal{A}}(F_i, \hat{p}_i)$, the estimated computational cost given their tentative thread allocations. To select one for processing, it prioritizes the most computationally heavy one whose tentative allocation can be satisfied. When Scheduler is triggered by an event that t threads are freed by a worker, it is to load F_i with the highest $C_{\mathcal{A}}(F_i, \hat{p}_i)$ as long as $\hat{p}_i \leq t$.

The intuition behind such ordering is to mitigate stragglers at the end of each superstep. Considering the sublinear speedup of intra-subgraph parallelism (Equation 2), prioritizing larger subgraphs can further promote inter-subgraph parallelism, since we have a better chance of overlapping them with more and shorter loads.

Intra-subgraph parallelism. When the unallocated threads cannot satisfy the tentative allocation of any pending subgraph, Scheduler allocates these free threads to active workers, one at a time, as it thrives to keep all CPU cores busy. It decides the receiving worker based on cost analysis. (1) With pending subgraphs, it estimates $\Delta_i = C_{\mathcal{A}}(F_i, p_i) - C_{\mathcal{A}}(F_i, p_i + 1)$ for all active workers, and selects W_j with the highest Δ_j, *i.e.,* W_j gets the most speedup from the extra thread. (2) With no subgraph pending processing for the round, it selects W_j with the maximum $C_{\mathcal{A}}(F_j, p_j)$ to improve the straggler.

Such reallocations are temporary; whenever a new worker W_j becomes active, these available threads are retracted by Scheduler and reassigned to W_j to meet its demand of $\hat{p}_j$ threads.

Example 5: Continuing with Example 3, we compute WCC over G (Figure 1a) on a 4-core machine. Suppose that each subgraph takes 1 time unit to load/write, and each message passing takes 1 time unit. The tentative allocation assigns 2 threads to each worker. As shown in Figure 3, WCC passes all 32 message in 19 time units, leveraging both pipelined processing and two-level parallelism. □


Shortcut (B) Start a new round w/ tracker flag in T
PendingEval
(2) Taken by Evaluator
UnderEval
(3) Complete computation w/ subgraph changes
Discharging
In Memory
On Disk
(1) Fully loaded from disk
Shortcut (C) Complete computation w/o subgraph changes
Shortcut (A) Start a new round w/ tracker flag in F
(4) Completely written to disk
Active
(5) Start a new round w/ tracker flag in T
Converged
(6) all tracker flags in F
Figure 4: The state diagram of a worker.

5.2 System Optimizations

We now present the unique optimization strategies of MiniGraph.

Recall Example 5. In Figure 3, Subgraph A is not updated in both IncEval steps, and its worker receives no messages (updates); thus it requires no further processing, and there is no need to even load Subgraph A. To reduce the unnecessary I/O for loading/discharging Subgraph A, MiniGraph employs (a) a list $\mathcal{M}$ of flags that helps track message exchanges among workers, and (b) a lightweight state machine for modeling the progress of each worker. Both $\mathcal{M}$ and state machine are maintained by StateManager.

Message tracking. StateManager builds a list $\mathcal{M}$ of flags, one for each worker to indicate whether the worker has any messages. It sets $\mathcal{M}[i]$ true if worker W_i has at least one pending update to pull from MessageStore. Otherwise, W_i requires no incremental work to be done, and thus IncEval can be safely skipped in the next round.

Example 6: Continuing with Example 5, the step-by-step WCC execution is shown in Figure 3. Before starting Step IncEval 1, the flags are [F,T,T,T] for Subgraph A, B, C and D, respectively. Note that Subgraph A receives no messages in this round, and hence its flag is F. Similarly, the flags are [F,F,F,T] for Step IncEval 2. □

Worker states and state transitions. We use a finite state machine to model the progress at each worker W_i, and flag $\mathcal{M}[i]$ to trigger state transitions of W_i. As shown in Figure 4, at any point, a worker is in one of the five states: Active, Converged, PendingEval, UnderEval, and Discharging. The first two indicate that the corresponding subgraph is on-disk, while the rest are in-memory states.

For a PEval or IncEval round, worker W_i traverses all five states. As shown in Figure 4, it starts from Active, indicating that W_i requires further computation (*e.g.,* with unconsumed updates). Then, (1) after its subgraph data F_i is loaded into the memory by Loader, W_i becomes PendingEval, meaning that W_i is ready to run and requests thread allocation; (2) W_i turns to UnderEval, after it is taken by Evaluator and is under active computation; (3) when W_i concludes with changes made to F_i, its metadata (*e.g.,* status variables, partial results) and messages aggregated in MessageStore, W_i is set to Discharging; and (4) once F_i is fully persisted on disk, Discharger releases its memory space and sets W_i to Converged, indicating that it is done for the round. MiniGraph concludes the round when all workers get past UnderEval. If there are messages cached in MessageStore, MiniGraph (5) resets Converged workers back to Active, and starts a new round of (IncEval) computation. Otherwise, MiniGraph (6) calls Assemble to produce the final results.

Shortcuts in state transitions. Under certain conditions, some states in a round of computation can be skipped without affecting the correctness. That is, MiniGraph can take some "shortcuts" in state transitions, and reduce unnecessary computation and I/O. Below are example shortcuts identified by StateManager, marked as bold arrows in Figure 4. Consider worker W_i with its flag $\mathcal{M}[i]$.

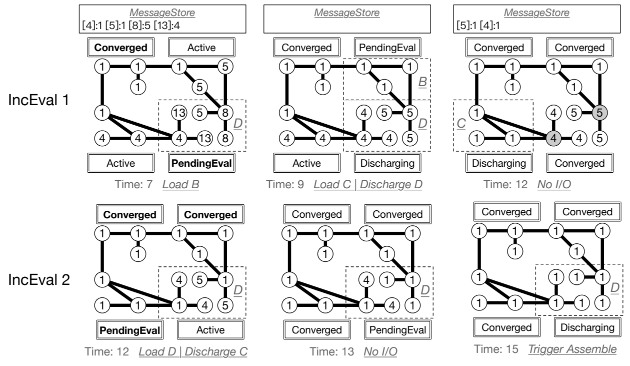


Figure 5: Optimized WCC execution on G with shortcuts.

Shortcut (A). To start of a round of IncEval computation, the state of W_i is typically reset to Active following transition (5). If $\mathcal{M}[i]$ = F, however, we may keep W_i in Converged state since it requires no further processing. That is, we can skip handling subgraph F_i in the round. This shortcut is frequently exploited when, *e.g.,* the input graph is not well-connected and F_i is "isolated".

Shortcut (B). MiniGraph starts a new round of IncEval as soon as all workers finish the current round (*i.e.,* get past UnderEval). If W_i is still in Discharging, *i.e.,* if its changed F_i is not yet fully persisted onto the disk, W_i is set to PendingEval directly, such that it starts the new round without going through the disk. This shortcut is exploited at the end of each round and substantially reduces I/O.

Shortcut (C). When W_i completes with no changes to F_i or its metadata, W_i skips Discharging and is set to Converged directly, avoiding redundant disk writes. This shortcut is effective for, *e.g.,* SSSP.

Example 7: Continuing with Example 5–6, we show execution of WCC in Figure 5 by exploiting shortcuts in state transitions (in bold state labels). PEval execution remains the same as in Figure 3.

The start of Step IncEval 1 (Time 7) sees two shortcuts: (1) worker A takes Shortcut (A) to skip computation entirely; and (2) worker D skips discharging/loading by taking Shortcut (B); it enters PendingEval directly for the next round of IncEval. Because of the two shortcuts, Step IncEval 1 takes 5 time units to complete (Time 7–12), as opposed to 7 time units (Time 7–14) in Example 5.

Similarly, we optimize Step IncEval 2. It takes 3 time units (Time 12–15), a 40% reduction from 5 units (Time 14–19) in Example 5. □

6 EXPERIMENTAL EVALUATION

Using real-life graphs, we evaluated MiniGraph for its (1) efficiency, (2) scalability, (3) resource scheduling and (4) performance.

Experimental setup. We start with the settings.

Datasets. We used five real-life and synthesized datasets (Table 2). Among these, `datagen`, `clueWeb` and `hyperlink12` exceed the memory capacity of our testbed. Here `datagen` is a synthesized dataset in the LDBC [39] benchmark suite; and `hyperlink12` is sampled from `hyperlink` [2] via BFS sampling [38]. We used edge-cut [47] as the default partitioner, and tested other partitioners.

We report additional results over three small real-life graphs of different types (*e.g.,* road network, network topology) in [7].

Baselines. We evaluated three out-of-core systems as baselines: GridGraph [83], GraphChi [45] and XStream [62]. GridGraph is the state-of-the-art, assuming an off-the-shelf hardware platform. We did not test Mosaic [51] since it requires an Intel Xeon Phi coprocessor,

Name	Type	$\|V\|$	$\|E\|$	MaxDegree	Raw Data
`friendster` [6]	social network	65.6M	1.8B	5124	30.14GB
`web-sk` [61]	Web	50M	1.9B	8.5M	32GB
`datagen` [10, 39]	synthetic	29M	2.6B	2288	75GB
`clueWeb` [61]	Web	1.7B	7.9B	6.4M	137GB
`hyperlink12` [2]	Web	7B	7.6B	5.1M	155GB

Table 2: Graph datasets.

which has been discontinued since 2018 [55]. We also omitted in-memory systems (*e.g.,* [50, 57, 63, 78, 80]) since MiniGraph targets applications when the input graph cannot fit into the memory.

We tested four variants of MiniGraph: (1) MiniGraph$_{\text{Seq}}$, which exposes PIE as interface without EMap and VMap (Section 4). (2) MiniGraph$_{\text{NoShort}}$, which disables shortcuts (Section 5.2). (3) MiniGraph$_{\text{VCut}}$ and MiniGraph$_{\text{HCut}}$, which use vertex-cut [5] and hybrid-cut [19], respectively, as graph partitioner.

We also compared with GraphScope [24] and Gluon [21], which use multiple machines, not a single machine as MiniGraph.

Algorithms and queries. We ran PIE+ programs for WCC, SSSP, PR, BFS, RW and Sim (Section 4) [7]. The first five are included in various benchmarks [8, 13]. For baselines, we used their out-of-box implementations. No baseline supports Sim out-of-box. We implemented the VC-based Sim algorithm of [29] on GraphChi. However, the algorithm cannot be implemented on GridGraph or XStream without changing their internal working, because a necessary global storage is not supported by the APIs of either system.

For each input graph for SSSP and BFS, we randomly picked 10 vertices and used them as sources. For RW, we initiated 100 walkers randomly, each taking at most 10 steps. For Sim, we randomly generated 20 query patterns, each of which has 6 vertices and 10 edges.

We deployed the single-machine systems on a workstation running Ubuntu Server 20.04 LTS, powered with an Intel Core i9-7900X CPU @3.30GHz and 64GB of DDR4-2666 memory. The CPU has a 13.75MB LLC, shared among 10 cores (20 hyperthreads).

Unless stated otherwise, we set n partitions as reported in Table 4 such that we have $\hat{p}_i = 4$ by default (see Section 5). We set a different n for PR (*s.t.* $\hat{p}_i = 10$), guided by our experiments with varying n (see [7] for details). Graphs are loaded from a 1TB WD Blue WDS100T2B0A SATA SSD, whose average sequential read throughput is 560MB/s. Each experiment was repeated 5 times; we report the average here. We report results for some algorithms on some graphs; the other results are consistent and are reported in [7].

Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluated the efficiency and I/O of MiniGraph versus out-of-core systems. Some experiments are conducted under a *memory budget*, *i.e.,* the ratio of the available memory to the size of the raw input. For instance, a 50% memory budget means that exactly *half* of the graph can be held in memory. We used Linux `cgroups` to enforce the memory budget, similar to [11].

SSSP. For SSSP over different graphs, Table 3 reports the number of supersteps, the volume of disk read traffic, I/O reduction from shortcuts, the average CPU utilization, the correlation coefficient r between real-time I/O and CPU usage, and the CPU cache hit rate. Table 3 only shows comparison with GridGraph, the best performing baseline. Table 4 shows the performance of all the systems.

Dataset	Metric	SSSP		WCC		PR	
		MiniGraph	GridGraph	MiniGraph	GridGraph	MiniGraph	GridGraph
`friendster`	# Supersteps	**8**	32	**6**	21	8	10
	Disk Read (GB)	**78**	115.1	**50**	135	**107**	160
	Shortcut I/O (GB)	**-12**	N/A	**-22**	N/A	**-10.4**	N/A
	Avg. CPU Util.	**33.74%**	4.45%	**48.2%**	6.83%	**68.46%**	62.38%
	I/O-Compute Corr.	**0.095**	-0.113	**0.163**	-0.202	**0.185**	-0.156
	Cache Hits	**45.33%**	9.59%	**48.25%**	12.04%	**34.8%**	**36.2%**
`web-sk`	# Supersteps	**10**	63	**9**	120	15	20
	Disk Read (GB)	**112.5**	232	**42**	367	**87**	232
	Shortcut I/O (GB)	**-30.9**	N/A	**-24.6**	N/A	**-133.9**	N/A
	Avg. CPU Util.	**15.76%**	5.83%	**25.04%**	5.16%	42%	**42%**
	I/O-Compute Corr.	**0.008**	0.003	**0.013**	0.009	**0.082**	-0.039
	Cache Hits	**50.89%**	6.37%	**37.42%**	11.63%	**50.22%**	46.04%

Table 3: Runtime statistics for SSSP, WCC and PR.

(1) MiniGraph performs the best over all workloads. Over two real-life large `clueWeb` and `hyperlink12`, MiniGraph performs substantially better than all the baselines. With the memory capacity being only 47% (resp. 41%) of `clueWeb` (resp. `hyperlink12`), it outperforms GridGraph, by 4.6× (resp. 3.2×). Moreover, GraphChi and XStream are not able to handle the two large graphs (Table 4). Over the large synthetic `datagen`, the improvement is relatively moderate (1.5×), since the graph is extremely sparse and the VC of GridGraph does not incur significant I/O. The results are consistent over small graphs, which justifies memory budgeting, a common practice for benchmarking out-of-core systems (*e.g.,* [11]).

(2) With a 50% memory budget on `friendster`, MiniGraph beats GridGraph, GraphChi and XStream by 1.5×, 2.7× and 15.2×, respectively. It is 2.8×, 3.5× and 28.9× faster on `web-sk`, a better improvement, because `web-sk` is sparser and has a larger diameter. As remarked earlier, the VC of the baselines supports within-neighborhood operations only, and requires more supersteps to converge on graphs with larger diameters. GridGraph, *e.g.,* takes 32 supersteps on `friendster` and 63 on `web-sk`. In contrast, MiniGraph supports beyond-neighborhood computations via its hybrid parallel model. It takes 8 supersteps on `friendster` and 10 on `web-sk`. This justifies the need for GC and two-level parallelism.

Note that over `friendster`, GridGraph takes 4× the supersteps of MiniGraph, while its execution time is 2.7×. The main reason lies in their different parallel models; a GC superstep is quite different from a VC superstep as it processes a subgraph. MiniGraph generally takes more computationally heavy supersteps than GridGraph does. This said, its superstep progresses the computation further with each superstep, and reduces the overall work required.

(3) MiniGraph also substantially reduces disk I/O (Table 3). It generates 32.3% and 51.6% less disk read than GridGraph on `friendster` and `web-sk`, respectively. More specifically, (a) shortcuts account for 32.3% and 25.8% of the I/O reduction on the two graphs, respectively. (b) The rest of I/O reduction roots in the reduced supersteps.

(4) The average CPU utilization of MiniGraph is 29.3% (resp. 9.9%) higher than GridGraph on `friendster` (resp. `web-sk`). GridGraph exhibits negative correlations ($r < -0.1$) between I/O and CPU usage, indicating that CPU may starve when the system is busy at loading data. In contrast, it is $0 < r < 0.1$ for MiniGraph; that is, its I/O does not block computations. These justify the pipelined architecture.

(5) MiniGraph has a much better CPU cache locality than GridGraph. It achieves a 45% cache hit rate on `friendster` and 51% on `web-sk`, while it is 10% for GridGraph at best (Table 3). This is because MiniGraph supports inter-subgraph parallelism, while VC systems (*e.g.,* GridGraph) frequently access all neighbors of each vertex, which inevitably incurs random reads across the entire graph.

Data	Memory Budget	#Partitions (PR/Others)	SSSP				WCC				PR			
			MiniGraph	GraphChi	GridGraph	XStream	MiniGraph	GraphChi	GridGraph	XStream	MiniGraph	GraphChi	GridGraph	XStream
`friendster`	50% (15.07GB)	4/10	**201.8**	535 (2.7×)	293.1 (1.5×)	3061 (15.2×)	**171.8**	1636 (9.5×)	204.7 (1.2×)	2037 (11.8×)	**190.104**	450.7 (1.9×)	485.3 (1.9×)	2685 (11.3×)
`web-sk`	50% (16GB)	4/10	**326.4**	1140 (3.5×)	917.9 (2.8×)	9437 (28.9×)	**172**	620.1 (3.6×)	704.6 (4.1×)	4056 (23.5×)	**248.3**	2288 (9.2×)	395 (1.6×)	2903 (11.7×)
`datagen`	85% (64GB)	4/4	**85**	2483.4 (29×)	127.1 (1.5×)	2350 (27.6×)	**80.4**	2688.6 (33×)	182.35 (2.3×)	6124 (76.1×)	**306.4**	1104.03 (3.6×)	1066 (3.5×)	3434.7 (11.2×)
`clueWeb`	47% (64GB)	4/10	**2514**	/	11534 (4.6×)	/	**2742**	/	11665 (4.3×)	/	**2022**	/	3803 (2.1×)	/
`hyperlink12`	41% (64GB)	4/4	**150.1**	/	475.6 (3.2×)	/	**1940**	/	12570 (6.5×)	/	**1553.3**	/	5209 (3.4×)	/

Table 4: Execution time for SSSP, WCC and PR (in seconds). "/" denotes that the experiment could not finish within 4 hours.

(6) MiniGraph also performs substantially better than its variants. We defer a detailed discussion to the case of WCC below.

WCC. As shown in Tables 3–4, MiniGraph beats all the baselines for WCC over all graphs. (1) It is up to 6.5×, 33.0× and 76.1× faster than GridGraph, GraphChi and XStream, respectively; note that for GraphChi and XStream, 33.0× and 76.1× do not aggregate results on `clueWeb` and `hyperlink12`, since the two systems could not handle the two large graphs (Table 4). (2) Over `web-sk`, it takes only 7.5% supersteps and 28.9% of disk read of GridGraph; moreover, it has better average CPU utilization (+41.4%) and cache locality (+36.2%). These lead to 4.1× performance improvement. (3) MiniGraph incurs 62.9% (resp. 88%) less disk read on `friendster` (resp. `web-sk`), with 16.3% (resp. 11.4%) reductions from shortcuts.

MiniGraph consistently beats its variants. As shown in Figure 6a, (a) It speeds up MiniGraph$_{\text{Seq}}$ by 61.0% and 78.1% on `friendster` and `web-sk`, respectively. This shows the benefit of its hybrid model that exploits intra-subgraph parallelism to improve multi-core parallelism. The hybrid model works better on denser graphs, which often allow a higher inherent intra-subgraph parallelism, to which EMap and VMap are more effective. (b) MiniGraph beats MiniGraph$_{\text{NoShort}}$ by 1.18× on average. The improvement is moderate, since all shortcuts apply to IncEval only, while PEval is the most costly for WCC. They reduce I/O by 44% on average, yet the effect gets diluted by other factors, *e.g.*, stragglers. (c) Shortcuts are less effective over well-connected graphs (*e.g.*, a 12% improvement over `friendster`), but are better over more "isolated" `web-sk` (36%).

Figure 6b shows the disk I/O and CPU utilization of MiniGraph and GridGraph over `clueWeb`. It verifies that MiniGraph overlaps I/O and CPU operations, and justifies its pipelined architecture.

PR. As reported in Table 4, (1) MiniGraph is 1.6–3.5× faster than GridGraph over all graphs, the best-performing baseline, although PR fits its VC model. (2) On `friendster` (resp. `web-sk`), MiniGraph incurs 33.1% (resp. 45.6%) less disk read traffic than GridGraph, with 19.6% (resp. 28.6%) reductions from shortcuts. These are higher than SSSP and WCC, since we use larger fragments in PR.

Sim. As shown in Figure 6c for Sim on `friendster` and `web-sk`, on average MiniGraph beats GraphChi by 2.2× and 1.6×, and reduces I/O traffic by 81% and 77.4%, respectively. One reason is that its hybrid model can support a more efficient Sim algorithm (see Appendix A.1 of [7]). Its complexity is $O((|E| + |V|)(|E_Q| + |V_Q|))$, where $|E|$ and $|V|$ (resp. $|E_Q|$ and $|V_Q|$) denote the size of edges and vertices of G (resp. pattern Q), respectively. In contrast, GraphChi can only use the Sim algorithm [29] under VC and take $O(|E|^2(|V_Q| + |E_Q|))$ time. This further justifies the need for supporting GC.

BFS and RW. Figure 6d reports the speedup of MiniGraph for BFS and RW versus GridGraph. Consistent with the other algorithms, MiniGraph beats all the competitors. For BFS, MiniGraph is 1.2–1.7× faster than GridGraph over various graphs. It is at least 1.5× better over more skewed Web graphs (*i.e.*, `web-sk` and `clueWeb`). Although the GC and VC algorithms for RW have similar complexity bounds, the improvement is still quite evident (52.5% on average).

HDD vs. SSD. We also tested the impact of using HDD as the secondary storage. When replacing SSD with a 3.6TB Seagate HDD, the average read throughput drops from 560MB/s to 125MB/s, and all the systems get slower, as expected. On `friendster`, *e.g.*, it is slowed by 2.6×, while it is 3.3× and 2.9× for GridGraph and GraphChi, respectively. This is because (a) MiniGraph incurs less I/O traffic (see Table 3), and (b) it issues a small number of bulk I/O requests, and works well with HDD that has a high seek cost.

Exp-2: Scalability. Under 50% memory budget, we evaluated the scalability of MiniGraph with the size $|G|$ of graphs G and the number m of CPU cores. We report the results of WCC and PR; the results of the other algorithms are consistent.

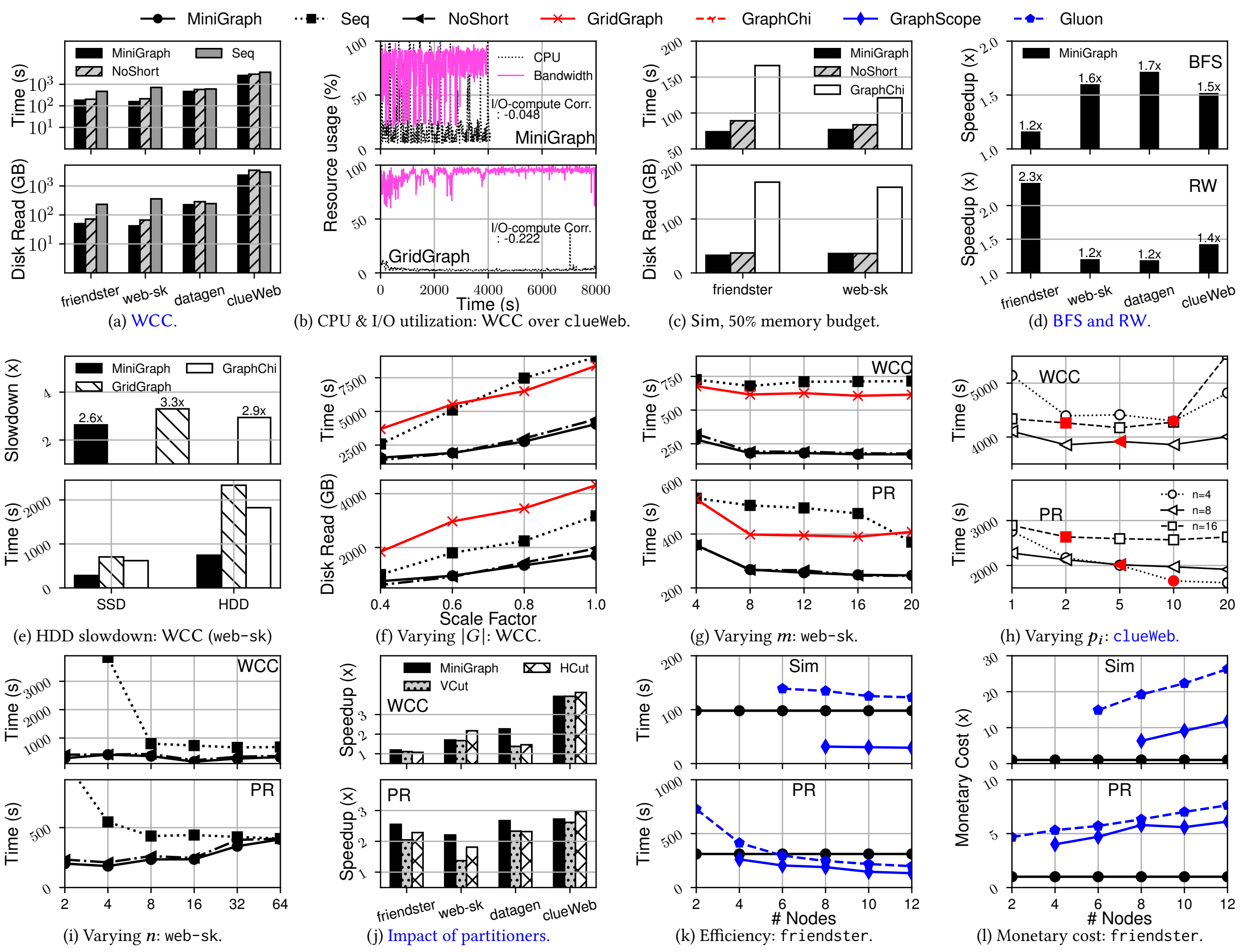
Varying $|G|$. We sampled graphs G from large `clueWeb` using Edge Sampling [38], with a scale factor δ that controls the fraction of edges to be sampled. As shown in Figure 6f when varying δ from 0.4 to 1.0 for WCC, (1) MiniGraph scales well with $|G|$. It takes 2.3× longer, while it is 2.6× for GridGraph. (2) MiniGraph also scales better than its variants MiniGraph$_{\text{Seq}}$ (3.5×) and MiniGraph$_{\text{NoShort}}$ (3.16×). These further verify the effectiveness of our hybrid parallel model and optimization strategies. (3) When $|G|$ grows 2.5×, MiniGraph incurs 2.29× I/O traffic, by its shortcut optimizations.

Varying m. Varying the number m of cores from 4 to 20, we ran WCC and PR on `web-sk`. As shown in Figure 6g, (1) MiniGraph scales well with m. (2) It scales better than GridGraph, which does not improve much (up to +4.3%) when m varies from 8 to 20. This is because GridGraph becomes I/O-bound when $m \geq 8$ (see also our findings in Figure 6b). (3) It also scales better than MiniGraph$_{\text{Seq}}$, which barely improves when $m \geq 4$ for the lack of intra-subgraph parallelism. These also justify the need for a hybrid parallel model.

Exp-3: Resource scheduling. We evaluated the scheduling strategy by testing MiniGraph in different settings of (1) thread allocation p_i, (2) the number n of partitions, and (3) graph partitioners.

Varying p_i. We first tested the impact of p_i, the number of threads allocated to processing a subgraph. Over `clueWeb`, Figure 6h reports how WCC and PR works with varying p_i. For each graph partitioning, we mark its tentative allocation $\hat{p}_i$ (Section 5) with colored points. As shown there, $\hat{p}_i$ strikes a balance between the two levels of parallelism and improves the overall multi-core parallelism, by selecting a near-optimal value for p_i based on n and G.

Varying n. The number n of partitions impacts the inter-subgraph parallelism of MiniGraph. Varying n from 2 to 64 on `web-sk`, Figure 6i shows its impact on WCC and PR. (1) For VC-based PR, its optimal n is 4, which minimizes fragmentation while keeping the



(a) WCC.

(b) CPU & I/O utilization: WCC over clueWeb.

(c) Sim, 50% memory budget.

(d) BFS and RW.

(e) HDD slowdown: WCC (web-sk)

(f) Varying $|G|$: WCC.

(g) Varying m: web-sk.

(h) Varying p_i: clueWeb.

(i) Varying n: web-sk.

(j) Impact of partitioners.

(k) Efficiency: friendster.

(l) Monetary cost: friendster.

Figure 6: Efficiency, scalability, and resource scheduling of MiniGraph, and its performance vs. multi-machine systems.

pipeline working. (2) MiniGraph$_{\text{Seq}}$ is very slow under a small n, as its CPU usage is limited by the low inter-subgraph parallelism. (3) For GC-based WCC, the optimal n is 16. Finding this optimum value requires a careful cost analysis (Section 8).

Impact of partitioners. We tested WCC and PR using different partitioners with $n = 4$. As reported in Figure 6j, MiniGraph beats all the baselines regardless of the partitioner used. Running PR on large clueWeb, for example, MiniGraph$_{\text{VCut}}$ and MiniGraph$_{\text{HCut}}$ outperform GridGraph by 2.71× and 2.98×, respectively.

Exp-4: Single-machine vs. multi-machine. We also evaluated the capacity of MiniGraph for graph analytics with a single machine versus multi-machine systems GraphScope and Gluon. To make a uniform testing environment, we deployed them in the cloud.

Execution time. Figure 6k reports the performance of the systems for Sim and PR over friendster. (1) For Sim, we used 8-vCPU 64GB-memory instances for all three systems since GraphScope and Gluon require more working memory. Gluon (resp. GraphScope) runs out-of-memory with fewer than 6 (resp. 8) nodes. MiniGraph, using a single instance, outperforms Gluon with up to 12 nodes by 29.3–46.3%. This further justifies the need for GC. While GraphScope takes less computation time, it requires an additional 200+s for preprocessing, which is not counted in Figure 6k. (2) For PR, MiniGraph runs on a single 8-vCPU 32GB-memory instance. Gluon uses a varying number of instances of the same configuration. GraphScope uses multiple 8-vCPU 64GB-memory instances, because it requires more than 32 GB in all cases of the experiment. Yet it still runs out-of-memory with 2 nodes. We find that MiniGraph performs comparably to a 4-node (resp. 6-node) deployment of GraphScope (resp. Gluon), and beats Gluon with fewer than 4 nodes.

Cost effectiveness. Figure 6l depicts the relative monetary cost of multi-machine systems for running Sim and PR over friendster, taking the cloud spending of MiniGraph as baseline. (1) While Gluon and GraphScope get faster with more nodes, they spend more. This echoes the observation of [53] that multi-machine parallelism is not always cost-effective. (2) GraphScope (resp. Gluon) pays at least 5.3× (resp. 13.9×) more than MiniGraph for Sim, and costs 3.0× (resp. 3.7×) more for PR. This further justifies the need for a single-machine system to save cost for small companies.

Summary. We find the following. (1) With the pipelined architecture and two-level parallelism, MiniGraph consistently outperforms the prior single-machine systems under all out-of-core workloads. It is up to 6.5×, 33.0× and 76.1× faster than GridGraph, GraphChi and XStream, respectively; moreover, it can handle large graphs that exceed the capacity of GraphChi and XStream. (2) Under BSP, it requires only a fraction of supersteps (<29%) and disk read traffic (<53.3%) of GridGraph for SSSP and WCC. (3) It improves the CPU utilization of GridGraph, the best-performing baseline, by up to 41.4%. (4) It scales well with graphs and the number of CPU cores. It runs all the algorithms on hyperlink12 of 155GB within 32 minutes, while no baseline finishes in 1.4 hours. (5) MiniGraph incurs less degradation when switching from SSD to HDD. (6) Its shortcut optimization effectively reduces I/O cost, especially on dense graphs. (7) Its two-level execution model balances inter-subgraph and intra-subgraph parallelism. (8) MiniGraph works better than Gluon with 12 machines on Sim, and saves the monetary cost of multi-machine systems from 3.0× to 13.9×.

7 RELATED WORK

We categorize the related work as follows.

Multi-machine systems. A number of multi-machine systems have been developed, to support big graph analytics by scaling out, *e.g.*, [3, 21, 27, 33, 35, 52, 56, 69, 72, 76, 82, 82]. Such systems adopt a shared-nothing architecture: they partition the input graph, and distribute the fragments to workers; all workers process their local fragments in-memory in parallel, and communicate with each other via message passing. Most of these systems adopt the VC model, except that GRAPE [27] proposes and supports GC. GraphScope [4, 24] extends GRAPE and can operate with either GC or VC.

In contrast to scaling out with multiple machines, (1) MiniGraph uses a single machine and explores multi-core parallelism. It aims to provide small businesses with a capacity of big graph analytics under limited resources. (2) It adopts a shared-memory architecture and employs the secondary storage as the memory extension. These introduce new challenges, *i.e.,* the memory constraint and I/O cost. (3) It develops (a) a pipelined architecture to mitigate I/O cost and improve CPU utilization, and (b) an efficient synchronization scheme among workers by using a shared data structure. (4) Extending GC from multi-machine to multi-core, MiniGraph proposes a hybrid parallel model. It integrates VC and GC in a unified interface to enrich inter-subgraph GC parallelism with intra-subgraph VC parallelism, as opposed to GraphScope, which supports VC and GC as two separate sets of programming interfaces. (5) It develops a two-level execution model to support its intra- and inter-subgraph parallelism. Under the memory constraint of a single machine, this better utilizes the multi-core resources. Moreover, it introduces an efficient method for a new (intractable) resource scheduling problem. (6) MiniGraph explores new optimization methods to skip computation rounds, and hence reduce unnecessary CPU and I/O.

Single-machine in-memory systems. When graphs are small enough to fit into the memory of a single machine, in-memory systems [50, 57, 63, 78, 80] aim to conduct computations with high performance. Such systems focus on optimization techniques on memory-resident graphs, *e.g.,* efficient task parallelization [57, 63, 80] and improved data locality [78]. Ligra [63] introduces a variant of VC model that allows user-specified parallelization of operations. Polymer [78] is designed for a NUMA architecture, where the memory access latency of a processor is not uniform across the address space.

In contrast, MiniGraph targets large graphs that exceed the memory capacity of a single machine. It performs iterative out-of-core computation, by actively swapping graph data between the memory and the disk. It deals with inevitable and possibly prohibitive I/O costs, a challenge that is not encountered by in-memory systems.

Single-machine out-of-core systems. Closer to this work are out-of-core systems [11, 32, 45, 51, 62, 71, 75, 83]. Most of the systems adopt VC and BSP. The techniques have mostly focused on reducing I/O, the major performance bottleneck. XStream [62] proposes stream reading of edges to maximize sequential disk accesses. GraphChi [45] divides the input data into small shards, each having a disjoint set of independent edges; it employs Parallel Sliding Windows such that shards in the window are allowed in memory, while those being pushed out are written onto the disk. Vora *et al.* [71] propose online adjustment of sharding, which avoids loading unneeded edges at the price of extra computation. GridGraph [83] improves the locality of edge sharding via 2-level partitioning based on source and destination IDs, and enables selective scheduling of data. Clip [11] allows asynchronous and repetitive processing of an in-memory shard; however, it must be explicitly invoked by users in their VC code, which might be error-prone. With extra hardware, Gill *et al.* [32] use persistent memory sticks as memory extension, which have higher throughput than SSDs (Intel has discontinued the Optane-only SSDs for the consumer market [64]). Mosaic [51] employs an Intel Xeon Phi coprocessor to accelerate CPU computations.

In contrast, MiniGraph (1) proposes a subgraph-based pipelined architecture to "cancel" I/O costs and promote sequential accesses to disks, to exploit the I/O bandwidth; (2) it supports both GC to simplify parallel programming and speed up beyond-neighborhood computation, and VC to parallelize edge/vertex operations in each subgraph; (3) it proposes a two-level execution model to balance inter-subgraph and intra-subgraph parallelism; and (4) it develops new optimization strategies to reduce I/O. (5) As opposed to [32, 51], it requires no dedicated hardware for storage or computation; it assumes an off-the-shelf machine that can be easily acquired.

Polyglot systems. Polyglot systems [18, 20, 22, 40, 41, 49, 65, 67, 81] take graphs as edge relation views, and cast graph queries into SQL, which requires large memory for intermediate data. In contrast, MiniGraph is a native graph system. It uses a single machine with limited memory, which is often insufficient for polyglot systems.

8 CONCLUSION

The novelty of MiniGraph consists of (1) the first single-machine system that (a) extends GC from multiple machines to multiple cores, and (b) enriches inter-subgraph parallelism (GC) with intra-subgraph parallelism (VC) under a hybrid parallel model; (2) a pipelined architecture to overlap I/O and CPU operations; (3) a two-level execution model to schedule and balance VC and GC; (4) a unique optimization scheme to further reduce I/O. Our experimental study has verified that with a single machine, MiniGraph performs better than some 12-machine parallel systems.

One topic for future work is to extend the architecture of MiniGraph by incorporating GPU and FPGA. We have only considered CPU as a "lowest common denominator" of single-machine systems. When GPU and FGPA are available, we need to revise our workload assignment and graph partitioning strategies. Another topic is to study the optimal graph partitions for a given algorithm.

REFERENCES

[1] 2010. Traffic dataset. http://www.dis.uniroma1.it/challenge9/download.shtml

[2] 2012. Hyperlink. http://webdatacommons.org/hyperlinkgraph/

[3] 2014. Timely Dataflow. https://github.com/frankmcsherry/timely-dataflow/

[4] 2020. GraphScope. https://graphscope.io/

[5] 2022. Apache Spark Partition Strategy. https://spark.apache.org/docs/1.4.0/api/java/org/apache/spark/graphx/PartitionStrategy.RandomVertexCut$.html

[6] 2022. Friendster dataset. https://snap.stanford.edu/data/com-Friendster.html

[7] 2022. Full version. https://shuhaoliu.github.io/assets/papers/minigraph-full.pdf

[8] 2022. Graph500 benchmark specifications. https://graph500.org/?page_id=12#sec-3

[9] 2022. Twitter dataset. http://socialcomputing.asu.edu/datasets/Twitter

[10] 2023. LDBC dataset. https://graphalytics.org/datasets

[11] 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*. 125–137.

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

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

[14] Dimitri P. Bertsekas. 2015. Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey. *CoRR* abs/1507.01030 (2015). http://arxiv.org/abs/1507.01030

[15] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.

[16] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *SIGKDD*. 1456–1465.

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

[18] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-Tran: A high performance distributed graph database with a decentralized architecture. *PVLDB* 15, 11 (2022), 2545–2558.

[19] 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 (TOPC)* 5, 3 (2019), 1–39.

[20] Joana MF da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. 2020. Kaskade: Graph views for efficient graph analytics. In *ICDE*.

[21] 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*.

[22] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR*.

[23] Wenfei Fan. 2022. Big Graphs: Challenges and Opportunities. *PVLDB* 15, 12 (2022), 3782–3797.

[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*. ACM, 1765–1779.

[26] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Trans. Database Syst.* 38, 3 (2013), 18.

[27] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Bohan Zhang, Zeyu Zheng, Yang Cao, and Chao Tian. 2017. Parallelizing Sequential Graph Computations. In *SIGMOD*. ACM, 495–510.

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

[29] Arash Fard, M. Usman Nisar, Lakshmish Ramaswamy, John A. Miller, and Matthew Saltz. 2013. A distributed vertex-centric approach for pattern matching in massive graphs. In *IEEE BigData*.

[30] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM* 34, 3 (1987), 596–615.

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

[32] 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.

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

[34] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *OSDI*. 65–80.

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

[36] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. 1995. Computing Simulations on Finite and Infinite Graphs. In *Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 453–462.

[37] John E. Hopcroft and Robert Endre Tarjan. 1973. Efficient Algorithms for Graph Manipulation [H] (Algorithm 447). *CACM* (1973), 372–378.

[38] Pili Hu and Wing Cheong Lau. 2013. A survey and taxonomy of graph sampling. *arXiv preprint arXiv:1308.5865* (2013).

[39] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. 2016. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB* 9, 13 (2016), 1317–1328.

[40] Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. 2015. Graph analytics using vertica relational database. In *IEEE Big Data*.

[41] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. 2014. Vertexica: Your relational friend for graph analytics! *PVLDB* 7, 13 (2014), 1669–1672.

[42] George Karypis and Vipin Kumar. 1998. Multilevelk-Way Partitioning Scheme for Irregular Graphs. *JPDC* 48, 1 (1998), 96–129.

[43] 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.

[44] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW*.

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

[46] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *SIGKDD*. ACM, 177–187.

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

[48] Walter T Ludwig. 1995. *Algorithms for Scheduling Malleable and Nonmalleable Parallel Tasks*. Ph.D. Dissertation. Department of Computer Sciences, University of Wisconsin-Madison.

[49] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. 2016. G-SQL: Fast query processing via graph exploration. *PVLDB* 9, 12 (2016), 900–911.

[50] 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*. 195–207.

[51] 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*. ACM.

[52] 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*. 135–146.

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

[54] Robin Milner. 1989. *Communication and Concurrency*. Prentice Hall.

[55] 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/.

[56] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *SOSP*. ACM, 439–455.

[57] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*. ACM, 456–471.

[58] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[59] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, and Seif Haridi. 2014. Distributed Vertex-Cut Partitioning. In *DAIS*.

[60] G. Ramalingam and Thomas Reps. 1996. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms* 21, 2 (1996), 267–305.

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

[62] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*. ACM, 472–488.

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

[64] Ryan Smith. 2022. Intel To Wind Down Optane Memory Business - 3D XPoint Storage Tech Reaches Its End. https://www.anandtech.com/show/17515/intel-to-wind-down-optane-memory-business.

[65] Benjamin A Steer, Alhamza Alnaimi, Marco ABFG Lotz, Felix Cuadrado, Luis M Vaquero, and Joan Varvenne. 2017. Cytosm: Declarative property graph queries without data migration. In *GRADES*.

[66] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. In *WSDM*.

[67] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGRAPH: An efficient relational-based property graph store. In *SIGMOD*. 1887–1901.

[68] Nilothpal Talukder and Mohammed J. Zaki. 2016. A distributed approach for graph mining in massive networks. *Data Mining and Knowledge Discovery* 30, 5 (2016), 1024–1052.

[69] 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.

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

[71] 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*. 507–522.

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

[73] Wikipedia. 2022. Stone–Weierstrass Theorem. https://en.wikipedia.org/wiki/Stone%E2%80%93Weierstrass_theorem

[74] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: Time to fuse for distributed graph-parallel computation. In *PPOPP*.

[75] 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.

[76] 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.

[77] 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.

[78] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-Aware Graph-Structured Analytics. *SIGPLAN Not.* 50, 8 (2015), 183–193.

[79] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2014. Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *TPDS* 25, 8 (2014), 2091–2100.

[80] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[81] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-one: graph processing in RDBMSs revisited. In *SIGMOD*. 1165–1180.

[82] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *USENIX OSDI*. 301–316.

[83] 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*. 375–386.

A CASE STUDIES

Example 4 has shown how PIE+ parallelizes WCC. Below we show how to program with PIE+ for problems that fit VC, GC or both.

A.1 PageRank

We first show how to program with PIE+ for PageRank, a problem defined in VC. The PIE+ programs are similar to its PIE counterparts [28], and hence we emphasize the use of EMap and VMap.

As remarked earlier, EMap and VMap support full-fledged VC programming (see [63]). This makes it possible for PIE+ to program an arbitrary VC algorithm, by simply recasting it into EMap and VMap. Consider PageRank (PR [17]), a typical VC program.

PR takes as input a directed graph $G = (V, E)$ (*e.g.,* hyperlinks among Web pages) and a threshold ϵ, and outputs the ranking scores of all vertices in G. For each vertex v in V, its ranking score is:

$$v.\mathsf{rank} = d \cdot \Sigma_{\{u|\langle u,v\rangle \in E\}} \frac{u.\mathsf{rank}}{u.\mathsf{degree}^{+}} + (1-d), \tag{4}$$

where d is a user-defined damping factor and $u.\mathsf{degree}^{+}$ denotes the out-degree of u. This update function is applied iteratively to refine the ranking scores, until the Euclidean distance between the ranks in two consecutive iterations is below the predefined threshold ϵ.

As shown in Algorithm 5, we simply implement Equation 4 with EMap and VMap, and embed them in PEval and IncEval.

PEval *and* IncEval. For each vertex v in V_i of subgraph F_i, PEval (1) declares its status variable v.sink, initialized as 0 (Line 7); (2) initializes v.rank with a random float in (0, 1) by invoking rand() with VMap (Line 1); (3) propagates v.rank to each out-neighbor u, accumulating the scores in u.sink; and (4) updates v.rank by Equation 4 based on the score accumulate in v.sink (Line 2).

Steps (3) and (4) are parallelized via EMap and VMap, with update functions Propagate and UpdateRank, respectively. EMap executes Propagate for all vertices in parallel. Each call to Propagate performs rank propagation along an edge $\langle v, u\rangle$. It sends the source rank v.rank to u and calculates accumulated ranking scores in u.sink by Equation 4 (Line 7). Then VMap runs UpdateRank in parallel, to update the score of every vertex v based on v.sink (Line 9).

IncEval is similar to PEval except that it (1) starts with border nodes updated by messages (Line 3), and (2) resets v.sink for each v in V_i to 0 (Line 4). Both steps are parallelized with VMap.

Assemble is triggered as soon as the changes between two consecutive IncEval iterations are below threshold ϵ. It simply collects the ranking scores of all vertices, and returns the scores as output.

Remarks. Algorithm 5 is a representative example for converting a VC algorithm into a PIE+ program. It shows how to parallelize vertex-centric operations with EMap and VMap. In general, given a VC algorithm such as a GAS program of PowerGraph [33], we may convert it into a PIE+ program as follows: (1) declare an intermediate hashmap Π for caching incoming messages at all vertices; (2) recast Scatter with VMap to generate the outgoing message for each vertex; (3) recast Gather with EMap to perform the message passing via adjacent edges of each vertex, and caching/aggregating them in the corresponding receiver entries of Π; (4) recast Apply with VMap to update each vertex based on its own status and its aggregated messages in Π; (5) embed the implementation above in PEval and IncEval; and (6) specify Assemble based on the output.

Algorithm 5: PR under PIE+.

```
Input: subgraph $F_i = (V_i, E_i)$, a threshold $\epsilon$.
Output: the rank of every $v$ in $V_i$.
/* declare status variable: $v$.sink = 0 for each $v \in V_i$;   */
Function PEval ($F_i$):
1   VMap ($V_i$, function ($v$) { $v$.rank := rand(); return $\emptyset$; });
2   EMap ($V_i$, Propagate); VMap ($V_i$, UpdateRank); return $\emptyset$;
Function IncEval ($F_i$, $M_i$):
3   VMap ($M_i$, function ($m$) { $v$.rank := $m$.rank; return $\emptyset$; });
4   VMap ($V_i$, function ($v$) { $v$.sink = 0; return $\emptyset$; });
5   EMap ($V_i$, Propagate); VMap ($V_i$, UpdateRank); return $\emptyset$;
6 Procedure Propagate ($u, v$):
7     $v$.sink := $v$.sink + $u$.rank/$u$.degree$^+$; return $\emptyset$;
8 Procedure UpdateRank ($v$):
9     $v$.rank := $d \cdot v$.sink + $(1-d)$; return $\emptyset$;
```

A.2 Graph Simulation

We next study graph simulation, an algorithm that is easy to write under graph-centric GC model, but is nontrivial under VC.

Consider a graph pattern $Q = (V_Q, E_Q, L_Q)$, which is defined just like graphs (Section 2). Given a pattern $Q = (V_Q, E_Q, L_Q)$, a *match* of Q in graph $G = (V, E, L)$ via graph simulation, denoted by Sim, is a binary relation $R \subseteq V_Q \times V$ such that (1) for each pair $(u, v) \in R$, $L_Q(u) = L(v)$; and (2) for each pattern vertex $u \in V_Q$, there exists a vertex $v \in V$ such that (a) $(u, v) \in R$, and (b) for each pattern edge $\langle u, u'\rangle \in E_Q$, there is an edge $\langle v, v'\rangle \in E$ such that $(u', v') \in R$. If G matches Q, there exists a *unique maximum* relation [36], denoted as $Q(G)$; otherwise, $Q(G)$ is an empty set. It is in quadratic time to compute $Q(G)$ [36].

Outline. We show how PIE+ implements the algorithm of [36] for Sim, denoted as $\mathcal{A}_{\mathsf{sim}}$. Given a pattern Q and a graph G, $\mathcal{A}_{\mathsf{sim}}$ computes the unique maximum relation $Q(G)$. It maintains a set sim(u) for each pattern vertex $u \in V_Q$, initialized as the set of all vertices v in G that bear the same label as $L_Q(u)$; it then refines sim(u) iteratively by removing invalid candidates until a fixpoint is reached.

Under PIE+, $\mathcal{A}_{\mathsf{sim}}$ is a minor adaptation from the PIE program of [28]. It works as follows on every subgraph F_i. (1) It declares a Boolean variable $x(u, v)$ for each pattern vertex u in Q and each vertex v in F_i; flag $x(u, v)$ is set true if v is a potential match of u; it is reset to false when v is removed from sim(u). (2) PEval refines sim(u) for all $u \in V_Q$ and computes $Q(F_i)$ following [36]. At the end of PEval, each worker aggregates $x(u, v)$ into MessageStore by taking a conjunction in v's corresponding entry. (3) IncEval is the incremental simulation algorithm of [26], which treats message updates as edge deletions. It works on each status variable in the messages, propagates the changes to the affected area, and refines sim(u). At the end of each IncEval process, *changed* flags $x(u, v)$ of border nodes v are aggregated by W_p just like in PEval. (4) Assemble is triggered when no more messages are exchanged; it takes a union of $Q(F_i)$ for all i and returns the union as $Q(G)$.

Use of EMap and VMap. Algorithm $\mathcal{A}_{\mathsf{sim}}$ parallelizes edge/vertex-level operations with EMap and VMap. (1) During PEval initialization, VMap is used to assign the initial match set sim(u) for all $u \in V_Q$ in parallel. (2) Later in PEval, VMap parallelizes refinement of $\mathsf{sim}(u)$ for each u, during which EMap is used to check invalid matches of neighbors of u in parallel. (3) In IncEval, VMap parallelizes incremental processing of each "edge deletion".

Remarks. PR is a representative example for converting a GC algorithm into a PIE+ program. In general, the conversion is as follows: (1) start from its PIE program, which typically parallelizes an efficient sequential algorithm; (2) identify independent operations, *e.g.*, loops that iterate on an edge/vertex set, as commonly found in graph algorithms; and (3) replace these loops with an EMap or a VMap, passing in an update function that is functionally equivalent.

A.3 Single-Source Shortest Path

We next present a PIE+ algorithm for single-source shorted path (SSSP), for which algorithms are known under both GC and VC.

Consider a directed graph $G = (V, E, L)$, where edge label $L(e)$ is a positive number for $e \in E$. The length of a path $\langle u_0, u_1, \ldots, u_k\rangle$ in G is $\Sigma_{i=0}^{k-1} L(\langle u_i, u_{i+1}\rangle)$. For a pair (s, d) of vertices in V, $\mathsf{dist}(s, d)$ denotes the shortest distance from s to d. Given such a graph G and a source $s \in V$, SSSP is to compute $\mathsf{dist}(s, v)$ for all $v \in V$.

Outline. For subgraph F_i, we declare a variable $x(v)$ for all vertices v in F_i, denoting $\mathsf{dist}(s, v)$. These status variable are initialized as ∞ except $x(s) = 0$. PEval implements the Dijkstra's algorithm [30], and computes $x(v)$ for all v along paths within F_i. MessageStore aggregates $x(v)$ of border nodes by taking min as aggregate function f_{aggr}. Then, IncEval follows the incremental SSSP algorithm in [60] upon receiving the updated $x(v)$ values as messages. It refines local $x(v)$ values iteratively, by recalculating shortest paths from border nodes v with updated $x(v)$. When no further updates can be made to any $x(v)$, Assemble aggregates $\mathsf{dist}(s, v)$ for all v in V.

Use of EMap. In both PEval and IncEval, a vertex v is marked *active* if its $x(v)$ is updated. All active vertices are cached in a min heap, sorted by $x(v)$. The algorithm iteratively (1) removes the minimum element v from the heap and marks it *inactive*; and (2) checks every neighbor u of v, and updates $x(u)$ with a smaller value if it exists. The PIE+ program employs EMap to parallelize accesses to v's neighbors, similar to BFSes in WCC (see Line 6 of Algorithm 3).

A.4 Breadth-First Search

We then program for breadth-first search (BFS), another problem for which we have known algorithms in both GC and GC.

Given a directed graph $G = (V, E)$ and a *root* vertex v_0 in V, BFS is an ordered graph traversal that (1) starts from root v_0, and (2) visits all vertices at the current depth level before moving on to the vertices at the next level. The depth of vertex v is defined by the length of the *shortest path* that starts from v_0 and finishes at v.

BFS is different from the procedure called in WCC (Algorithm 4, Section 4). WCC executes breadth-first traversal *within* each subgraphs; traversals on different subgraphs are independently because the order of visiting vertices does not really matter. BFS, however, requires a correctly ordered traversal across the *entire* graph.

Outline. For subgraph F_i, we declare an integer status variable $x(v)$ for all v in F_i. Variable $x(v)$ records the traversal depth of v; its initial value is -1, indicating that v has not been visited before. We use such variables to sort vertices in BFS and avoid repetitive visits. The status variable for root $x(v_0)$ is initialized as 0.

For each subgraph F_i in which v_0 resides (including all mirrors if G is partitioned via, *e.g.*, vertex-cut [5]), PEval traverses F_i via a recursive BFS procedure. Starting from root v_0, it finds each direct neighbors u of v_0 and set x(u) = 1. Then, after visiting depth-k vertices, it visits all *unvisited* direct neighbors of each depth-k vertex, and sets their status variables to $k + 1$. If a border node u is visited, it sends the traversal depth of u into its corresponding entry $\mathsf{MessageStore}[u]$, taking the minimum as f_{aggr}.

Then IncEval pulls messages from MessageStore and visits their neighbors recursively as in PEval but starts from border nodes with changed status. It concludes as soon as MessageStore is empty, and Assemble sorts visited vertices by their traversal depth.

Use of EMap. Like in WCC, we use EMap to parallelize visits to the neighboring vertices in both PEval and IncEval.

A.5 Random Walk

Finally, we provide a PIE+ program for graph random walk (RW), for which algorithms are known under both GC and VC.

RW is an important building block for many randomized graph processing (*e.g.*, graph sampling [38]) and learning algorithms. Given a directed graph $G = (V, E)$ and n source vertices in V, RW initiates by issuing n *walkers* at these sources. A walker walks in steps among vertices, while each step randomly selects one edge out of its current vertex. The walker keeps walking until a certain criteria is met, *e.g.*, it has taken an upper bound d of steps.

Outline. For subgraph F_i, PEval moves each walker w placed on F_i for a few steps and counts the total steps that w has taken, denoted by $c(w)$. When w is at vertex u, it terminates walk for w if either (1) u has no outgoing edge, or (2) $c(w) \geq d$. Otherwise, it takes a step by uniformly sampling one edge $\langle u, v\rangle$ going out of u, placing w on v, and incrementing $c(w)$. If v is a border vertex, it store $c(w)$ in the entry of v in MessageStore. IncEval for subgraph F_j pulls from MessageStore and replaces walkers on their corresponding vertices. It then follows the same lines as PEval. Finally, Assemble collects and returns the paths taken by all n walkers.

Use of VMap. RW employs VMap to parallelize operations of different walkers, which move independently and are fully parallelizable.

B IMPLEMENTATION DETAILS

MiniGraph is implemented in 12k+ lines of C++ code, based on the pipelined architecture of Figure 2. A thread is dedicated to Loader, which responds to read requests and loads subgraphs into memory, one at a time; similarly for Discharger. Both components perform synchronous I/O, and consume few CPU cycles. Evaluator maintains a thread pool for active workers, of size m on an m-core machine. Scheduler allocates threads to available cores. Moreover, Evaluator can move a thread from an active worker to another, without interrupting their executions and with negligible cost in light of the shared memory. The dynamic thread reallocation is easy since EMap and VMap readily decompose computation on each subgraph into *independent* calls to (low-cost) update functions f_E and f_V. MiniGraph employs MessageStore as the central exchange point for messages. To make a low memory footprint and improve concurrency, it is implemented as an array; it incurs a lower space and time cost than a key-value store (*e.g.*, a hashmap).

MessageStore. The execution model of MiniGraph is essentially subgraph-based, as opposed to the VC model. The synchronization among computation tasks is conducted via message passing. A typical message takes the form of a simple key-value pair, in which the key identifies a vertex and the value encodes its updates.

In MiniGraph, MessageStore works as the central exchange point for messages, by caching and managing all pending messages. Considering the large volume of messages and the high-concurrency accesses, the rule of thumb of its design is to *lower the memory footprint* and to *improve the access efficiency*.

One might want to implement MessageStore as a list of message buckets, which are referred to as *receiver-buckets*. By maintaining a bucket—which is implemented as a hashmap for quick lookups—for each subgraph, it groups messages destined for the same receiver. However, it (1) introduces value replication among buckets, and (2) incurs several times of space overhead to avoid key collisions, even with a memory-efficient cuckoo hashmap [58].

In light of this, for MiniGraph, we take a radically different approach. To make predictable use of memory while retaining a low access cost, we implement MessageStore as a single fixed-size message array, from which every task pulls its desired messages via random accesses. Since the input graph uses consecutive natural numbers to identify the vertices, MessageStore can eliminate key storage and use a large array to index the values. More specifically, MessageStore initializes a (segmented) array whose size is the same as the number of vertices of the input graph. (1) To cache message (id, msg), we put msg into MessageStore[id]. If a message already exists, we aggregate it with update by applying aggregateMessage defined in PEval. (2) To access a message for border vertex id, a worker can read directly from MessageStore[id].

One possible concern is that the pre-allocated array itself may be too much of an overhead. We argue that, for most real-life graphs, border vertices count for a significant proportion; therefore, MessageStore can make an efficient use of most cells. As compared to receiver-buckets, it benefits more from the elimination of key storage, value replication, and memory overhead of a hashmap.

In our experiments with real-life graphs, MessageStore takes only a small portion in memory. For example, under various workloads over `web-sk` (32GB in size), its peak memory consumption is 220MB, 0.3% of the 64GB memory of our workstation testbed.

C PROOF OF THEOREM 1

We study the decision problem of the scheduling problem, which is denoted by DSP and stated as follows:

- *Input*: Subgraphs $F_0, F_1, \ldots, F_{n-1}$, the cost function $C_{\mathcal{A}}$ (Equation 2), an m-core machine whose memory capacity is η and disk read bandwidth is γ, and a deadline B.
- *Question*: Does there exist a valid schedule $\mathcal{S}$ whose makespan is bounded by B, *i.e.,* $\max_{i\in[0,n)}\{t_i + C_{\mathcal{A}}(F_i, p_i)\} \le B$?

Proof of the NP-completeness. We verify the upper bound of DSP by developing an NP algorithm. The algorithm works as follows: (1) first guess a schedule $\mathcal{S} = (\bar{p}, \bar{t})$; and (2) then check whether the makespan $\max_{i\in[0,n)}\{t_i + C_{\mathcal{A}}(F_i, p_i)\} \le B$; if so, return `true`. The algorithm is clearly in NP since verification in step (2) can be done in polynomial time (PTIME); hence so is DSP.

We verify the lower bound of DSP by reduction from PARTITION problem, which is known to be NP-complete (cf. [31]). PARTITION is to decide, given a finite set A of positive integers, whether there exists a subset $A' \subseteq A$, such that A' and its complement $\bar{A}' = A \setminus A'$ have an equal sum, *i.e.,* $\sum_{i\in A'} a_i = \sum_{j\in\bar{A}'} b_j$.

Given a set of n positive integers $A = \{a_0, a_1, \ldots, a_{n-1}\}$, we construct a schedule $\mathcal{S}$ for a set of subgraphs $F_0, F_1, \ldots, F_{n-1}$, a positive integer m, two positive numbers η and γ, a deadline B, and a cost function $C_{\mathcal{A}}$ to ensure the following property: A can be partitioned into A' and $\bar{A}'$ of equal sum if and only if $\max_{i\in[0,n)}\{t_i + C_{\mathcal{A}}(F_i, p_i)\} \le B$. Without loss of generality, we assume $A' = \{a_0, a_1, \ldots, a_{k-1}\}$ and $\bar{A}' = \{a_k, a_{k+1}, \ldots, a_{n-1}\}$.

The construction details are presented as follows.

(1) The cost function $C_{\mathcal{A}}$ is defined such that for all $i \in [0, n)$, $C_{\mathcal{A}}(F_i, 1) = C_{\mathcal{A}}(F_i, 2) = a_i$. More specifically, (a) $h^{\text{seq}}_{\mathcal{A}}(\bar{x}_i(v)) = 0.25$ and $h^{\text{para}}_{\mathcal{A}}(\bar{x}_i(v)) = 0.25$ are constants for any vertex v. (b) Fragment F_i is constructed as a collection of a_i connected vertex pairs. That is, F_i has a_i pairs of vertices. For an undirected graph, there exists an edge between each pair of vertices; for a direct graph, each vertex in the pair has an outgoing edge to the other. As a result, the average degree d_i of F_i is 1. One can easily verify from Equation 2 that $C_{\mathcal{A}}(F_i, 1) = C_{\mathcal{A}}(F_i, 2) = a_i$ for all $i \in [0, n)$.

(2) We set $m = 2$ and $\eta = \max_{|K|=\min\{4,n\}} \sum_{K\subseteq[0,n)} s(F_i)$. Intuitively, the machine has two cores (*e.g.,* Core 0 and Core 1), and the four largest fragments can fit into the memory at the same time. We also set $\gamma = \infty$, *i.e.,* loading time for a fragment is negligible.

(3) Schedule $\mathcal{S}$ assigns $p_i = 1$ for all $i \in [0, n)$. It sets the start time $t_0 = t_k = 0$, and $t_i = t_{i-1} + a_{i-1}$ for all $0 < i < k$ and $k < i < n$. In other words, fragments are partitioned into two groups $G_0 = \{F_0, F_1, \ldots F_{k-1}\}$ and $G_1 = \{F_k, F_{k+1}, \ldots F_{n-1}\}$. Each fragment in G_α is executed one after another on Core α, for $\alpha \in \{0, 1\}$.

(4) Finally, we set $B = \frac{1}{2}\sum_{a\in A} a$. Note that we assume *w.l.o.g.* that B is a positive integer and thus $\sum_{a\in A} a = 2B$.

We next verify the correctness of our reduction.

($\Rightarrow$) Suppose that A can be partitioned into A' and $\bar{A}'$ of equal sum. We have $\sum_{a\in A'} a = \sum_{a\in\bar{A}'} a = B$. The makespan of schedule $\mathcal{S}$ is

$$\begin{aligned}\max_{i\in[0,n)}\{t_i + C_{\mathcal{A}}(F_i, p_i)\} &= \max\{t_{k-1} + a_{k-1}, t_{n-1} + a_{n-1}\}\\ &= \max\{\Sigma_{i=0}^{k-1} a_i, \Sigma_{j=k}^{n-1} a_j\}\\ &= \max\{B, B\} = B.\end{aligned}$$

($\Leftarrow$) Suppose that the makespan of schedule $\mathcal{S}$ can meet deadline B, *i.e.,* $\max_{i\in[0,n)}\{t_i + C_{\mathcal{A}}(F_i, p_i)\} \le B$. We have that $\max\{\Sigma_{i=0}^{k-1} a_i, \Sigma_{j=k}^{n-1} a_j\} \le B$. As a consequence, one can deduce:

$$\Sigma_{i=0}^{k-1} a_i \le B, \tag{5}$$

$$\Sigma_{j=k}^{n-1} a_j \le B. \tag{6}$$

Here we construct $A' = \{a_0, a_1, \ldots, a_{k-1}\}$ and its complement $\bar{A}' = \{a_k, a_{k+1}, \ldots, a_{n-1}\}$. From Equations 5 and 6, we have that $\Sigma_{a\in A'} a \le B$ and $\Sigma_{b\in\bar{A}'} b \le B$, respectively. Provided that B is defined such that $\sum_{a\in A} a = 2B$, we have $\Sigma_{a\in A'} a = \Sigma_{b\in\bar{A}'} b = B$. In other words, A' and its complement $\bar{A}'$ have an equal sum B.

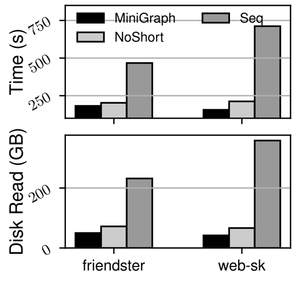


Figure 7: PR: 50% memory budget.

Name	Type	$\lvert V\rvert$	$\lvert E\rvert$	MaxDegree	Raw Data
roadNetCA [1]	road network	2M	2.7M	23	83MB
skitter [46]	network topology	1.6M	11M	35455	142MB
twitter [9, 44]	social network	41.6M	1.5B	3M	25GB

Table 5: Additional graph datasets.

D ADDITIONAL EXPERIMENTAL RESULTS

Below we report additional experimental results.

D.1 Additional Results for PageRank

This section shows some additional results for PR.

Figure 7 shows the performance of MiniGraph and its variants for PR, on friendster and web-sk. We find the following.

(1) Compared to MiniGraph$_{\text{NoShort}}$, on average MiniGraph is 11.2% and 35.9% faster over the two graphs, respectively. Moreover, shortcuts are much more effective on web-sk than on friendster. The reasons include the following: (a) web-sk has more connected components. Its largest connected component has 70.85% vertices, and 13.63% vertices are dangling nodes that are not connected to any other vertex. Under PR, some parts of the graph may already converge while other parts still require iterative computations. This improves the opportunity for shortcut (A) (Section 5.2). (b) On the contrary, friendster has a single large connected component only. As a result, PR actively updates the ranking cores of *all* vertices in *every* superstep, which effectively invalidates both Shortcuts (A) and (C). Out of the three shortcuts listed in Section 5.2, Shortcut (B) is the only one that can be actively exploited.

(2) MiniGraph is significantly faster than MiniGraph$_{\text{Seq}}$. It is 156.6% and 455.9% faster over friendster and web-sk, respectively. This is because of the VC nature of PR. It programs independent and massively parallelizable operations on individual vertices and edges; therefore, it would under-utilize the multi-core parallelism if intra-subgraph parallelism cannot be exploited on a single machine.

D.2 Additional Graph Datasets

Smaller graphs. We experimented with three smaller real-life graphs, including roadNetCA, skitter, and twitter, They are of different types, and Table 5 lists their details. Because roadNetCA and skitter are small in size, we ran the experiments with the entire graph in memory without imposing a memory budget. Over twitter, the tests were conducted under 50% memory budget.

Table 6 shows the performance of MiniGraph, GraphChi, GridGraph and XStream. For out-of-core workloads (*i.e.,*twitter), MiniGraph beats all baseline systems. On average, it beats GridGraph, GraphChi and Xtream by 1.4×, 4.2× and 12.4×, respectively.

MiniGraph underperforms prior systems in two cases over small graphs that can fit entirely into the memory. On skitter, *e.g.*, GridGraph takes 0.67× of the time of MiniGraph. This is because MiniGraph takes longer to initialize auxiliary data structures when it starts. The margin is small (a 0.18s difference); for larger graphs, it is amortized by the I/O and computational costs.

Additional results over clueWeb. Table 7 shows runtime statistics for some applications over clueWeb. The raw graph is 137GB large; it way exceed the physical memory capacity (47% of clueWeb) of our workstation testbed. That said, we no longer use cgroups to enforce a "software" memory budget in this experiment.

As shown in Table 4, MiniGraph is on average 4.59×, 4.25× and 2.1× faster than GridGraph for SSSP, WCC and PR, respectively. Table 7 further reports the number of supersteps, the volume of disk read traffic, I/O reduction from shortcuts, the average CPU utilization, the correlation coefficient r between real-time I/O and CPU usage, and the CPU cache hit rate. MiniGraph does better in all the aspects. The results are consistent for other algorithms.

The experimental results are consistent with applications over web-sk, another Web graph evaluated in Section 6. On average, (1) for SSSP, and WCC, MiniGraph reduces the BSP supersteps by 73.5%, and 91.9%, respectively. (2) Compared to GridGraph, MiniGraph induces 42.2%, 56.89%, and 46% less disk read traffic, in which shortcut optimizations contribute 5.3%, 15.3% and 12% for SSSP, WCC, and PR, respectively. (3) Compared to GridGraph, MiniGraph improves CPU utilization by 14.9%–45.2%.

D.3 Scalability

Under 50% memory budget, we next present additional experimental results on the scalability of MiniGraph.

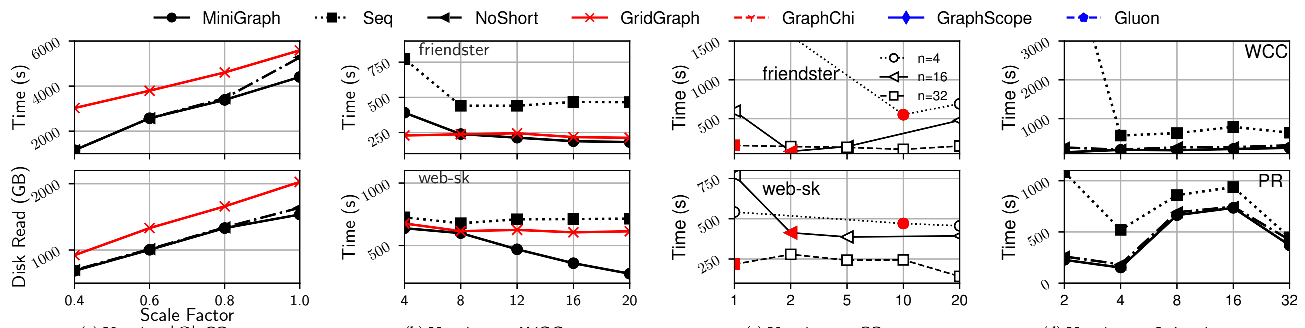
Varying $|G|$ for PageRank. As shown in Figure 8a when varying the scaling factor δ from 0.4 to 1.0 and G is clueWeb, (1) MiniGraph scales well with $|G|$. It takes 1.7× longer, while it is 1.8× for GridGraph. (2) MiniGraph also scales better than its variants. MiniGraph$_{\text{NoShort}}$ increases 4.6× when δ varies from 0.4 to 1.0. These further verify the effectiveness of our optimization strategies. (3) When $|G|$ grows 2.5×, MiniGraph induces only 2.2× I/O traffic, a large part due to the shortcut optimizations.

Varying m for WCC. Varying the number m of cores from 4 to 20, we ran WCC over friendster and web-sk. As shown in Figure 8b, (1) MiniGraph scales sub-linearly with m on both graphs. When varying m from 4 to 20, its performance improves 2.1× and 2.3× over friendster and web-sk, respectively. (2) It scales better than GridGraph, which does not improve much (up to +2.1% over friendster and +29.7% over web-sk) when m varies from 4 to 20. This is because GridGraph is always I/O-bound, which echoes our findings in Figure 6b. (3) It also scales better than MiniGraph$_{\text{Seq}}$, whose performance barely improves when $m \leq 8$ due to the lack of intra-subgraph parallelism. This further justifies the need for a hybrid parallel model that supports both GC and VC.

Varying p_i for PageRank. Besides testing the impact of varying p_i over clueWeb (see Figure 6h), Figure 8c reports its impact on PR

Data	Memory Budget	#Partitions (PR/Others)	SSSP				WCC				PR			
			MiniGraph	GraphChi	GridGraph	XStream	MiniGraph	GraphChi	GridGraph	XStream	MiniGraph	GraphChi	GridGraph	XStream
roadNetCA	100%	1/1	8.66	22.5 (2.6×)	10.55 (1.2×)	**2 (0.2×)**	**2.76**	17.2 (6×)	18.22 (6.6×)	2.93 (1.1×)	**0.25**	0.91 (3.5×)	0.71 (2.7×)	2.34 (2.6×)
skitter	100%	1/1	0.53	41.64 (78.5×)	**0.35 (0.67×)**	0.69 (1.3×)	**0.16**	18.43 (115.2×)	0.33 (2.1×)	0.59 (3.9×)	**0.27**	1.27 (4.7×)	0.82 (3.0×)	0.98 (3.6×)
twitter	50% (12.5GB)	4/10	**150.8**	802.8(5.3×)	195.4(1.29×)	2365(15.×)	**159.5**	594.8(3.7×)	186(1.2×)	1983(12.4×)	**224.2**	782.1(3.5×)	371.3(1.7×)	2183(9.7×)

Table 6: Execution time for SSSP, WCC and PR (in seconds) over more real-life datasets.



(a) Varying $|G|$: PR. (b) Varying m: WCC. (c) Varying p_i: PR. (d) Varying n: friendster.

Figure 8: Scalability of MiniGraph.

Metric	SSSP		WCC		PR	
	MiniGraph	GridGraph	MiniGraph	GridGraph	MiniGraph	GridGraph
# Supersteps	**18**	68	**16**	199	20	20
Disk Read (GB)	**2007**	3471.4	**1843.2**	4275.9	**1105.92**	2048
Shortcut I/O (GB)	-184.4	N/A	-655.36	N/A	-245.8	N/A
Average CPU Utilization	**18%**	3.1%	**19.8%**	4.1%	**55.3%**	10.1%
I/O-Compute Correlation	**-0.048**	-0.222	**0.021**	-0.093	-0.047	**0.029**
Cache Hit Rate	**53.27%**	50.13%	**55.38%**	53.66%	**69.37%**	63.28%

Table 7: Runtime statistics for SSSP, WCC and PR over clueWeb.

Cost model	$C_{\mathcal{A}}$	Model (a)	Model (b)
Normalized loss over S_{test}	0.16	0.22	0.22
Normalized loss over S'_{test}	0.40	0.50	0.43
Improvement web-sk (%)	39.0%	27.2%	27.3%
Improvement clueWeb (%)	30.0%	16.5%	17.1%

Table 8: Accuracy and effectiveness of cost model formulations.

over both friendster and web-sk. For each graph partitioning, we mark its tentative allocation $\hat{p}_i$ (Section 5) with colored points. The results show that for PR, our simple heuristic algorithm for deciding $\hat{p}_i$ again selects a near-optimal value for p_i. It further verifies that MiniGraph strikes a balance between the two levels of parallelism and improves the overall multi-core parallelism.

Varying n over friendster. In addition to experiments with different partitioning of web-sk (see Figure 6i), we also tested the impact of n over friendster. Varying the number n of partitions from 2 to 64, Figure 8d shows the performance of WCC and PR.

(1) For VC-based PR, its optimal n is also 4 (consistent with results over web-sk). It suggests that regardless of the graph distribution, VC algorithms with massively parallelizable operations always prefer exploiting intra-subgraph parallelism to inter-subgraph parallelism in MiniGraph. This is reasonable: a higher inter-subgraph parallelism often leads to a more fragmented graph, which generates more messages (updates) across supersteps, inducing more undesirable computational overheads for message aggregation.

(2) For GC-based WCC, the optimal n is 4. The result shows a pattern similar to the one over web-sk.

D.4 Cost Model Formulation

We also empirically verified (1) the accuracy of the cost model $C_{\mathcal{A}}$ in estimating the computation costs for a given PIE+ algorithm $\mathcal{A}$ (Equation 2, Section 5.1) and (2) its generalizability to graphs that are of the same type (*e.g.*, Web graphs) as training graphs. We also evaluated its overall impact on the system performance once applied in our resource scheduling strategy.

For WCC, the graph G is undirected, partitioned into n fragments $F_1, F_2, \ldots, F_n$. Following [25], the metric variables $x_i(v)$ include

$$x_i(v) = \{d_L, d_G, r, D\},$$

where $d_L(v)$ (resp. $d_G(v)$) denotes the degree of vertex v in fragment F_i (resp. graph G), $r(v)$ denotes the number of mirrors of v among all fragments (for, *e.g.*, vertex-cut [5] or hybrid-cut [19]), and constant metric D denotes the average degree of all vertices in G.

In addition to $C_{\mathcal{A}}$, We evaluated two alternative cost models:

Model (a) is a polynomial regression model in the same form of $C_{\mathcal{A}}$ yet taking fewer metric variables. Specifically, it takes a subset

$$x_i^{(a)}(v) = \{d_L, r\}$$

of metric variables in $x_i(v)$, without considering v's degree in G and the constant metric D in the polynomial.

Model (b) is a multi-layer perceptron, whose input layer takes p_i and all metric variables in $x_i(v)$ as input. In addition to the input and the output layer, it has two fully connected hidden layers, and makes use of ReLU as the activation function.

Below we empirically show that compared to models (a) and (b) above, the proposed cost model $C_{\mathcal{A}}$ strikes a better balance among accuracy, generalizability and training cost.

Model accuracy. Once trained for algorithm $\mathcal{A}$, $C_{\mathcal{A}}$ is expected to (1) make an *accurate* estimation of the computational cost; and (2) generalize well when applied to graphs of the same type. Both properties are inherited from the cost model proposed in [25].

To verify the two properties when $\mathcal{A}$ is the PIE+ program for WCC, we collected 588 (resp. 521) samples from the logs for which MiniGraph executed WCC over web-sk (resp. clue-web) in 30 runs. We then randomly selected 470 from the collected samples over `web-sk`, grouping them as the *training set* S_{train} and the rest 118 samples as the *test set* S_{test}. The samples over `clueWeb` are grouped as another *test set* S'_{test}. Both are of the Web graph type.

The cost models $C_{\mathcal{A}}$ and models (a) and (b) are trained with S_{train} and tested against both S_{test} and S'_{test} of two graphs.

Accuracy. Table 8 shows the normalized loss of $C_{\mathcal{A}}$, models (a) and (b) over test set S_{test}. Provided that S_v is collected from the same graph as the training set (`web-sk`), $C_{\mathcal{A}}$ results in 57.9% and 61.5% smaller loss than models (a) and (b), respectively.

While $C_{\mathcal{A}}$ and models (a) have a similar formulation, model (a) does not work as well as $C_{\mathcal{A}}$, because it removes d_G and D from the metric variables. Intuitively, these two variables are closely related to the synchronization cost for a vertex by *e.g.*, determining the number of calls to the aggregation function f_{aggr}. As explained in [25], they are important factors for GC algorithms where no explicit vertex functions are involved. The variables d_G and D can help estimate the amortized input size of vertex computation.

As a deep model with two hidden layers, model (b) requires more data to train. With 470 samples from S_{train}, it incurs overfitting and inaccurate estimates on S_{test}. This justifies $C_{\mathcal{A}}$ as a simpler polynomial formulation to strike a better balance between the training cost and accuracy, especially when the training data is insufficient.

Generalizability to different graphs. Table 8 also presents the normalized loss of the three cost models over test set S'_{test}, which are sampled from the running logs over a different graph of the same type (`clueWeb`, both are Web graphs). Here $C_{\mathcal{A}}$ exhibits the best loss among the three models under tests; its estimations are 27.2% and 5.0% more accurate than models (a) and (b), respectively.

It proves that the cost model formulation $C_{\mathcal{A}}$ has better generalizability when applied to graphs of the same type as the training graphs. The reasons behind are similar: model (a) ignores the effect of d_G and D, which are some key characteristics for computation with the GC-based WCC algorithm; and model (b) overfits S_{train}.

Impact on the system performance. We next deployed the trained model $C_{\mathcal{A}}$ and models (a) and (b) as the cost model in our scheduling strategy. It changes the resource scheduling behavior of MiniGraph. As a naive baseline for comparison, we deploy a randomized scheduling strategy, which randomly picks a pending subgraph for processing whenever sufficient memory is available.

We report the average execution time of MiniGraph across 10 runs under different scheduling strategies. As shown in Table 8 over `web-sk`, models $C_{\mathcal{A}}$ and (b) lead to 39%, 27.2%, and 27.3% improvement over the naive baseline, respectively. On different graph `clueWeb`, the improvement is 30.0%, 16.5% and 17.1%, respectively.

These justify the use of a cost estimation in resource scheduling. Moreover, in all cases, $C_{\mathcal{A}}$ leads to the best system performance. Recall that our scheduling strategy depends on an accurate cost estimate to (1) order the subgraphs in the pipelined processing; and (2) allocate idle threads at runtime. Model $C_{\mathcal{A}}$ allows MiniGraph to have a better chance in mitigating the stragglers, which potentially leads to improvements on the overall system performance.