Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction

Shuang Song
University of Texas at Austin
songshuang1990@utexas.edu

Xu Liu
College of William and Mary
xl10@cs.wm.edu

Qinzhe Wu
University of Texas at Austin
qw2699@utexas.edu

Andreas Gerstlauer
University of Texas at Austin
gerstl@ece.utexas.edu

Tao Li
University of Florida
taoli@ece.ufl.edu

Lizy K. John
University of Texas at Austin
ljohn@ece.utexas.edu

ABSTRACT

Graph processing systems are important in the big data domain. However, processing graphs in parallel often introduces redundant computations in existing algorithms and models. Prior work has proposed techniques to optimize redundancies for out-of-core graph systems, rather than distributed graph systems. In this paper, we study various state-of-the-art distributed graph systems and observe root causes for these pervasively existing redundancies. To reduce redundancies without sacrificing parallelism, we further propose *SLFE*, a distributed graph processing system, designed with the principle of "start late or finish early". *SLFE* employs a novel preprocessing stage to obtain a graph's topological knowledge with negligible overhead. *SLFE*'s redundancy-aware vertex-centric computation model can then utilize such knowledge to reduce the redundant computations at runtime. *SLFE* also provides a set of APIs to improve programmability. Our experiments on an 8-machine high-performance cluster show that *SLFE* outperforms all well-known distributed graph processing systems with the inputs of real-world graphs, yielding up to 75× speedup. Moreover, *SLFE* outperforms two state-of-the-art shared memory graph systems on a high-end machine with up to 1644× speedup. *SLFE*'s redundancy-reduction schemes are generally applicable to other vertex-centric graph processing systems.

PVLDB Reference Format:
Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction. *PVLDB*, 12(2): 154-168, 2018.
DOI: https://doi.org/10.14778/3282495.3282501

1. INTRODUCTION

The amount of data generated every day is growing exponentially in the big data era. By 2020, the digital data volume stored in the world is expected to reach 44 zettabytes [53, 10]. A significant portion of this data is stored as graphs in various domains, such as online retail, social networks, and bio-informatics [8]. Hence, developing distributed systems to efficiently analyze large-scale graphs has received extensive attentions in recent years.

To achieve high performance, existing graph systems exploit massive parallelism using either distributed [42, 38, 23, 65, 48, 17, 46, 24, 43, 61] or shared memory models [35, 47, 44, 52]. Such systems process graphs in a repeated-relaxing manner (e.g., using Bellman-Ford algorithm variants [12] to iteratively process a vertex with its active neighbors) rather than in a sequential but work-optimal order [7, 65, 41, 40]. This introduces a fundamental trade-off between available parallelism and redundant computations [41, 40]. We study several popular graph processing systems [65, 23, 18] and find that redundant computations pervasively exist, which we elaborate on in Section 2.

The root causes of computational redundancies in graph analytics vary across applications, which is due to the nature (i.e., the core aggregation function) of different graph algorithms. For example, applications such as Single Source Shortest Path (SSSP) employ $min()$ as their core aggregation function. In each iteration, the values of active neighboring vertices are fed into the $min()$ aggregation function, and the result is assigned to the destination vertex. Typically, a vertex needs multiple value updates in different iterations because the value updates in any source vertices require recomputing the destination vertex's property. However, only one minimum or maximum value is needed in the end. Therefore, we define the redundancies in these applications as the computations triggered by the updates with intermediate (not final min/max) values. We propose a "start late" approach to bypass such redundant updates.

In contrast, some other graph applications (e.g., PageRank (PR)) utilize the arithmetic operations (e.g., $sum()$) to accumulate the values of neighboring vertices iteratively until no vertex has further changes (a.k.a final convergence). For algorithms of this kind, there are no computational redundancies caused by intermediate updates. However, our analysis shows that most vertices are early converged (the vertex's value is stabilized) before a graph's final convergence. Hence, following computations on the early-converged vertices are redundant. We propose a "finish early" approach to terminate the subsequent computations on these vertices to eliminate such redundancies.

Proceedings of the VLDB Endowment, Vol. 12, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3282495.3282501

We develop *SLFE* (pronounced as "Selfie"), a distributed graph processing system that reduces redundancies to achieve high-performance graph analytics. *SLFE* employs a novel preprocessing technique that produces a graph's topological information to guide redundancy reductions in the execution phase via the principal of "start late or finish early". In contrast to the prior work that leverages dynamic re-sharding/partitioning [59] or multi-round partitioning [34] to reduce redundancy in out-of-core graph systems, our strategy has the following benefits: 1) it does not incur any extra partitioning effort [1]; 2) it does not rely on any specific ingress methodology, so it can be easily adopted by other systems; 3) it has extremely low preprocessing overhead, which is suitable for online optimization; and 4) it produces guidance that is reusable by various graph algorithms for the redundancy optimizations.

To balance the communication and computation on the fly, *SLFE* uses the state-of-the-art `push`/`pull` computation model. The `push` operation sends the update of source vertices to their successors, while the `pull` operation extracts information from predecessors for a given destination vertex. To the best of our knowledge, *SLFE* is the first graph system with a set of redundancy-reduction aware `push`/`pull` functions to make use of guidance produced in preprocessing. Moreover, *SLFE* also provides a set of system APIs to enable redundancy reductions as well as programming simplicity/flexibility for different graph applications. We summarize the contributions of this paper as follows:

- We perform a thorough study on state-of-the-art graph processing systems and observe the pervasive existence of large amounts of computational redundancies. We further identify the provenance of these redundancies.
- We design a novel and lightweight preprocessing technique to extract a graph's topological information (i.e., propagation level). This technique enables both "start late" and "finish early" redundancy reduction principles for many graph applications.
- We develop *SLFE*, a distributed graph processing system that employs various techniques to demonstrate the benefit from optimizing redundancies in graph applications.
- We evaluate *SLFE* with extensive experiments and compare it with three state-of-the-art distributed and two shared-memory graph processing systems. Experiments with five popular applications on seven real-world graphs show that *SLFE* significantly outperforms these systems, yielding speedups up to 74.8× (16.3× on average) and 1644× (56× on average) over existing distributed and shared memory systems, respectively.

The rest of this paper is organized as follows. Section 2 presents our observations on state-of-the-art graph systems and motivates the *SLFE* design. Section 3 discusses *SLFE*'s redundancy reduction approach and elaborates system details. The experimental setup and evaluation results are presented in Section 4. In Section 5, we discuss the limitations of *SLFE*. We review the related work in Section 6 and present some conclusions in Section 7.

[1]The partitioning phase in distributed graph systems is expensive [58, 55, 39, 65].

Table 1: A list of graph analytical applications with two different aggregation functions [59].

Graph Analytical Applications	Aggregation Function
PageRank, NumPaths, SpMV, TriangleCounting, BeliefPropagation, HeatSimulation, TunkRank	Arithmetic (*sum* or *product*)
SingleSourceSP, MinimalSpanningTree, ConnectedComponents, WidestPath, ApproximateDiameter, Clique	Comparsion (*min* or *max*)

2. OBSERVATIONS AND MOTIVATION

In this section, we review the graph applications that can benefit from *SLFE* and present observations of computational redundancies to motivate *SLFE*.

2.1 Graph Applications

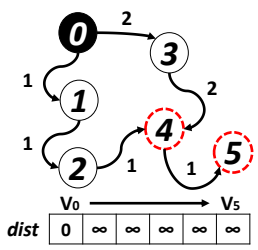
Most popular graph applications can be classified into two categories based on their aggregation functions of either arithmetic operations or *min*/*max* comparisons. We analyze graph applications implemented atop several systems [35, 38, 47, 52, 24, 23, 18, 42] and summarize our findings in Table 1. For applications from both categories, *SLFE* aims to provide a unified solution to reduce their existing computational redundancies. It is worth noting that some graph applications do not employ any aggregation function, e.g., BFS visits each vertex only once. This kind of graph applications seldom introduces redundant computations, which is not the focus of *SLFE*. To explain the motivation behind *SLFE*, we choose SSSP (*comparison*) and PR (*arithmetic*) to show that redundant computations pervasively exist, and then discuss the provenance of such redundancy.

2.2 Computational Redundancy

We observe that state-of-the-art graph systems prefer to execute graph applications in a Bellman-Ford [12] way to utilize the massive parallelism available in hardware. Such implementations often introduce computational redundancies to graph applications with heavyweight *min*/*max* or arithmetic operations.

Figure 1 shows an example of SSSP execution (using *min*() as the core computing operation) in modern graph systems. To simplify the explanation, we denote vertex 0 as V_0, and an edge from vertex 0 to 1 as E_{01}. We leverage updates on V_4 and V_5 to demonstrate the provenance of computational redundancy. The vertex property *dist[v]* is initialized to 0 for V_0 and ∞ for other vertices. During $Iter_1$, the *dist* of V_1 and V_3 are synchronously updated to 1 and 2, respectively (updates are marked in gray). In the next iteration, the updates of V_1 and V_3 are propagated via the edges (E_{12} and E_{34}). Hence, the *dist* of V_2 and V_4 are computed to 2 and 4 correspondingly. Similarly, V_4's property is replaced by 3 (i.e., minimum *dist*) in $Iter_3$ and the *dist* of V_5 updates to 5. Due to the fact that V_4's *dist* is updated in $Iter_3$, its successor—V_5's *dist* has to be recomputed in $Iter_4$ and updated to its minimum distance 4.

From this example, we can see that multiple rounds of computations are needed to calculate the shortest path for V_4 and its successor, V_5. Such computations include multiple additions, *min* comparisons, and synchronous updates - all of which are time consuming in modern distributed graph systems. Similar behaviors are observed in other graph algorithms aggregated with *min*()/*max*() operations



(a) An example graph

	Iter1	Iter2	Iter3	Iter4
V0	0	0	0	0
V1	1	1	1	1
V2	∞	2	2	2
V3	2	2	2	2
V4	∞	4	3	3
V5	∞	∞	5	4

(b) SSSP's iteration plot

Figure 1: Example of SSSP computations.

as well. Such redundancies are due to the repeated-relaxing manner [7, 41, 40], where vertices are involved in computations at multiple propagation levels (e.g., V_4 resides in levels 2 and 3). Table 2 summarizes the number of updates/computations per vertex of SSSP in PowerLyra [18] and Gemini [65]. Both systems have a high number of per-vertex computations, 9.1 and 7.5 on average for PowerLyra and Gemini respectively. Note that this number is 1 if no redundant computation exists.

Some other applications such as PR use arithmetic *sum*() function for an aggregation process. Iteratively, the values of all immediate source vertices need to be fetched for every destination vertex's computation in PR. The convergence for this kind of algorithms is defined as the property of all vertices with no further change. There are two reasons that a vertex's value becomes stable:

- All the source vertices provide the same inputs as those in the past iteration.
- The precision supported by the underlying hardware cannot reveal the changes, as *float* can support 7 decimal digits of precision and *double* has 15 decimal digits of precision [2].

For instance, due to the limited hardware precision, even though the $\sum PR(v_{src})$ of two iterations are different, being divided by the same denominator (number of links of v_{dst}) can produce the same result. Generally, dozens to hundreds of iterations are needed to converge an entire graph. However, we find that many vertices reach a converged/stable state earlier than other vertices. To quantify the percentage of these early-converged vertices, we record every vertex's computations of seven graphs. Figure 2 shows that a large amount of vertices have their properties stabilized when the program reaches 90% of the execution time. For instance, in *orkut* and *delicious* graphs, 99% vertices are early-converged. The average percentage of such vertices is 83% across all the investigated graphs, indicating a room of redundancy reduction for graph applications based on arithmetic operations.

Even though the provenance of redundancies varies across applications, *SLFE* proposes a unified preprocessing method

Table 2: Updates per vertex of SSSP in PowerLyra [18] and Gemini [65] on a single machine. Details of the graphs (*orkut-friendster*) are shown in Table 4. "-" indicates failed execution due to exceeding memory capacity.

	orkut	liveJournal	wiki	delicious	pokec	s-twitter	friendster
PowerL	12.4	8.75	10.3	6.75	9.25	7.57	-
Gemini	9.91	7.66	7.28	5.6	9.42	4.51	8.18

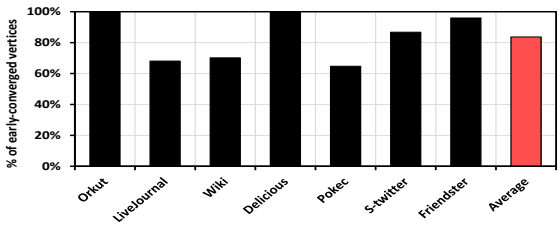


Figure 2: Percentage of seven real-world graphs' early-converged vertices in PR.

to generate optimization guidance for both types of redundancies. For *min*/*max*-based applications, *SLFE* provides a vertex's propagation sequences. Thus, computation in all but the last iteration can be avoided ("start late"). For applications with arithmetic aggregation functions, *SLFE* leverages the vertex's propagation sequences to justify the status of each vertex. Once a vertex's property has not been updated for x iterations ($x \geq$ its maximum/latest propagation level), the following computations on it will be avoided ("finish early").

3. SLFE SYSTEM DESIGN

To reduce redundancies for better performance, we develop *SLFE*, a topology-guided distributed graph processing system, which employs the concept of "start late or finish early". In this section, we first overview *SLFE*'s approach and then discuss its key system designs in detail.

3.1 Methodology and System Overview

Methodology Overview. *SLFE* aims to optimize the redundancy in various graph applications written with modern distributed graph processing systems. *SLFE* employs a novel preprocessing step that generates reusable graph topological information to guide redundancy optimization on the fly. Such information, known as Redundancy Reduction Guidance (RRG), captures the *maximum* propagation level of each vertex in a given graph. Due to the fact that the same vertex can exist at different propagation levels, each vertex will hold a RRG value—its *maximum* propagation level. At runtime, *SLFE* utilizes RRG to schedule the graph operations for redundancy reductions. *SLFE* adopts a system approach to minimize programming efforts and achieve high performance.

System Overview. Modern distributed graph processing systems [23, 38, 42, 65] typically consist of two phases: preprocessing and execution; *SLFE* follows the same design principle. As Figure 3 shows, *SLFE* loads, partitions, formats the entire graph in the preprocessing phase, and then generates the RRG for redundancy optimization. The subsequent execution phase accepts the preprocessing outputs (i.e., formatted graph and RRG). Section 3.2 and 3.3 elaborate on *SELF*'s partitioner and RRG generation, respectively.

SLFE's execution phase consists of three components as shown in Figure 3: Redundancy Reduction (RR)-aware runtime functions, Redundancy Reduction (RR) APIs, and graph applications. RR-aware runtime functions are an iterative-relaxing procedure to implement the hybrid `push`/`pull` computation model. Meanwhile, these functions

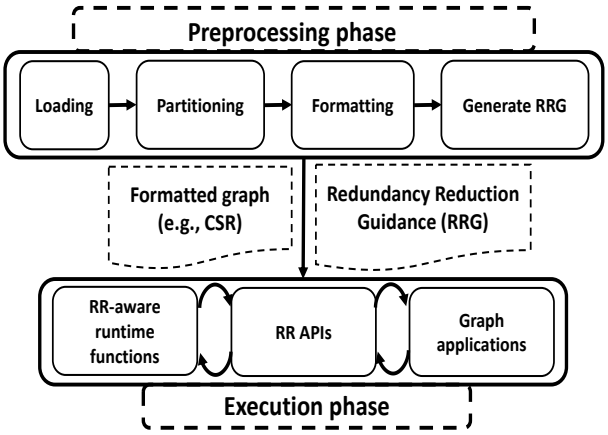


Figure 3: System overview of *SLFE*.

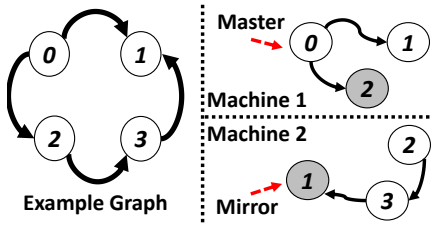


Figure 4: Example of the chunking partitioning.

	$Iter_1$	$Iter_2$	$Iter_3$	RRG
V_0	0	0	0	0
V_1	1	1	3	3
V_2	1	1	1	1
V_3	0	2	2	2

Figure 5: RRG for the example graph shown in Figure 4.

also apply RRG to optimize redundant operations at runtime. *SLFE*'s RR APIs bridge such computation model with various user-defined graph applications. From the user perspective, one can program various applications via *SLFE*'s RR APIs to transparently benefit from redundancy optimization. Sections 3.4-3.6 discuss each component of *SLFE*'s execution phase.

Section 3.7 further elaborates the work stealing technique used in *SLFE* to minimize work imbalance caused by redundancy elimination. Section 3.8 discusses the correctness of *SLFE*'s redundancy optimization. Section 3.9 provides a discussion of the technique generalization.

3.2 Chunking Partitioner

At the beginning of *SLFE*'s preprocessing phase, *SLFE* loads a graph's raw edgelist file as input. Since many large-scale real-world graphs often possess natural locality, storing adjacent vertices close to each other can preserve such locality [14, 57] with a minimal partitioning cost. We employ an efficient chunking partitioning scheme [65] to evenly partition a large-scale graph into contiguous chunks and assign a chunk to each machine in a cluster. Figure 4 shows an example of partitioning a graph $G = (V, E)$ for a 2-machine cluster. The vertex set V is divided into 2 contiguous subsets. To balance the workload across machines, the chunk-based partitioner allocates an equal amount of edges in each machine. Like existing approaches [42, 23, 18], some vertices are duplicated across different machines (e.g., V_1 and V_2) to reduce remote accesses. After partitioning, V_0-V_1 are in "Machine 1", while the rest are in "Machine 2". Partitioning graphs according to either incoming or outgoing edges yields the same results in this example.

After the partitioning, *SLFE* yields the desired formats: compressed sparse row (CSR) for the `push` operation and compressed sparse column (CSC) for the `pull` operation. *SLFE*'s graph partitioning and formatting schemes are not particularly designed for redundancy optimization. Actually, these schemes are commonly available in many other graph systems [23, 18, 38, 35, 56, 24, 9, 52, 65]. Thus, it dramatically enhances the applicability of *SLFE*'s techniques to other graph systems. The next step of the preprocessing phase is to generate RRG.

3.3 Redundancy Reduction Guidance

To guide the redundancy reduction in the graph execution phase, *SLFE* proposes a novel metric — Redundancy Reduction Guidance (RRG). Generating RRG follows a propagation manner (e.g., breadth-first search [16]) to record the iteration number of a vertex's update. We show the RRG generation with an example graph in Figure 5. In $Iter_1$, V_1 and V_2 receive an update from root V_0; their RRG values are updated to 1 (i.e., $Iter_1$). V_3 is the vertex updated in $Iter_2$. Later in $Iter_3$, due to the activeness of V_3, V_1 is revisited and its RRG value gets updated to 3 (i.e., $Iter_3$). The last column in Figure 5 illustrates the final RRG values of vertices in the example graph. Such RRG information represents a graph's topology, which can be reused by many graph applications for the redundancy reduction purpose.

Algorithm 1 shows the pseudo-code of our proposed preprocessing technique to generate the RRG. The declarations and initializations of the data structures are declared in line 1-3. The *fill_source* function in line 4 initializes all roots to 0 and other vertices to ∞. Starting from line 6, this procedure iterates through all the destination vertices to check whether it has an update in the current iteration. For all v_{dst}'s neighbors with incoming edges, if a neighbor's *dist* is computed in the past round, it notifies v_{dst} to update its RRG (line 9-10). This update indicates that v_{dst} resides in a new propagation sequence, which occurs later than the cached one. For an acyclic graph, a RRG update will activate v_{dst}. Finally, line 11-14 calculates v_{dst}. The weights of all edges are treated as 1 (line 12), as we only need to obtain the topological knowledge of the graph. Moreover, we use *visited* to only allow one computation per vertex. This is due to the fact that the first "visit" updates v_{dst} by its shortest distance, when all the edge weights are identical. This further minimizes the preprocessing overhead. Once v_{dst}'s *dist* is updated, it becomes active to propagate its value to the succeeding vertices.

After Algorithm 1, each vertex maintains a RRG value. Such value indicates the last propagation level that the vertex receives at least one update from the active source vertices. Any computation/update to the vertex happens before this point can be safely ignored for the redundancy reductions ("start late"). In the execution phase, such information can schedule the beginning of vertex computation for an application with $min()/max()$ aggregation function. For

Algorithm 1 Preprocessing to Generate RRG.

```
1: bool * visited = new bool[NumVerts];
2: uint32_t * RRG = new uint32_t[NumVerts];
3: int Iter = 1; int dist[NumVerts];
4: graph→fill_source(dist); //initialize vertices
5: for (int Iter=1; active vertex exists; Iter++)
6:   for v_dst ∈ V
7:     for v_src ∈ v_dst.incomingNeighbors
8:       if v_src.active
9:         if RRG[v_dst]] < Iter
10:          RRG[v_dst] = Iter;
11:        if ! visited[v_dst]
12:          dist[v_dst] = dist[v_src] + 1;
13:          visited[v_dst] = true;
14:          v_dst.active = true;
```

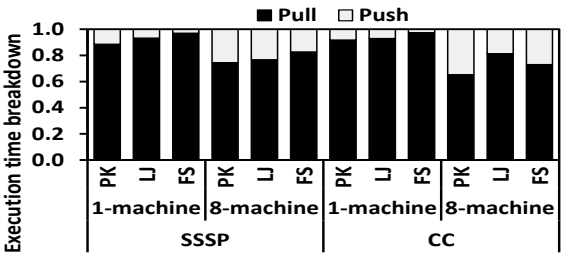


Figure 6: SSSP and CC execution time breakdown of `pull` and `push` mode, which are measured in 1-machine and 8-machine setup with *pokec* (PK), *liveJournal* (LJ), and *friendster* (FS) graphs.

instance, the V_1's RRG in Figure 5 is 3, all the computations happen before this iteration can be safely bypassed.

Even though applications with arithmetic operations can leverage the same RRG data to remove computational redundancies, the intuition behind is different. Considering the fact that most vertices converge earlier than graph's global convergence, RRG is used to justify the status of a vertex's stability. *SLFE* treats a vertex's RRG as the number of iterations needed to receive any new values from source vertices. If no change occurs on a vertex's property (e.g., rank in PR) for x rounds ($x \geq RRG$), it is considered as a stabilized/converged vertex. Thus, its further computations, known as redundancies, are bypassed ("finish early").

Overall, we can see that the proposed preprocessing technique is an extra step after finalizing graph partitions, which is generally applicable to any partitioning schemes and data formats. Thus, other state-of-the-art graph systems [65, 42, 23, 18, 61, 35, 24, 7] can easily adopt it. The overhead of our scheme is low and is thoroughly evaluated in Section 4. In the next section, we discuss how *SLFE* applies RRG in the execution phase.

3.4 RR-aware Runtime Functions

During graph execution, the number of outgoing/incoming edges of active vertices in each iteration varies dramatically. Thus, modern graph processing systems [65, 44, 52, 11] leverage direction-aware propagation model—`push` and `pull` to dynamically balance the communication and computation. This model optimizes the graph processing procedure on the fly. However, such a model increases the difficulty in applying redundancy reduction schemes at runtime. For instance, where do the redundant computations happen and how to incorporate the generated RRG in the model? To answer these questions, we first analyze the `push`/`pull` propagation model.

Algorithm 2 Pull Mode Computation.

```
1: def pullEdge_singleRuler(pullFunc, Ruler){
2:   pull = true;
3:   for v_dst ∈ V do
4:     if Ruler ≥ RRG[v_dst]
5:       pullFunc(v_dst, v_dst.incomingNeighbors);
6: }
7: def pullEdge_multiRuler(pullFunc, RulerS){
8:   pull = true;
9:   for v_dst ∈ V do
10:    if RulerS[v_dst] < RRG[v_dst]
11:      pullFunc(v_dst, v_dst.incomingNeighbors);
12: }
```

Algorithm 3 Push Mode Computation.

```
1: def pushEdge(pushFunc){
2:   if pull do
3:     activateAllVertices();
4:     pull = false;
5:   for v_src ∈ V do
6:     if v_src.hasOutgoing & v_src.active do
7:       pushFunc(v_src, v_src.outgoingNeighbors);
8: }
```

We measure the execution time of `pull`/`push` mode in SSSP and CC [2] with three natural graphs. The same measurements are performed on a single machine as well as a distributed cluster of eight machines to demonstrate the increasing effect of communications. As Figure 6 shows, SSSP and CC on a single machine spend more than 92.8% and 94% of their execution time in the `pull` mode. When we run them on 8 machines, the runtime in `pull` mode still consumes more than 78% and 73% in SSSP and CC, respectively. The small decrement in `pull` is due to the communication overhead (most communications happen in `push` rather than `pull`) caused by the increased cluster size. The advanced InfiniBand network minimizes such communication overhead. Thus, `pull` still contains most of the computations in the distributed setup. *SLFE* mainly optimizes redundancies in `pull`, while maintains sufficient parallelism to efficiently utilize all hardware threads [30]. We also attempt to eliminate redundancies in `push`. However, as the number of `push` operations is significantly less than the number of `pull` operations, the overhead of checking and removing redundancies surpasses the performance benefit. This insight is also reported in previous work [4]. Hence, rather than redundancy reductions, *SLFE* leverages the `push` to ensure the application's correctness.

The computations in `pull` mode extract the values of source vertices via incoming edges, and then apply a user defined *pullFunc* on the destination vertex. Algorithm 2 demonstrates the `pull` runtime design, which consists of two functions — *pullEdge_singleRuler* and *pullEdge_multiRuler*. At the beginning, the variable *pull* is set to true (we will explain the usage of this variable with `push`). In *pullEdge_singleRuler* function, for all v_{dst}'s, a single *Ruler* is used to control their executions (line 4). As aforementioned, the RRG of each vertex is used to optimize the redundancies. For instance, *min*/*max*-based algorithm uses the current iteration number as the single *Ruler*. If a

[2]We exclude applications with arithmetic aggregation functions, as they always execute in the `pull` mode to iteratively compute all vertices [4].

Table 3: RR APIs provided by *SLFE*.

min/max:	**void** edgeProc(pushFunc, pullFunc, activeVerts, Ruler);
arith:	**void** edgeProc(pushFunc, pullFunc);
	void vertexUpdate(vertexFunc);

vertex v_x's *RRG* is 4, the beginning of its iterations will be delayed after iteration 3. The *pullEdge_multiRuler* receives a *RulerS* array that is transparent to users. Each vertex has its own "Ruler" to follow. For iterative applications with heavy arithmetic operations, *RulerS* records each vertex's number of iterations that its property is continuously stable. Once $Rulers[v_x]$ passes its *RRG*, any further computation is eliminated.

In contrast, the `push` operation propagates a source vertex's value to all its neighbors via outgoing edges. Moreover, only one `push` function is shared by all applications (Algorithm 3), since `push` does not employ redundancy optimization, but only guarantees the result correctness. The details of Algorithm 3 are described as follows: In line 2-4, it checks whether the last iteration is in `pull` or not. If yes, this function activates all the vertices and sets the *pull* back to $false$. The "active list" technique [42] is commonly deployed by modern distributed systems [43, 65, 23, 38, 18, 61] to improve the communication efficiency. Thus, it only sends the property of active source vertices (line 6-7). However, due to the redundancy optimization, some "active" vertices may have been deactivated before reaching the `push` mode. Their successors may lose opportunities to check the properties of these predecessors. Such coincidences can potentially result in correctness issues. Therefore, we need to reactivate all the vertices in the transition phase (i.e., `pull`→`push`). Then, the active v_{src} vertices with outgoing edges use user-defined *pushFunc* to propagate its information. The next section presents the APIs that are used to bridge these RR-aware `push`/`pull` computation models with graph applications.

3.5 RR APIs

SLFE defines a set of application programming interfaces (APIs) to transparently optimize redundancy, as shown in Table 3. The *edgeProc* interface functions traverse a graph along the edges, while *vertexUpdate* applies application-specific operations to a vertex's property.

In the *min/max* API, $activeVerts$ records the number of active vertices in each iteration and terminates the execution early once no active vertex exists. *Ruler* compares with each vertex's *RRG* to schedule the computations. This API will be utilized for applications with *min/max* aggregation functions such as SSSP. In contrast, *edgeProc* for the *arith* API does not need any redundancy reduction inputs from the user side. In both *edgeProc* APIs, the number of active outgoing edges in the current iteration dynamically drives the decision of using either the `push` or `pull` computation model. The *vertexUpdate* applies user-defined *vertexFunc* to each vertex at the end of each iteration.

3.6 Programming with SLFE

This section presents SSSP and PR applications implemented atop *SLFE* as examples to show the programmability of *SLFE*. These examples show that with *SLFE*'s RR APIs, optimizing redundant computation requires minimum programming efforts.

Algorithm 4 Single Source Shortest Path.

```
1: float * dist = new float[numV];
2: v_root.active = true; dist[v_root] = 0.0;
3: uint32_t activeVerts = 1; uint32_t iter = 0;
4: pushFunc(v_src, v_src.outgoingNeighbors)
5:   for v_dst ∈ v_src.outgoingNeighbors
6:     float newDist = dist[v_src] + v_dst.edgeData;
7:     if newDist < dist[v_dst]
8:       dist[v_dst] = newDist; v_dst.active = true;
9: pullFunc(v_dst, v_dst.incomingNeighbors)
10:  float miniDist = MAX;
11:    for v_src ∈ v_dst.incomingNeighbors
12:      float newDist = dist[v_src] + v_src.edgeData;
13:      if newDist < miniDist
14:        miniDist = newDist;
15:    if dist[v_dst] > miniDist
16:      dist[v_dst] = miniDist; v_dst.active = true;
17: while (activeVerts)
18:    slfe.edgeProc(pushFunc, pullFunc,
19:                  activeVerts, iter++ ); // iter is Ruler
```

Algorithm 5 PageRank.

```
1: float* rank = new float[numV];
2: //graph traverse is similar to SSSP shown in Algorithm 4
3: //use the edgeProc(pushFunc, pullFunc)
4: float vOp(v_x)
5:   rank[v_x] = 0.15 + 0.85*rank[v_x];
6:   if v_x.hasOutgoing > 0
7:     rank[v_x] /= v_x.outEdges;
8:   return rank[v_x];
9: slfe.vertexUpdate(vOp);
10: //vertexUpdate is a system API to iterate through all Vs
11: uint32_t* stableCnt = new uint32_t[numV]; //RulerS
12: float* stableValue = new float[numV];
13: vertexUpdate(vOp)
14:   for v_x ∈ V
15:     if stableCnt[v_x] < RRG[v_x]
16:       float rank = vOp(v_x);
17:       if rank = stableValue[v_x]   stableCnt[v_x]++;
18:       else stableCnt[v_x] = 0; stableValue[v_x] = rank[v_x];
```

3.6.1 Single Source Shortest Path

SSSP follows a relaxation-based algorithm to iteratively compute the shortest distance from a given root to other vertices. SSSP requires user-defined *pushFunc*, *pullFunc*, activeVerts, and iteration counter (singleRuler for redundancy reduction) for *SLFE* to process the active vertices along with the connected edges. Algorithm 4 shows the pseudo-code of SSSP, where a property *dist* of each vertex stores its shortest distance. In `push` mode (line 4-8), each v_{dst} of v_{src} will receive a *newDist* composed by $dist[v_{src}]$ and the weight of a connected edge. To trigger such a computation, v_{src} needs to be active in this iteration. If the *newDist* is smaller than the current *dist* of v_{dst}, v_{dst} will be updated with this smaller value. Similarly, `pull` mode (line 9-16) iterates through a v_{dst}'s source vertices locally, and summarizes to get a local *miniDist*. If *miniDist* is smaller than $dist[v_{dst}]$, then it will be sent to the machine owning v_{dst} via message passing interface (MPI) [22]. Vertices with *dist* updates will be activated for the next round. Once there is no active vertex anymore, the process will terminate. Clearly, compared to the SSSP implementations on other systems, our SSSP does not incur any extra effort from the programming perspective.

3.6.2 PageRank

PageRank algorithm iteratively increases the relative rating of a vertex based on the weights of all connected vertices to rank the importance of each vertex in the graph. The propagation process (i.e., *pushFunc* and *pullFunc*) in PageRank is similar to the SSSP example shown above, hence, we only demonstrate the difference here. Algorithm 5 shows the implementation of PR application in *SLFE*. The *rank* array stores the properties of all vertices. Differing from SSSP, PR has to apply an extra user-defined function (line 4-8) on vertices' aggregated properties (*rank*) after each iterative propagation process. PR provides such function to *SLFE*'s *vertexUpdate* API. The pseudo-code of *vertexUpdate* is also shown in Algorithm 5 (line 11-18) to help understand how *SLFE* achieves redundancy minimizations for PR like applications. The vertex's status monitoring process happens in this function with the idea of tracking the number of continuous iterations that a vertex's *rank* has not been changed. Such a stable iteration will increase the *stableCnt* by 1 (line 17). If v_x has a new *rank*, its *stableCnt* will be erased and $stableValue$ will cache this new value (line 18). Once v_x's total number of stabilized iterations exceeds its *RRG*, we consider it as a early-converged vertex (line 15). Any further computation on it will be replaced by loading the cached *rank* from $stableValue$.

These two examples show that the implementation of graph applications on *SLFE* is very straightforward. Additionally, *SLFE*'s redundancy reduction philosophy does not incur any heavy modification on the manner that the graph application used to be coded.

3.7 Work Stealing

The workload balance of graph processing depends on many factors such as the initial partitioning quality, the density of active vertices on-the-fly, and so on. To overcome the load imbalances arising from the uneven redundancy reductions, we follow the idea of [13, 21, 20, 65] to implement a fine-grained work stealing mechanism in *SLFE*. In execution, each graph is split into mini-chunks, and each mini-chunk contains 256 vertices. Such design can enhance the hardware (i.e., core and memory systems) utilization and take advantages of hardware prefetching. To minimize the overhead of stealing work, each thread memorizes the starting point of the assigned mini-chunk, and simply uses a for loop to iterate vertices in the mini-chunk.

During the execution, all threads first try to finish up their originally assigned graph chunks before starting to steal remaining tasks from the "busy" threads. The starting offsets and other metadata shared by threads are preserved via the atomic accesses such as $\_sync\_fetch\_and\_*$. Although redundancy reduction may impact the workload balance across computation units, this explicit work stealing strategy can indeed solve the problem. The inter-machine balance is guaranteed by the chunking-based partitioning as described in [65]. We examine the quality of inter-machine workload balance in *SLFE*. Results in Section 4.3 show that *SLFE*'s redundancy-aware computation does not break the load balance achieved by the partitioning.

3.8 Correctness

Start Late. Most graph processing algorithms consist of many iterations of evaluating certain nodal function f_v applied at each individual vertex v. Such function $f_v : \mathcal{V}^{(t)} \rightarrow \mathcal{V}^{(t+1)}$ takes the current value of all source vertices $\mathcal{V}^{(t)}$ stored at iteration t and produces the next state value $\mathcal{V}^{(t+1)}$ for vertex v. For example, in the case of SSSP, the function f_v corresponds to the function $min()$, the input state $\mathcal{V}^{(t)}$ is reduced to only the set of current minimal distances $\{d_{n_1}^{(t)}, \ldots, d_{n_k}^{(t)}\}$ from the source to all immediate neighbors of vertex v, and the output produced is the current minimal distance from the source to vertex v (i.e $min(d_{n_1}^{(t)}+e_{n_1 v}, \ldots, d_{n_k}^{(t)}+e_{n_k v})$, where e_{ij} denotes the weight of the edge from i and j).

THEOREM 1. *SSSP produced from the* ***delayed vertex computation*** *converges to the original output.*

PROOF. The nodal/aggregation function f_v at each vertex in the SSSP algorithm is the $min()$ function, which is a monotonically decreasing function. The number of edges and all the edge weights are finite, therefore the value of $d_{n_k}^{(t)}$ is bounded by below as $t \rightarrow \infty$. Thus, by the monotone convergence theorem [28], the bypassed/delayed update procedure converges for SSSP. Moreover, since the initial graph state is the same for the original and the bypassed update procedure, these two procedures converge to the same value. □

If the output sequence $\{f_v(\mathcal{V}^{(0)}), \ldots, f_v(\mathcal{V}^{(t)})\}$ produced by the function f_v converges as $t \rightarrow \infty$ for all $v \in \mathcal{V}$, then the output produced from the **delayed update procedure** converges to the original output $f_v(\mathcal{V}^{(t)})$ as $t \rightarrow \infty$. Similar proofs can be applied on other graph applications with monotonic behaviors.

Finish Early. For graph applications with heavyweight arithmetic operations, *SLFE* monitors the value of each vertex, and determines each vertex's convergence accordingly. The value of a vertex V depends on its source vertices. V's RRG approximately measures the maximum propagation steps for V to receive an update. Thus, if V's value has been stable for a certain number of iterations larger than its RRG, it means no further change will be propagated to this vertex. *SLFE* bypasses the subsequent computations on such early-converged vertices. We experimentally verify *SLFE*'s accuracy by comparing *SLFE*'s results (e.g., vertices' properties) with the ones produced by other systems [65, 23, 18].

3.9 SLFE's Generality

The idea of redundancy reduction in *SLFE* is applicable to other graph frameworks. The reason we propose a new framework *SLFE* for our experiments is that *SLFE* employs a set of state-of-the-art optimizations, which exposes the performance bottlenecks in the redundant computation, rather than other components. This section describes how other graph frameworks can utilize our RR optimization.

The Gather-Apply-Scatter (GAS) computation model [23] has been widely adopted by many popular graph systems [23, 64, 58, 24, 56, 45]. The RRG provided by *SLFE*'s unique preprocessing stage can be used to schedule the vertex-centric GAS operations. For example, if the RRG reveals that an active vertex in the worklist has a certain amount of redundant computations, this vertex is removed from the worklist to avoid redundant computation on it. Take PowerGraph [23] as an example, one can adapt *SLFE*'s

Table 4: The graph datasets [37, 33, 32] used in experiments.

Real graph	—V—	—E—	AvgDegree	Type
pokec (PK)	1.6M	30.6M	18.8	Social
orkut (OK)	3.1M	117.2M	38.1	Social
livejournal (LJ)	4.8M	69M	14.23	Social
wiki (WK)	12.1M	378.1M	31.1	Hyperlink
delicious (DI)	33.8M	301.2M	8.9	Folksonomy
s-twitter (ST)	11.3M	85.3M	7.5	Social
friendster (FS)	65.6M	1.8B	27.5	Social
Synthetic graph	**—V—**	**—E—**	**AvgDegree**	**Type**
RMAT1	100M	2B	20	RMAT
RMAT2	300M	6B	20	RMAT
RMAT3	500M	10B	20	RMAT

methodology in the *receive message* stage to avoid performing redundant computation on vertices in each super-step. This optimization can save the network costs incurred in redundant *gather* and *scatter* operations, and eliminate the redundant computation in the user-defined *apply* operations.

4. EVALUATION

We evaluate *SLFE* on a 8-machine cluster with 2nd generation of Xeon Phi 7250 processor (Knights Landing) and InfiniBand network switch (up to 100Gb/s). Each machine has 68 physical cores; each core has a 32KB L1 I/D cache and a pair of cores share a 1MB L2 cache. The main memory system consists of 96GB DDR4 DRAM and 16GB MCDRAM; MCDRAM, configured as the last level cache shared by all the cores, has 5× bandwidth of DDR4.

We compare *SLFE* with three distributed graph processing systems—PowerGraph [23], PowerLyra [18], and Gemini [65]. In addition, we compare *SLFE*'s performance in a single machine with two shared-memory systems, GraphChi [35] and Ligra [52]. All the experiments include five popular graph applications from the two categories according to Table 1 (*min*/*max*: Single Source Shortest Path (SSSP), ConnectedComponents(CC), WidestPath (WP); *arithmetic*: PageRank (PR) and TunkRank(TR)). We report the average results of five repeated executions (standard deviation $< 5\%$). The input for these applications include seven real-world graphs and three large synthetic RMAT graphs [32] with the number of vertices and edges ranging from 1.6 to 500 millions and from 30 million to 10 billion, respectively, as shown in Table 4.

Experiment Outline. Section 4.1 compares *SLFE*'s end-to-end performance (preprocessing and execution time) with other three distributed systems running with the largest scale: 8 machines and 68 cores per machine. In this experiment, we use the real-world graphs as the input to show the performance in practical usage. Section 4.2.1 evaluates *SLFE*'s intra-machine scalability (scale-up), with the comparison with the two shared-memory systems. As limited to the memory size, we only use the real-world graphs as the input. Section 4.2.2 evaluates *SLFE*'s inter-machine scalability (scale-out). We use the RMAT graphs as the input to make sure each machine and each core has enough computation under large scale. Section 4.3 shows some micro metrics (e.g., number of computations and network traffic) to verify that our performance gains are due to the optimization of redundant computation.

Table 5: 8-machine end-to-end runtime and improvement over the state-of-the-art distributed systems. The end-to-end runtime includes both preprocessing time and an application's execution time. Gemini [65] and *SLFE* use the same preprocessing methodology.

Preprocessing time (loading + partitioning + formatting)								
		PK	OK	LJ	WK	DI	ST	FS
	PowerG [s]	74.2	277	168	891	736	210	4497
	PowerL [s]	84.4	312	191	982	811	239	5052
	SLFE [s]	3.46	10.5	7.9	35.2	45.3	14.1	295
RRG generation time								
		PK	OK	LJ	WK	DI	ST	FS
	SLFE [s]	0.05	0.08	0.13	0.6	0.68	0.3	1.62
Application execution time								
		PK	OK	LJ	WK	DI	ST	FS
SSSP	PowerG [s]	12.9	34.2	27.5	69.9	78.4	24.5	511
	PowerL [s]	10.3	23.0	18.8	34.5	18.9	17.3	243
	SLFE [s]	0.58	2.5	4.0	2.8	3.1	2.3	6.25
	Speedup(×)	19.8	11.2	5.7	17.4	12.4	8.9	56.4
CC	PowerG [s]	7.1	19.4	15.1	26.7	47.6	14.3	236
	PowerL [s]	5.7	10.4	10.8	15.6	14.2	3.0	112
	SLFE [s]	0.39	0.19	0.45	0.52	0.8	0.46	3.06
	Speedup(×)	16.2	74.8	28.4	39.2	32.5	14.2	53.2
WP	PowerG [s]	7.0	15.5	19.8	47.8	29.4	7.0	299
	PowerL [s]	6.1	10.2	16.0	33.1	11.1	5.3	164
	SLFE [s]	0.33	0.87	0.65	0.84	2.4	0.69	3.8
	Speedup(×)	19.8	14.5	27.4	47.3	7.5	8.8	58.3
PR	PowerG [s]	210	227	524	810	511	430	2874
	PowerL [s]	129	84.2	193	321	67.5	90.9	1415
	SLFE [s]	5.8	2.5	6.1	12.1	4.6	6.8	37.8
	Speedup(×)	28.4	55.3	52.1	42.1	40.4	29.1	53.3
TR	PowerG [s]	37.1	92.8	179	243	234	80.6	676
	PowerL [s]	14.3	34.7	80.4	137	57.7	15.5	304
	SLFE [s]	2.7	1.2	4.5	4.5	5.0	1.4	17.1
	Speedup(×)	8.5	47.3	26.7	40.5	23.2	25.2	26.5
GMEAN		25.1×						

Thorough Evaluation. We perform additional experiments, such as scale-out evaluation on real-world graphs and trend-line analysis on real-world graphs, and memory footprint analysis. Due to the space limitation, we upload these data to `https://github.com/songshuangVLDB19/VLDB19_Appendix`.

4.1 End-to-end Performance Evaluation

As *SLFE* aims to reduce computational redundancies for distributed graph processing systems, comparing to other state-of-the-art distributed systems can help quantify *SLFE*'s computational efficiency and high performance improvement. Table 5 reports the 8-machine performance of PowerGraph, PowerLyra and *SLFE*, running five popular applications on seven real graphs. The first part of Table 5 reports the preprocessing time of the three systems, which includes the graph loading, partitioning, and formatting steps. Since *SLFE* extends the general preprocessing phase to generate RRG information, such time cost is reported as well. For applications' execution time, the results show that *SLFE* outperforms these two systems in all cases significantly (25.1× on average), with up to 74.8× for CC on the OK graph. For the FS graph with more than 1 billion edges, *SLFE* achieves the highest average speedup (47.9×) among all input graphs. Thus, in contrast to these in-memory distributed systems, *SLFE* can handle large graphs more efficiently.

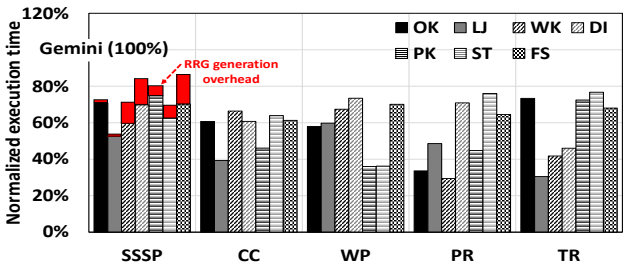


Figure 7: *SLFE*'s time of execution phase and RRG generation time normalized to Gemini [65] on a 8-machine cluster.

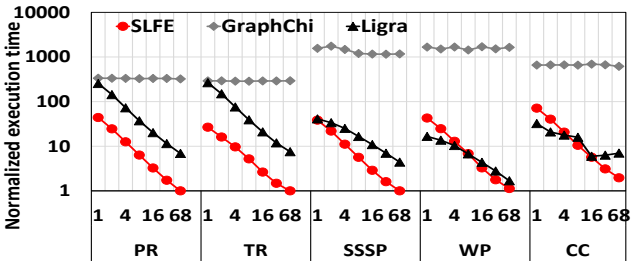


Figure 8: Intra-machine scalability (1-68 cores) of *SLFE*, GraphChi [35], and Ligra [52] on a single-machine setup (average of seven real world graphs).

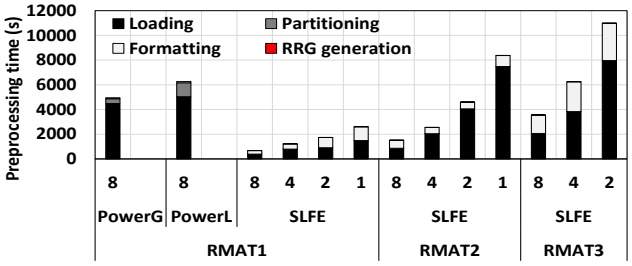


Figure 9: Inter-machine scalability of the preprocessing phase of PowerGraph [23], PowerLyra [18], and *SLFE* on three synthetic graphs. *SLFE* and Gemini [65] use the same preprocessing methodology except for the RRG generation. PowerGraph and PowerLyra can only execute the smallest RMAT1 graph in a 8-machine setup because of their inefficient memory usage [50, 27, 63, 65], while *SLFE* fails to process RMAT3 in a single machine. For all the cases, RRG generation is invisible because it incurs very small overhead.

While PowerGraph and PowerLyra are the general distributed graph platforms that provide many options for designers to test their ideas, Gemini is a dedicated computation-centric system that utilizes most of the state-of-the-art optimization techniques. In [65], Gemini is reported to outperform PowerGraph, GraphX [24], and PowerLyra by 19.1× on average. We compare *SLFE*'s performance with Gemini in Figure 7. Since *SLFE* and Gemini utilize the same preprocessing method (preprocessing time is already reported in Table 5), this end-to-end comparison between *SLFE* and Gemini only includes the time in the execution phase and RRG generation. Regarding to the time in the execution phase, *SLFE* outperforms Gemini by 34.2%, 43.1%, 42.7%, 47.5% and 41.6% on SSSP, CC, WP, PR, and TR, respectively. When including the RRG generation overhead, *SLFE* still yields an average of 25.1% (across the seven real graphs) end-to-end performance boost on SSSP, the one with the smallest performance improvement of the five applications. Additionally, such preprocessing overhead can be amortized, because it can be repeatedly utilized by execution with different inputs or even different graph applications. These performance gains show the effectiveness of *SLFE*'s unique redundancy optimization.

For the distributed graph processing, the update on a vertex triggers either a local atomic operation or a remote synchronization via the network. In contrast to other distributed platforms, *SLFE* reduces the number of computations, resulting in fewer updates, and thus less communication across distributed machines. In Figure 7, such benefits can be observed on relatively smaller graphs such as OK, LJ, and WK, where communication effect is amplified (up to 71% improvement). For the large FS graph, *SLFE* outperforms Gemini in all applications by 33.2% on average. Such improvement is mainly from the optimization of redundant computation, which dominates the execution time.

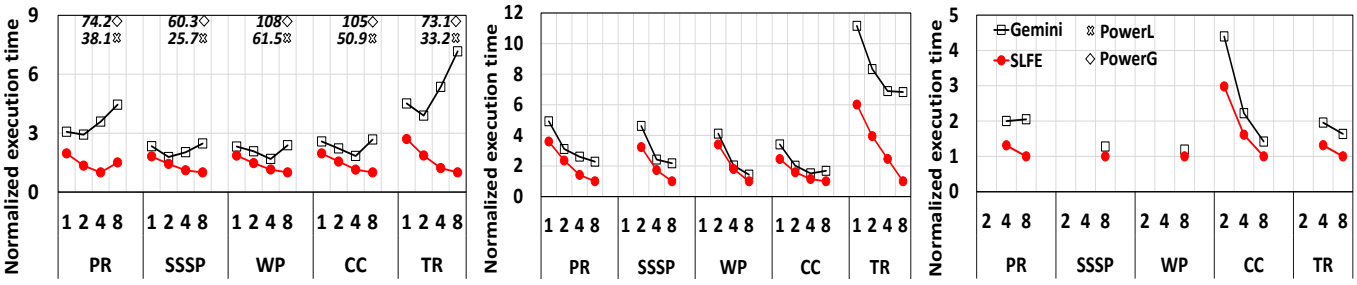
4.2 Scalability Evaluation

4.2.1 *Intra-machine Scalability*

Next, we evaluate the intra-machine scalability of *SLFE* by using 1 to 68 cores to run five applications with all real graphs that fit in a single machine's memory. Overall, Figure 8 shows that *SLFE* achieves nearly linear scale-up in all cases. For instance, compared to 1-core and 16-core cases, running on 68 cores achieves an average speedup of 44.7× and 3.5×, respectively. Although the pressure on shared hardware resources becomes more intensive as core count goes up, *SLFE* still maintains a decent speedup curve. Moreover, we compare *SLFE*'s scalability with two state-of-the-art single-machine systems—GraphChi and Ligra. GraphChi uses cost-efficiency to trade-off performance, where its bottleneck is the intensive I/O accesses. Therefore, as shown in Figure 8, it does not provides an outstanding intra-machine scalability. In contrast, Ligra takes the advantages of processing entire graph loaded in memory. Thus, compared to *SLFE*, Ligra has a very competitive scale-up trend. However, due to its excessive amount of computations and memory accesses, Ligra reaches the sub-optimal performance in most cases. *SLFE* reduces the computational redundancies, which results in less CPU usage and memory accesses in the shared-memory platform. Such optimization helps *SLFE* achieve up to 1644× and 7.5× speedups over GraphChi and Ligra when using the maximum of 68 cores. The absolute runtime data of this experiment have been uploaded to [1].

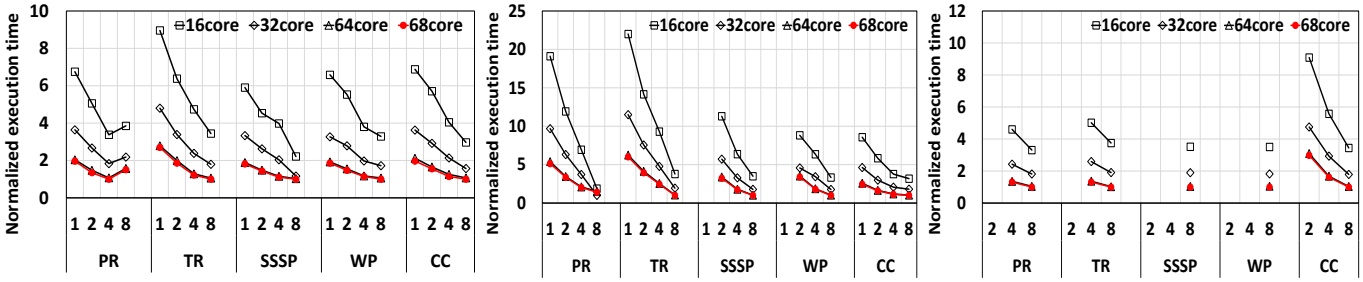
4.2.2 *Inter-machine Scalability*

To demonstrate *SLFE*'s inter-machine scalability, we use PaRMAT [32] to generate three large-scale synthetic RMAT graphs. Due to the inefficient memory usage [50, 27, 65, 63], PowerGraph and PowerLyra can only handle the smallest RMAT1 graph in our 8-machine cluster. However, Gemini and *SLFE* can operate these RMAT graphs in most cases. Figure 9 summarizes these four systems' preprocessing time. For RMAT1 graph on a 8-machine setup, *SLFE* finishes the preprocessing procedure much faster than the PowerGraph and PowerLyra. Moreover, as shown in Figure 9, *SLFE*'s preprocessing phase also provides a good scale-out scalability for all the three RMAT graphs. Compared to the original



(a) RMAT 1 (b) RMAT 2 (c) RMAT 3

Figure 10: Inter-machine scalability of the execution phase of Gemini [65], PowerGraph [23], PowerLyra [18], and *SLFE* (1-8 machines) on three synthetic RMAT graphs. Note: missing points are due to the failure of exceeding memory capacity.



(a) RMAT 1 (b) RMAT 2 (c) RMAT 3

Figure 11: Trend-line analysis of *SLFE*'s execution phase (1-8 machines with 16, 32, 64, and 68 cores per machine) on three synthetic RMAT graphs. Note: missing points are due to the failure of exceeding memory capacity.

preprocessing steps (e.g., loading, partitioning, and formatting), the proposed RRG generation overhead is negligible.

After comparing the preprocessing cost, we demonstrate the inter-machine scalability of the execution phase of the four systems in Figure 10. We also include the sum of *SLFE*'s execution time and RRG generation time to further verify the feasibility of our proposed approach in various scale-out configurations. As shown in Figure 10a, *SLFE* can achieve a significant speedup (up to 108× and 51.5×) over PowerGraph and PowerLyra for all the five applications with the RMAT1 graph on our 8-machine cluster. Gemini has an inflection point at 2-machine or 4-machine in all the five applications. Such behavior is due to the fact that the communication overhead surpasses the benefits obtained from adding more computation resources. However, with optimizing redundant computations, *SLFE* incurs less communication overhead so that it still scales down as the cluster size increases. On average, compared to the 1-machine setup, *SLFE* running 8-machine provides an average of 2.9× execution time speedup across the five applications with the three RMAT graphs. Among all the configurations (1 to 8 machines), *SLFE* outperforms Gemini by up to 7.2× (1.9× on average). This clearly indicates the feasibility and practicability of *SLFE*'s design principle.

In addition to the intra/inter-machine scalability experiments, we also report *SLFE*'s trend-lines by varying the number of cores per server as well as the number of server machines in the cluster. Figure 11 shows such analysis on the three synthetic graphs. We observe that when running PageRank (PR) with RMAT1 with 4 and 8 machines, the growing communication imposed by scaling out surpasses the benefit of additional computing resource. Even though TunkRank (TR) algorithm is similar to the PageRank, it does not face to such performance inflection. We further investigate this issue and find that compared to TunkRank, the redundancy reduction process in PageRank starts much earlier, which removing a larger amount of work. Insufficient computation with more communication overhead leads to the scaling loss. Overall, we can see that a larger cluster with more cores per machine can always speed up *SLFE*'s execution. On average, *SLFE* running with 68 cores can vertically (i.e., with the same number of machines) achieve 3.4×, 1.8×, and 1.1× speedup over 16-core, 32-core, and 64-core, respectively. Horizontally (i.e., with same amount of cores per machine), *SLFE* running with 8 machines achieve 3.6×, 2.6×, and 1.5× speedups over running on 1, 2, 4 machines, respectively.

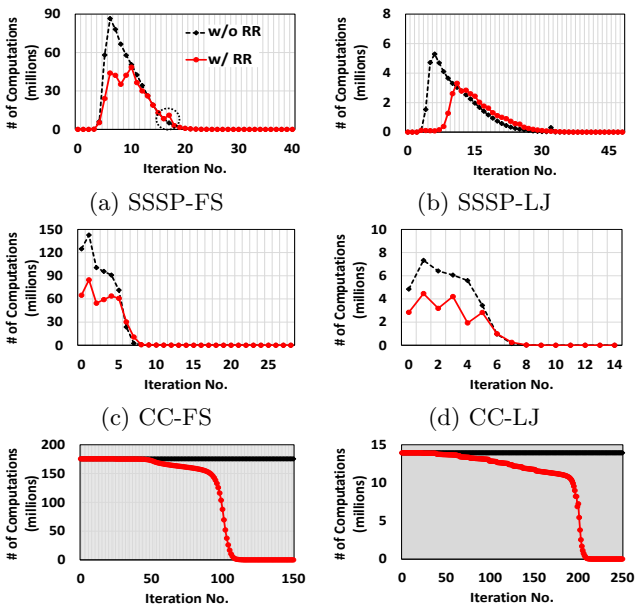
4.3 Further Discussions

To further understand *SLFE*'s gain from the redundancy optimization, we measure several micro metrics.

4.3.1 *Number of Computations*

The computation here is defined as an update on a vertex, which includes a min/max or arithmetic operation and the corresponding synchronization operations. We study the number of computations during execution phase of SSSP, CC, and PR in Figure 12. The reason for choosing these three applications is that they represent converging trends among the five applications: WP and SSSP have a similar converging trend; PageRank and TunkRank have a similar converging trend. In Figure 12, the "w/o RR" curves represent *SLFE* without RR enabled, and the same trends are observed in Gemini, PowerGraph, and PowerLyra systems (not shown). The "w/ RR" curves are obtained from *SLFE* with RR.

The SSSP initiates from a given root, and its number of computation dramatically increases when more vertices are



(a) SSSP-FS (b) SSSP-LJ

(c) CC-FS (d) CC-LJ

(e) PageRank-FS (f) PageRank-LJ

Figure 12: *SLFE*'s no. of computations per iteration.

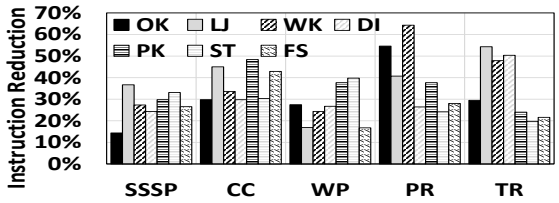


Figure 13: *SLFE*'s reductions on the number of instructions.

involved in the computation (Figure 12a and 12b). Redundant computations are reduced in the `pull` mode. Hence, compared to the original SSSP execution, *SLFE*'s ramping-up curves reach a much lower amount of computation. This phenomenon is caused by the "start late" approach, where intermediate updates are bypassed. Moreover, both curves ("w/RR" and "w/o RR") converge to the same point in the end, showing that the redundancy reduction leads to the correct results. As aforementioned in Section 3.4, the `push` function activates all vertices to deliver "unseen" updates of inactive vertices in the `pull`→`push` transition phase. We observe one such event (circled) in Figure 12a), which only incurs a small amount of immediate computations to guarantee the correctness. Figure 12c and 12d show that CC's number of computations is reduced along the converging. Like SSSP, CC's curves are finally merged in the end. In contrast, PR [35, 23, 18, 65] keeps updating each vertex in the execution. As more early-converged vertices are detected on-the-fly, the "finish early" principle on these vertices dramatically reduces the total amount of computations (Figure 12e and 12f).

4.3.2 Hardware and System Metrics

Instructions and Memory Accesses. We quantify the instruction reduction when *SLFE* enables redundancy reduction (RR) via counting the retired instructions using performance monitoring units [3, 6]. Such results are gathered from 8 machines. RR saves up to 64.3% (31.3% on average) instructions across all applications on all graphs. Moreover,

Table 6: *SLFE*'s reductions in terms of memory accesses.

Graph	—SSSP—	—CC—	—WP—	—PR—	—TR—
OK	20.4%	29.6%	25.6%	59.6%	32.2%
LJ	36.2%	48.9%	24.4%	45.8%	55.3%
WK	26.4%	35.7%	22.2%	62.1%	51.2%
DI	29.6%	30.2%	29%	22.7%	55.4%
PK	30.2%	45.2%	32.7%	39.9%	22.4%
ST	35.7%	31.3%	43.3%	25.4%	21.5%
FS	29.2%	45.7%	21.6%	33.2%	27.6%
GMEAN	29.2%	37.3%	27.6%	38.7%	35.2%

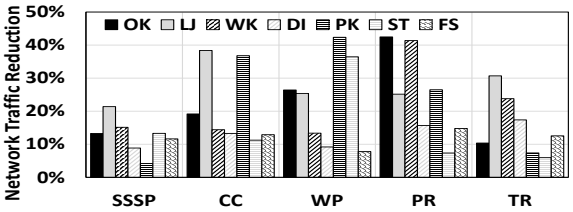


Figure 14: *SLFE*'s network traffic reduction on a 8-machine cluster.

we quantify the number of memory accesses via counting hardware load and store events. As shown in Table 6, *SLFE* reduces up to 62.1% memory accesses (33.3% on average).

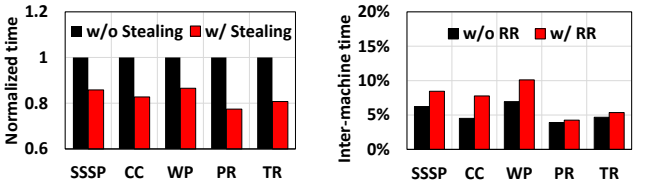
Network Traffic. The network traffic arises when synchronizing a vertex's master data and its mirror/remote data. Graph frameworks with different implementations have similar network traffic patterns. One important factor that impacts the traffic pattern is the partitioning strategy, which distributes vertices across machines in the cluster [36, 53, 54]. For simplicity, we compare *SLFE* to Gemini with the same partitioning strategy and traffic format (4-byte vertex ID and 8-byte data for each update). It is worth noting that, with this configuration, *SLFE* without RR generates the same amount of network traffic as Gemini. Figure 14 shows the improvement on network traffic via RR. Overall, RR yields up to 42.5% (19.3% on average) traffic reduction for five applications across all graphs.

Memory Footprint. Compared to PowerLyra and PowerGraph, *SLFE* reduces footprint by 80.3% and 72.6% on a 8-machine cluster. Compared to Gemini, *SLFE*'s footprint is 7.3% larger due to the storage of RRG.

4.3.3 Workload Balance

In the end, we analyze *SLFE*'s intra/inter-machine balance of the five applications with all real-world graphs running on 8 machines. Figure 15a shows the intra-machine case, where the baseline is the runtime achieved without work stealing support. From the figure, we can see that work stealing yields an average of 21% speedup for *arithmetic*-based PR and TR, while the work stealing speeds up the *min*/*max*-based applications by 15%.

In the inter-machine case, workload balance is mainly ensured by the quality of the initial graph partitioning. Figure 15b shows, without RR, the average time difference between the earliest and latest finished machines is less than 7% across all applications, thanks to the chunking-based partitioning approach [65]. With RR, *min*/*max*-based algorithms have a slightly higher inter-machine imbalance compared to others. This is due to the imbalanced message passing in the `push` mode after redundancy optimization. In contrast, PR and TR have much less on-the-fly commu-



(a) Intra-machine imbalance (b) Inter-machine imbalance

Figure 15: *SLFE*'s RR effects on balance.

nication messages, because they mostly execute in the `pull` mode. Overall, *SLFE*'s RR only increases an average of 2% inter-machine imbalance for all applications.

5. LIMITATIONS

SLFE has two limitations. First, redundancy reduction guidance needs to be generated in the preprocessing phase. Even though generating this re-usable topological information for the redundancy reduction purpose incurs extremely low cost, it is considered as overhead atop the original graph processing flow. Our future work is to further minimize the preprocessing overhead. Second, although not observed in our experiments, *SLFE*'s efficient redundancy reduction could potentially incur workload imbalance across computation units when the amount of eliminated redundancies varies. For the intra-machine case, we use work stealing to address this issue. However, it is challenging to address the potential inter-machine load imbalance due to costly communication via network. In the future, we will investigate various inter-machine work balancing schemes [31, 62] and integrate them into *SLFE*.

6. RELATED WORK

There are many distributed graph processing systems. Pregel [42] is the first one that proposes a vertex-centric programming model and Bulk Synchronous Processing (BSP) computation model, which have been widely adopted by other graph systems [38, 35, 23, 18, 61, 65, 48]. Some existing works like X-Stream [47] and Chaos [46] developed the edge-centric graph processing engine that sequence memory and I/O accesses. Other than the innovations on computation model, Powerlyra [18] and PowerSwitch [61] leveraged hybrid partitioning (vertex/edge cut) schemes and hybrid processing engines (sync/async) to accelerate graph analytics. Trinity [51] combined graph processing and databases into a single system. Apache Giraph [26] provided the ability to run on existing Hadoop [5] infrastructure and was highly optimized in terms of scalability by Facebook [19]. However, none of these existing distributed frameworks aim to improve graph processing performance by optimizing the redundant computations. Unlike these approaches, *SLFE* is the first one to optimize distributed graph processing with a novel redundancy-reduction design.

Other than the distributed solution, GraphChi [35] is a leading graph engine that analyzes large-scale graphs in a single PC; its parallel sharding window (PSW) technique efficiently utilizes the secondary storage. Based on this scheme, Vora et al. [59] optimized GraphChi to only load edges with new values. This optimization relies on the particular re-sharding technique of disk-based systems, which is not applicable to systems with entire graphs stored in distributed memories. Kusum et al. [34] proposed a graph reduction method to improve computational efficiency of Galois [43]. Such method performs iterative graph algorithms in a two-phase manner, which incurs an extra round of graph partitioning. This cannot be applied to the distributed systems, because the preprocessing is the most expensive process in the distributed systems [58, 55, 39, 65]. By contrast, *SLFE*'s solution is suitable for the distributed systems, as it does not rely on any specific partitioning strategies, and does not need any extra partitioning effort.

Orthogonal to the system optimizations, graph library, algorithm, and language designs are another related approaches. CombBLAS [15] offers an extensible parallel graph library with a set of linear algebra primitives. Parallel Boost Graph Library (PBGL) [25] provides graph data structures and message passing mechanisms (MPI) to parallelize applications. These libraries can employ *SLFE*'s redundancy reduction schemes. Wang et al. [60] leveraged articulation points to identify common sub-graphs, and reused the sub-graphs' results to remove redundancies in the betweenness centrality algorithm. Maleki et al. [41] proposed the Dijkstra Strip Mined Relaxation to reduce the communication overhead in SSSP. These algorithm optimizations speedup specific graph algorithms. In contrast, *SLFE* provides a system solution that is widely applicable to multiple graph applications. As for the graph domain-specific languages (DSL), Green-Marl [29] allows developers to describe graph algorithms intuitively and expose the data-level parallelism inherent in the algorithms. Sevenich et al. [49] adopt two DSLs to enable high-level optimizations from the compiler and skip the API invocation overheads. Our redundancy reduction optimization is orthogonal to these works.

7. CONCLUSIONS

In this paper, we propose *SLFE*, a novel topology-guided distributed graph processing system. With the design principle of "start late or finish early", *SLFE* reduces redundant computations to achieve better performance. *SLFE*, as a general framework, combines lightweight preprocessing techniques, system APIs, and runtime libraries to enable efficient redundancy optimization. Experimental results on an 8-machine high-end distributed cluster show that *SLFE* significantly outperforms state-of-the-art distributed and shared memory graph processing systems, yielding up to 75× and 1644× speedups, respectively. Moreover, *SLFE*'s redundancy detection and optimization schemes can be easily adopted in other graph processing systems.

8. ACKNOWLEDGEMENTS

We thank all the anonymous reviewers for the detailed comments on the paper. This research was supported in part by Google Faculty Research Award, National Science Foundation under grants 1745813, 1725743, 1822989, and 1822459, and NSF of China under grant 61602377, and ICR of Shaanxi China under grant 2018KW-006. Authors would also like to acknowledge computational resources from Texas Advanced Computing Center (TACC). Any opinions, findings, conclusions or recommendations are those of the authors and not of the National Science Foundation or other sponsors.

9. REFERENCES

[1] The github repository for the appendix of Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction. https://github.com/songshuangVLDB19/VLDB19_Appendix.

[2] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[3] Intel performance counter monitor. https://software.intel.com/en-us/articles/intel-performance-counter-monitor#abstracting, 2012.

[4] Avoid active vertex tracking in static pagerank. https://issues.apache.org/jira/browse/SPARK-3427, 2014.

[5] Apache hadoop. http://hadoop.apache.org/, Feb. 2018.

[6] Linux profiling with performance counters (perf). https://perf.wiki.kernel.org/index.php/Main_Page, 2018.

[7] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, 2017. USENIX Association.

[8] K. Ammar and M. T. Özsu. Wgb: Towards a universal graph benchmark. In *Advancing Big Data Benchmarks*, pages 58–72. Springer, 2014.

[9] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 313–322, May 2016.

[10] C. Baru, M. Bhandarkar, R. Nambiar, et al. Setting the direction for big data benchmark standards. In *Selected Topics in Performance Evaluation and Benchmarking*, pages 197–208. Springer, 2013.

[11] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[12] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[14] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, 2004. ACM.

[15] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[16] A. Bundy and L. Wallen. Breadth-first search. In *Catalogue of Artificial Intelligence Tools*, pages 13–13. Springer, 1984.

[17] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 215–226, New York, NY, USA, 2014. ACM.

[18] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[19] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[20] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *2008 37th International Conference on Parallel Processing*, pages 536–545, Sept 2008.

[21] K. F. Faxen. Efficient work stealing for fine grained parallelism. In *2010 39th International Conference on Parallel Processing*, pages 313–322, Sept 2010.

[22] M. P. Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[24] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, 2014. USENIX Association.

[25] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations.

[26] M. Han and K. Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.

[27] B. Heintz and A. Chandra. Enabling scalable social group analytics via hypergraph analysis systems. In *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'15, pages 14–14, Berkeley, CA, USA, 2015. USENIX Association.

[28] F. B. Hildebrand. *Methods of applied mathematics*. Courier Corporation, 2012.

[29] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[30] S. Karamati, J. Young, and R. Vuduc. An energy-efficient single-source shortest path algorithm.

In *2018 IEEE 32th International Parallel and Distributed Processing Symposium*, May 2018.

[31] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[32] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[33] J. Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, New York, NY, USA, 2013. ACM.

[34] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 245–257, New York, NY, USA, 2016. ACM.

[35] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[36] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John. Data partitioning strategies for graph workloads on heterogeneous clusters. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2015.

[37] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[38] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[39] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[40] S. Maleki, D. Nguyen, A. Lenharth, M. Garzarán, D. Padua, and K. Pingali. Dsmr: A parallel algorithm for single-source shortest path problem. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 32:1–32:14, New York, NY, USA, 2016. ACM.

[41] S. Maleki, D. Nguyen, A. Lenharth, M. Garzarán, D. Padua, and K. Pingali. Dsmr: A shared and distributed memory algorithm for single-source shortest path problem. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 39:1–39:2, New York, NY, USA, 2016. ACM.

[42] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[43] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[44] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[45] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. Energy efficient architecture for graph analytics accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 166–177, Piscataway, NJ, USA, 2016. IEEE Press.

[46] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, New York, NY, USA, 2015. ACM.

[47] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[48] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[49] M. Sevenich, S. Hong, O. van Rest, Z. Wu, J. Banerjee, and H. Chafi. Using domain-specific languages for analytic graph databases. *PVLDB*, 9(13):1257–1268, 2016.

[50] Z. Shang, F. Li, J. X. Yu, Z. Zhang, and H. Cheng. Graph analytics through fine-grained parallelism. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 463–478, New York, NY, USA, 2016. ACM.

[51] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[52] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[53] S. Song, M. Li, X. Zheng, M. LeBeane, J. H. Ryoo, R. Panda, A. Gerstlauer, and L. K. John. Proxy-guided load balancing of graph processing workloads on heterogeneous clusters. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 77–86, Aug 2016.

[54] S. Song, X. Zheng, A. Gerstlauer, and L. K. John. Fine-grained power analysis of emerging graph processing workloads for cloud operations management. In *2016 IEEE International Conference*

on Big Data (Big Data), pages 2121–2126, Dec 2016.

[55] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[56] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *PVLDB*, 8(11):1214–1225, 2015.

[57] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[58] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *PVLDB*, 10(5):493–504, 2017.

[59] K. Vora, G. Xu, and R. Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, 2016. USENIX Association.

[60] L. Wang, F. Yang, L. Zhuang, H. Cui, F. Lv, and X. Feng. Articulation points guided redundancy elimination for betweenness centrality. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 7:1–7:13, New York, NY, USA, 2016. ACM.

[61] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 194–204, New York, NY, USA, 2015. ACM.

[62] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1307–1317, Republic and Canton of Geneva, Switzerland, 2015.

[63] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300, Savannah, GA, 2016. USENIX Association.

[64] Y. Zhou, L. Liu, K. Lee, and Q. Zhang. Graphtwist: Fast iterative graph computation with two-tier optimizations. *PVLDB*, 8(11):1262–1273, 2015.

[65] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, GA, 2016. USENIX Association.