The Dissertation Committee for Shuang Song
certifies that this is the approved version of the following dissertation:

Improving Distributed Graph Processing by Load Balancing and Redundancy Reduction

Committee:

Lizy K John, Supervisor

Xu Liu

Vijay K Garg

Andreas M Gerstlauer

Earl E Swartzlander

Improving Distributed Graph Processing by Load Balancing and Redundancy Reduction

by

Shuang Song

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2020

Acknowledgments

First, I would like to thank my advisor, Professor Lizy K. John, for her guidance and support throughout my time in graduate school. Professor John taught me the fundamentals of computer architecture and performance evaluation during my early coursework. In my later years, she motivated me to continue hard work and provided invaluable guidance on my research. Other than work, Professor John has always been very supportive and provided me with much advice throughout my hard times.

I would like to thank my Ph.D. committee for their feedback at my qualifying exam and defense. Their constructive criticism helped the quality of this dissertation. I would like to give special thanks to Professor Andreas Gerstlauer and Professor Xu Liu. They have collaborated with me on many of my research projects. Their insightful advice improved the quality of my research.

I was blessed to meet all of the members of the LCA research group. While there are too many names to mention, I would like to specifically acknowledge Jee Ho Ryoo, Michael LeBeane, Reena Panda, Jiajun Wang, Qinzhe Wu, Zhigang Wei, and Ruihao Li. They have accompanied me and supported me throughout my time in graduate school. I do not know if I would have made it without their friendship and support.

I would also like to thank the organizations that supported me financially during my Ph.D. studies. National Science Foundation (NSF), Semiconductor Research Corporation (SRC), Samsung Austin Research and Development Center (SARC), and the University of Texas at Austin provided grants for my research.

I am very grateful to my parents for their emotional support throughout my Ph.D. program. They have always been a great source of encouragement towards this degree. I never would have made it without them.

Lastly, I would like to thank myself for not giving up in this lengthy Ph.D. program. Many people came and left throughout my Ph.D. time. These hard times made me become a stronger person. I am truly thankful for the past and all the things I have at this moment. I am looking forward to the future.

Improving Distributed Graph Processing by Load Balancing and Redundancy Reduction

Shuang Song, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Lizy K John

The amount of data generated every day is growing exponentially in the big data era. A significant portion of this data is stored as graphs in various domains, such as online retail and social networks. Analyzing large-scale graphs provides important insights that are highly utilized in many areas, such as recommendation systems, banking systems, and medical diagnosis. To accommodate analysis on large-scale graphs, developers from industry and academia design the distributed graph processing systems.

However, processing graphs in a distributed manner suffers performance inefficiencies caused by workload imbalance and redundant computations. For instance, while data centers are trending towards a large amount of heterogeneous processing machines, graph partitioners still operate under the assumption of uniform computing resources. This leads to load imbalance that degrades the overall performance. Even with a balanced data distribution, the irregularity of graph applications can result in different amounts of dynamic operations on each machine in the cluster. Such imbalanced work distribution slows down the execution speed. Besides these, redundancy also impacts the performance of distributed graph analysis. To utilize the available parallelism of computing clusters, distributed graph systems deploy the repeated-relaxing computation model (e.g., Bellman-Ford algorithm variants) rather than in a sequential but work-optimal order. Studies performed in this dissertation show that redundant computations pervasively exist and significantly impact the performance efficiency of distributed graph processing.

This dissertation explores novel techniques to reduce the workload imbalance and redundant computations of analyzing large-scale graphs in a distributed setup. It evaluates proposed techniques on both pre-processing and execution modules to enable fair data distribution, lightweight workload balancing, and redundancy optimization for future distributed graph processing systems.

The first contribution of this dissertation is the Heterogeneity-aware Partitioning (*HAP*) that aims to balance load distribution of distributed graph processing in heterogeneous clusters. *HAP* proposes a number of methodologies to estimate various machines' computational power on graph analytics. It also extends several state-of-the-art partitioning algorithms for heterogeneity-aware data distribution. Utilizing the estimation of machines' graph processing capability to guide extended partitioning algorithms can reduce load imbalance when processing a large-scale graph in heterogeneous clusters. This results in significant performance improvement.

Another contribution of the dissertation is the *Hula*, which optimizes the workload balance of distributed graph analytics on the fly. *Hula* offers a hybrid

graph partitioning algorithm to split a large-scale graph in a locality-friendly manner and generate metadata for lightweight dynamic workload balancing. To track machines' work intensity, *Hula* inserts hardware timers to count the time spent on the important operations (e.g., computational operations and atomic operations). This information can guide *Hula*'s workload scheduler to arrange work migration. With the support of metadata generated by the hybrid partitioner, *Hula*'s migration scheme only requires a minimal amount of data to transfer work between machines in the cluster. *Hula*'s workload balancing design achieves a lightweight imbalance reduction on the fly.

Finally, this dissertation focuses on improving the computational efficiency of distributed graph processing. To do so, it reveals the root cause and the amount of redundant computations in distributed graph processing. *SLFE* is proposed as a system solution to reduce these redundant operations. *SLFE* develops a lightweight pre-processing technique to obtain the maximum propagation order of each vertex in a given graph. This information is defined as Redundancy Reduction Guidance (RRG) and is utilized by *SLFE*'s Redundancy Reduction (RR)-aware computing model to prune redundant operations on the fly. Moreover, *SLFE* provides RR-aware APIs to maintain high promgrammablity. These techniques allow the redundancy optimizations of distributed graph processing to become transparent to users.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The growth of the Internet has made the Web graph a popular object for research and analysis. Other large graphs, like transportation routes, social network connections, paths of disease outbreaks, or citation relationships among published research work, have been processed for decades [76]. To extract the insightful knowledge, algorithms like shortest paths computation, clustering, and connected components are frequently applied atop these graphs.

However, efficiently processing graph algorithms on a large-scale graph is very challenging. Graph applications often exhibit high data dependency, irregular memory access patterns, various amount of work per vertex, and a changing degree of parallelism over the course of execution. Existing platforms (e.g., MapReduce) cannot handle these issues and often lead to a sub-optimal performance and usability issues in graph processing, as their computation models are designed to facilitate aggregation and SQL-like queries. Thus, industries and academia have proposed to design graph processing systems that can provide users a scalable, high performance, and fault-tolerant infrastructure to easily implement and deploy their graph applications. For instance, Google provides the Pregel [76] graph processing system that has already hosts hundreds of users' graph applications. Facebook im-

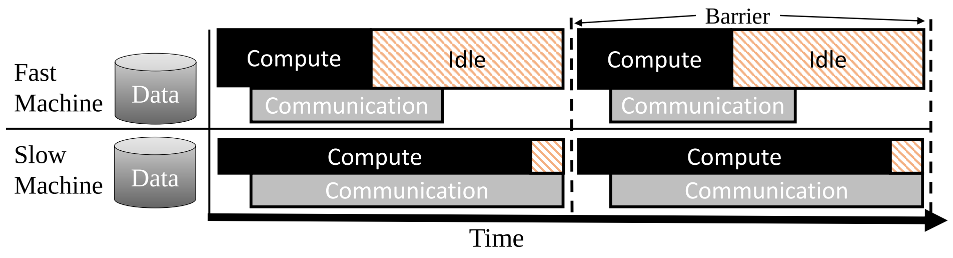


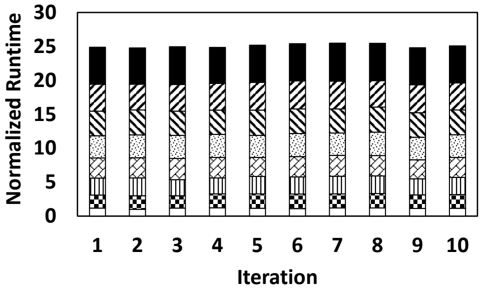
Figure 1.1: Workload imbalance due to even data partition in a heterogeneous computing cluster. The fast node is waiting at a barrier for a slower straggler node to finish computing.

plements and utilizes Giraph [33, 48] to study its growing social network with high performance (e.g., executing PageRank on 1.39 billion users with over 1 trillion social connections in less than 3 minutes per iteration with only 200 machines).

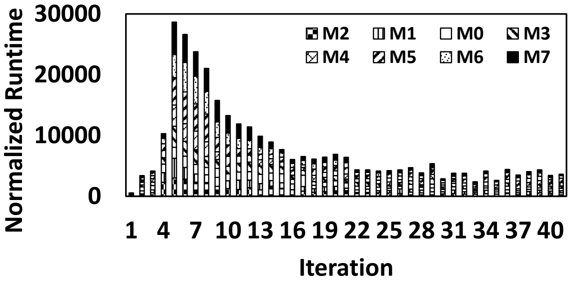
1.1 Problem Description

To process large-scale graphs with high performance, graph processing systems exploit massive computing parallelism and memory capacity using distributed models. However, distributing the large-scale graph among machines in a cluster introduces the workload imbalance issue that can result in sub-optimal performance.

For instance, data centers are populated with heterogeneous computing units (from expensive enterprise servers to networks of off-the-shelf commodity parts) for a variety of economic and performance related reasons. While data centers are trending towards heterogeneity, distributed graph processing frameworks still assume that computing resources are uniform. As shown in Figure 1.1, this assumption leads to the workload imbalance issue, causing the fast computing ma-



(a) PageRank



(b) SSSP

Figure 1.2: The normalized runtime of PageRank (10 iterations) and SSSP (41 iterations till convergence) executing friendster graph on a 8-machine cluster. Note that M0 stands for machine 0. The "slowest" machine (e.g., M7 in PageRank) bounds the overall performance.

chine to finish executing its chunk of data sooner than the slow machine. The slow "stragglers" decrease the overall cluster throughput whenever a synchronization is required, which results in low average processor occupancy and huge processing inefficiency. Observation shows that an 8 core machine spends more than 40% of its execution time waiting at a barrier for a 4 core machine to complete its task [69]. Ideally, for distributed graph processing, computing machines with various processing capability should reach the barrier at approximately the same time for higher computing efficiency.

Besides heterogeneity existing in data centers, the irregularity of graph analytics can also cause workload imbalance for distributed graph processing. For instance, irregular execution patterns of graph applications result in different amounts of dynamic operations[1]. To provide some context, this dissertation profiles multiple

[1]Dynamic operations in distributed graph processing include memory accesses, atomic operations, arithmetic operations, network transmissions, etc.

Table 1.1: Percentage of being the "slowest" machine executing PageRank and SSSP applications with *friendster* graph on a 8-machine cluster.

Application	M0	M1	M2	M3	M4	M5	M6	M7
PageRank	0	0	0	0	0	0	0	100
SSSP	17.1	2.4	9.8	9.8	7.3	29.1	4.9	19.5

stationary and non-stationary graph applications [61] executing real-world graphs on an 8-machine cluster. The real-world graphs are partitioned by the state-of-the-art vertex-cut algorithm used in many prior approaches [36, 87, 103, 132], where edges are evenly distributed among machines. Figure 1.2 shows PageRank (stationary) and SSSP (non-stationary) executing friendster graph as examples. The largest runtime difference among 8 machines in one iteration can be as high as $\sim 5\times$ for PageRank application and $\sim 7\times$ for SSSP. As identified by HPCToolkit [16], imbalance here is mainly caused by the uneven distribution of memory accesses and atomic operations.

More severely, an industry-level graph application [129] often consists of multiple graph mining kernels that perform both stationary and non-stationary computation. For example, applying PageRank and SSSP to an input graph in one application further complicates the load balance; as Table 1.1 shows, PageRank and SSSP have different workload imbalance patterns. Existing distributed graph processing systems [32, 36, 45, 48, 71, 76, 103, 128, 132, 133] rely on various graph partitioning schemes to distribute workloads at the pre-processing stage. However, it is difficult to design a common static partitioning algorithm that avoids imbalance for any given graph and application. Thus, dynamic load balancing

Table 1.2: Updates per vertex of SSSP in PowerLyra [32] and Gemini [132] on a single machine. Details of the graphs (*orkut-friendster*) are shown in Table 3.2. "-" 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

schemes [61, 93] are adopted to alleviate this imbalance issue. However, similar to other big data frameworks [17], the large overhead of balancing workloads on the fly can easily surpass the performance gains for distributed graph analytics.

Other than workload imbalance, processing graphs in a distributed manner also introduces a trade-off between parallelism and redundant computations. Distributed graph processing analyzes a graph in a repeated-relaxing manner (e.g., using Bellman-Ford algorithm variants to iteratively process a vertex with its active neighbors) rather than in a sequential work-optimal order to enjoy the benefits provided by a distributed cluster. More interestingly, 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 vertex require recomputing the destination vertex's property. However, only one minimum or maximum value is needed in the end. Table 1.2 summarizes the number of computations per vertex of

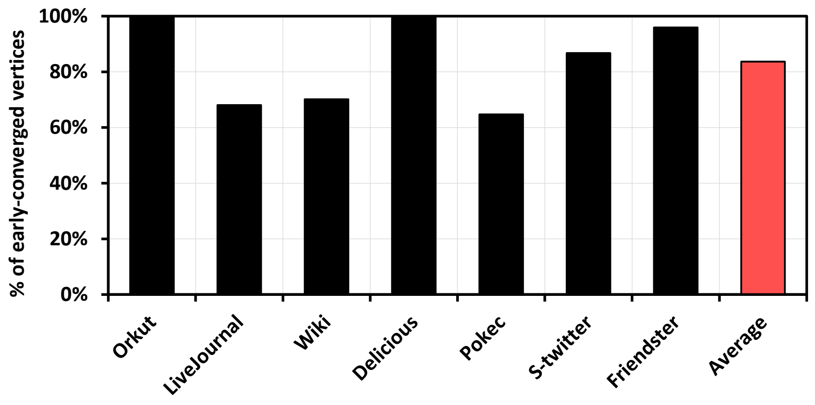


Figure 1.3: Percentage of seven real-world graphs' early-converged vertices in PageRank.

SSSP in two state-of-the-art distributed graph processing systems – PowerLyra and Gemini. Both systems have a high number of per-vertex computations. Note that ideally this number is 1 if no redundant computation exists.

In contrast, some other graph applications (e.g., PageRank) 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, analysis in Figure 1.3 shows that 83% of vertices across seven real-world graphs are early-converged (the vertex's value is stabilized) before a graph's final convergence. Hence, the following computations on these early-converged vertices are redundant and waste computational power. Providing a system solution to squeeze out redundant operations can help improve the computational efficiency of distributed graph processing.

1.2 Contributions

This dissertation proposes novel techniques to improve workload balance and computational efficiency of distributed graph processing. The primary contributions can be broken down into the following three topics:

1. **Heterogeneity-aware Partitioning for Distributed Graph Processing (*HAP*)**: *HAP* [58, 69, 100] proposes two lightweight methodologies to estimate a cluster's heterogeneity (defined as skew factor). In addition, it also offers a number of static graph partitioning strategies to utilize the skew factor for a better data distribution and load balance in a heterogeneous environment. Distributed graph processing systems can deploy *HAP* to achieve heterogeneity-aware data distribution.

2. **Auto-balancing Distributed Graph Processing On the Fly (*Hula*)**: *Hula* develops a hybrid partitioning scheme to maintain a graph's natural locality and generate metadata for lightweight online workload migrations. Meanwhile, it utilizes hardware timers to monitor balance status and support the workload scheduler on arranging work migration. The migration scheme proposed by *Hula* incurs minimal data movement during work migration. Finally, *Hula* integrates its lightweight workload balancing strategy with various state-of-the-art techniques to reduce workload imbalance on the fly for graph analytics in a distributed setup.

3. **A Distributed Graph Processing System with Redundancy Reduction (*SLFE*)**: *SLFE* [103] creates a novel and lightweight pre-processing tech-

nique to capture the maximum propagation level of each vertex in a given graph. Such information, defined as Redundancy Reduction Guidance (RRG), is used by *SLFE*'s Redundancy Reduction (RR)-aware runtime functions for redundancy optimizations during execution. To achieve a high programmability, *SLFE* implements RR-aware APIs so that redundancy optimizations are transparent to users. *SLFE* provides a system solution to reduce redundant operations for distributed graph processing.

1.3 Thesis Statement

The performance of analyzing large-scale graphs in a distributed environment can be optimized by heterogeneity-aware data partitioning, lightweight workload balancing, and redundancy reduction. These optimisations are achieved via system components (e.g., workload scheduler, runtime functions, etc.) that utilizes the guidance and metadata generated through lightweight pre-processing techniques.

1.4 Dissertation Organization

The organization of this dissertation is as follows. Chapter 2 describes the relevant background information on graph partitioning methodologies, computational model, and computational redundancy. Moreover, it also summarizes the prior art of load balancing and redundancy optimizations in the graph processing domain. Chapter 3 describes the distributed cluster setup, workloads, graph data sets and tools used to produce data throughout the dissertation. Chapter 4 describes the static data partitioning strategies for graph workloads on heterogeneous clusters. Chapter 5 presents *Hula*, where a systematic solution has been proposed to dynamically achieve load balance in a distributed format. Chapter 6 explores a mechanism for the distributed graph processing system to reduce redundant computations. Finally, Chapter 7 concludes the dissertation and suggests the future work in the area.

Chapter 2

Background and Related Work

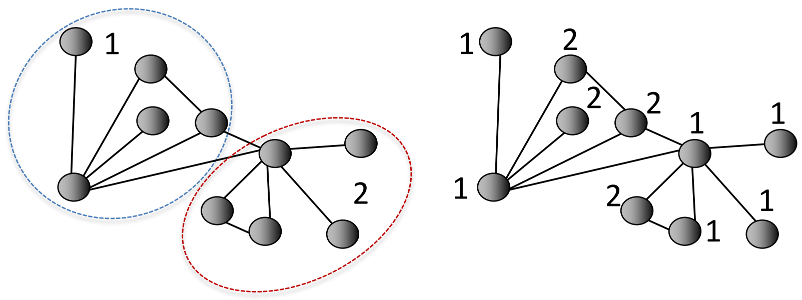
This chapter discusses relevant background information on graph partitioning methodologies, graph computational model, and computational redundancy. Moreover, it also summarizes the prior art of load balancing and redundancy optimizations for graph processing.

2.1 Graph Partitioning

Graph processing system generally consists of a pre-processing module and an application execution module. In the pre-processing module, a given graph will be partitioned and distributed across computing units. The quality of the partitioning process has a large impact on data duplication, communication overhead, and load balance. Many optimization methodologies in this dissertation build upon graph partitioning. Thus, this section will provide the fundamental background information of graph partitionings.

2.1.1 Offline vs Online Partitioning

Graphs can be partitioned using what are referred to as offline or online techniques [69]. Figure 2.1 illustrates the differences between offline and online graph



(a) Offline partitioning (b) Online partitioning

Figure 2.1: Offline vs streaming graph partitioning.

partitioning. Offline graph partitioning is the traditional best-cut problem. These partitioning strategies typically assume a global view of the graph and perform numerous iterations through the entire graph structure to achieve a high quality cut that balances the number of nodes and edges allocated to each partition while minimizing the number of edges that cross between partitions. A good example of a typical offline cut algorithm is the highly popular and successful METIS [60] algorithm. While offline cuts are generally of high quality, they are often unsuitable for the billion edge graphs common in big data analytics. The computational complexity of data ingress would be extremely large and would not typically amortize over the time of running the actual application of interest.

Online graph cutting algorithms, however, do not generally assume global knowledge of the graph structure and perform very few iterations through the graph. Instead, vertices and edges are streamed into an algorithm which makes an immediate decision on where to assign it. Online cuts run quickly, but are frequently subject to low quality, highly fragmented cuts, which can degrade the performance

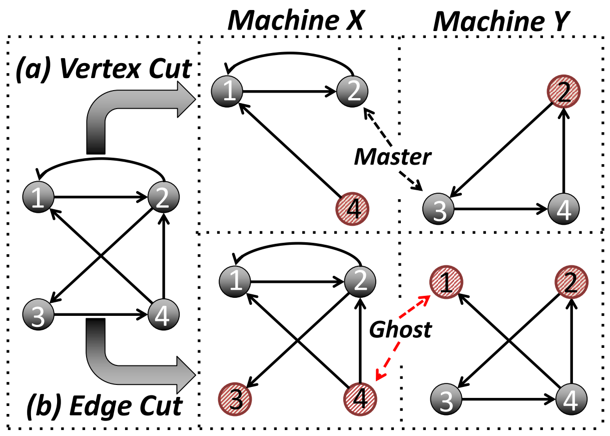
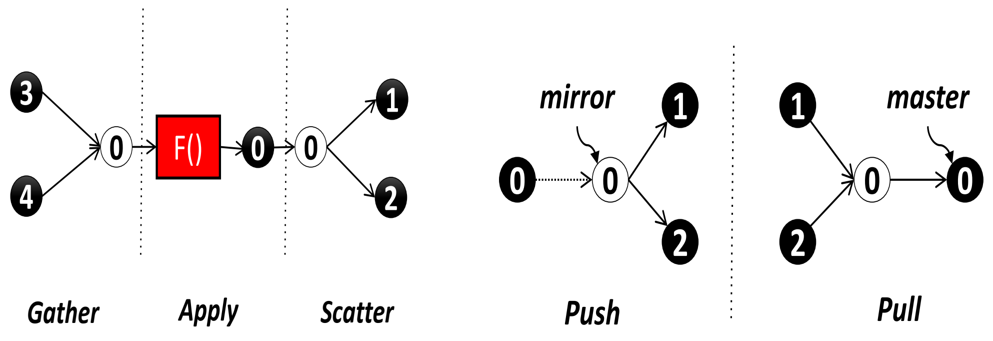


Figure 2.2: Edge vs vertex cuts.

of the application. More complicated online algorithms can be implemented, but the gains in running the actual algorithm of interest must outweigh the extra time added by a more complicated partitioner.

2.1.2 Edge vs Vertex Cuts

Whenever a graph is cut and split between two computing nodes, local copies of the elements that were assigned to a remote node are made. These copies, called *ghosts* or *mirrors*, are used to synchronize changes across the network and largely define the communication overhead of a graph processing framework [69]. In general, a graph partitioning algorithm has the choice of cutting a graph according to its edges or its vertices, as illustrated in Figure 2.2. An *edge cut* assigns vertices to partitions, and a *vertex cut* assigns edges to partitions. The optimal strategy largely depends upon the graph structure. Graphs with a large number of small degree vertices without major outliers perform better using edge cuts, since all the



(a) Gather-Apply-Scatter (b) Dual Update (Push/Pull)

Figure 2.3: GAS and Dual Update computation models.

edges attached to a given vertex are all owned by the same node. However, many real-world natural graphs follow what is known as a *power-law* distribution. Under a power-law degree distribution the probability that a vertex has degree d is given by:

$$P(d) \propto d^{-\alpha} \tag{2.1}$$

where the exponent α is a positive constant that controls the skewness of the degree distribution. Essentially, small values of α lead to high graph density where a small number of vertices have an extremely high degree. For these extremely high degree vertices, vertex cuts are preferred in order to improve load balance in partitions. However, vertex cuts suffer from the drawback that assigning edges to partitions can cause an excessive amount of network communication. This overhead occurs because the edges for a given vertex can easily be split across several nodes, requiring synchronization between every execution iteration.

2.2 Model of Computation

Once a given graph is partitioned and distributed in pre-processing, the execution of a graph application will begin. Applications implemented atop advanced graph processing systems are implemented and executed either in a vertex-centric manner or an edge-centric one. This section will discuss the state-of-the-art computation models in graph processing domain.

2.2.1 Vertex-Centric Computation

Vertex-centric graph computation employs the“think like a vertex” idiom to express graph applications. Many modern graph processing systems [18, 32, 45, 48, 68, 71, 99, 103, 128, 132] are implemented based on this, as it provides a flexible and intuitive programming method for users. Among all the vertex-centric computation models, bulk synchronous parallel (BSP), gather/apply/scatter (GAS) and dual update propagation are the most popular models of graph computations.

Bulk Synchronous Parallel BSP model is the first graph computational abstraction introduced in Pregel [76]. This allows all vertex-programs run simultaneously in a sequence of super-steps. With in a super-step, a program instance will receive the messages from last super-step. After processing the incoming messages, this program instance will compute outgoing messages. These will be sent to its out-neighbors. A synchronous barrier is placed between super-steps to maintain atomicity. The program will be terminated when there are no active message remaining.

Gather-Apply-Scatter Other than BSP, systems like [32, 45] employs GAS computation model. Logically, each vertex runs through the three phases of a vertex program independently of each other with barriers to enforce synchronization and correctness. Figure 2.3a illustrates the GAS computational paradigm on a set of vertices. During the gather phase of a vertex, the graph system performs a user defined map/reduce operation on the edges and vertices adjacent to a vertex *v*. The reduction from the gather phase is then passed on to the apply operation, which uses the current value in *v* and the reduced gather output to compute a new value for *v*. Finally, the new vertex state from the apply function is passed to the scatter stage, which makes the new value of *v* visible to neighboring vertices and edges during the next GAS operation. This sequence of events is repeated until a user defined stopping condition or the other convergence criteria has been reached.

Dual Update Propagation As graph system design advances, researchers realize that the number of active edges varies dramatically during graph processing. For instance, the active edge set of the Connected Components (CC) application is dense (i.e., has a large amount of active edges) in the first few iterations, and gets increasingly sparse (i.e., has a small amount of active edges) as more vertices receive their final labels. Based on such observation, state-of-the-art graph processing systems [22, 99, 103, 125, 132] have proposed direction-aware dual update propagation model. As Figure 2.3b shows, sparse active edge set prefers the `push` model (where updates are passed to neighboring vertices via outgoing edges), as the system only traverses outgoing edges of active vertices where new updates are generated. In contrast, dense active edge set benefits more from the `pull` model (shown in Figure 2.3b, where each vertex's update is done by collecting states of neighboring vertices along incoming edges), as this significantly reduces the contention in communication and synchronization.

2.2.2 Edge-Centric Computation

Vertex-centric graph computation involves random vertex access based on the activeness of vertices. After locating a vertex, the connected edges will be sequentially accessed. Compared to this, edge-centric computation streams a large number of edges and picking up the edges connected to active vertices to be processed. This trade-offs the data utilization (i.e., loading edges connected to a inactive vertex will lower the utilization rate of loaded data) to a sequential data access. Xstream [89] implements the edge-centric gather-scatter computation model to achieve a sequential access on slow storage (e.g., magnetic disk). In edge-centric scatter, edges will be sequentially read from storage and writes the updates. In gather, the updates will be sequentially read and applied to relevant vertices. This brings a performance improvement for out-of-core graph computations, where the sequential accesses on disks provides 500 times higher bandwidth compared to random accesses.

2.3 Computational Redundancy

Most popular graph applications can be classified into two categories based on their aggregation functions of either arithmetic operations or *min*/*max* compar-

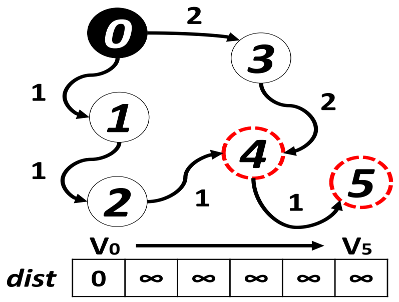
Table 2.1: A list of graph analytical applications with two different aggregation functions [113].

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

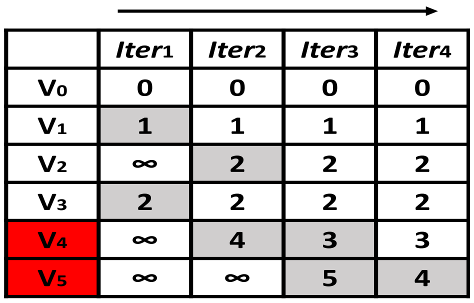
isons. This dissertation analyzes graph applications implemented atop several systems [32, 45, 46, 68, 71, 76, 89, 99] and summarizes the findings in Table 2.1. It is also worth noting that some graph applications do not employ any aggregation function, e.g., BFS visits each vertex only once. This kind of graph applications seldom introduces redundant computations, which is not the focus of this dissertation. To further explain the provenance of computational redundancies in graph analytics, this dissertation chooses Single Source Shortest Path (*comparison*) and PageRank (*arithmetic*) to represent the applications from each category.

The state-of-the-art graph processing systems prefer to execute graph applications in a Bellman-Ford [23] 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 2.4 shows an example of Single Source Shortest Path (SSSP) execution (using $min()$ as the core computing operation) in modern graph systems. To simplify the explanation, vertex 0 is denoted as V_0, and an edge from vertex 0 to 1



(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 2.4: Example of SSSP computations.

is denoted as E_{01}. Updates on V_4 and V_5 are used 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.

This example clearly illustrates 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 as well. Such redundancies are due to the repeated-relaxing manner [18, 74, 75], where vertices are involved in computations at multiple propagation levels (e.g., V_4 resides in levels 2 and 3).

Some other applications such as PageRank (PR) use arithmetic $sum()$ function for an aggregation process. The ranking equation is defined as follows:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v)}{L(v)} \tag{2.2}$$

This equation states that the values of all source vertices (denoted as v) in the set B_u need to be fetched to compute a destination vertex u's ranking value in each iteration. The d is a constant damping factor and N is the total number of vertices. The convergence for this kind of algorithms is defined as the property/values 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 [4].

For instance, due to the limited hardware precision, even though the $\sum PR(v)$ of two iterations are different, being divided by the same denominator (number of links, which is denoted as $L(v)$) can produce the same result. Generally, dozens to hundreds of iterations are needed to converge an entire graph.

2.4 Related Work

This dissertation aims to improve distributed graph processing via load balancing and redundancy reduction. Thus, this section first reviews state-of-the-art graph processing system designs in Section 2.4.1, and then summarizes the prior works on optimizing load balance and computational redundancy in Section 2.4.2 and Section 2.4.3, respectively.

2.4.1 Graph Processing Systems

As the magnitude of digital data grows exponentially, distributed graph processing system becomes increasingly important. Pregel [76] 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 [32, 45, 68, 71, 93, 119, 132]. Apache Giraph [48] originates as the open-source counterpart to Pregel, which is highly optimized by Facebook [33] to achieve high scalability. Some existing works like X-Stream [89] and Chaos [88] develop the edge-centric graph processing engine that sequence memory and I/O accesses. Other than the innovations on computation model, Powerlyra [32] and PowerSwitch [119] leverage hybrid partitioning (vertex/edge cut) schemes and hybrid processing engines (sync/async) to accelerate graph analytics. Trinity [97] combines graph processing and databases into a single system. Gemini [132] observes that the overhead of achieving scalability on prior distributed graph processing frameworks [32, 45] becomes a major limiting factor for efficiency with modern high-speed interconnection networks. Thus, it provides multiple optimizations targeting computation efficiency. With all these efforts, the performance of mining insightful knowledge on a large-scale graph has been significantly improved.

Other than the distributed solution, GraphChi [68] 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. Galois [80]'s programming model and library allow programmers to exploit data parallelism in graph analytics without having to write explicitly parallel code. Ligra [99] proposes two simple routines (one for mapping over edges and one for mapping over vertices) to efficiently process graph applications on a shared-memory machine. Xstream [89] deploys a novel edge-centric model to serialize data accesses for all storage media. Clip [18] reduced disk I/O by squeezing out the values of loaded data. GraphQ [114] introduced a query framework to efficiently explore its subgraphs one by one on a single PC. Compared to distributed graph processing, these prior works provide a cost-efficient alternative to analyze a large-scale graph.

2.4.2 Load Balancing for Graph Processing

In order to achieve a better performance in a distributed environment, distributed graph processing systems leverage both static and dynamic load balancing schemes to minimize the idle/waiting time of each computing unit. Static techniques aim to achieve a better data distribution before the execution, while dynamic schemes balance the workload by migrating graph data on the fly. Prior works from these two optimization areas will be reviewed separately in the following sections.

2.4.2.1 Balancing via Static Graph Partitioning

To evenly partition graph in a distributed setup, [60] provides a set of offline programs to cut data set with a high quality. However, these programs are often unsuitable for the billion edge graphs common in graph analytic, as its computational complexity cannot be easily amortized over the running time of actual application of interest. Compared to these offline partitioning schemes, online graph partitioning algorithms can split the data set quickly with a reasonable partitioning quality. Gonzalez et al. implement online random and greedy vertex-cut partitioners to minimize communications as well as ensure work balance for real world graphs following the power-law distribution [45]. Chen et al. develop a new distributed greedy heuristic hybrid-cut (vertex cut with edge cut) partitioning algorithm to mitigate poor data locality during communication [32]. Such brings a large speedup in graph workload execution time with a small increase of graph ingress time. HDRF [85] provides a stream-based partition to evenly distribute graph data and minimize the replication factor for real-world graphs. [104] and [107] splits a graph in a one-pass streaming manner, and achieves an even data distribution with a few number of cuts. However, none of these prior techniques aim to balance the data distribution in the heterogeneous environment. This dissertation provides a set of heterogeneity-aware methodologies to proportionally partition graph data among computing machines in a heterogeneous cluster.

2.4.2.2 Balancing via Dynamic Graph Partitioning

Even though the graph components can be distributed evenly in a homogeneous computing environment, load imbalance still exists in distributed graph analytcis [61, 93]. Aforementioned static graph partitioning algorithms can handle the imbalance incurred by an uneven data distribution [69, 100], they cannot alleviate the imbalance caused by the unpredictable behaviors existing in graph applications. Thus, Mizan [61] and GPS [93] attempt to eliminate the workload imbalance problem existing in distributed graph processing via dynamic load balancing methods. The common workflow of these prior works can be summarized as 1) They identify the provenance of load imbalance at the machine level, and then pinpoint problematic vertices; 2) they migrate vertices and their attributes (e.g., state, data, adjacency list, and related messages) with a decentralized strategy; 3) after migration, the ownership of a moved vertex needs to be updated in the cluster to guarantee the correctness of future computations. Executing such procedure between two computing iterations exhibits impact on the performance and resource usage. For example, recording the activity of each vertex incurs a larger memory footprint. Migrating a vertex with a large amount of associated information can significantly exacerbate the communication overhead in the distributed setup. Like other big data frameworks, these overheads are rarely amortized over the execution time of an application program [17, 69]. Compared to these prior works, this dissertation proposes to leverage a novel partitioning method to evenly distribute data as well as to prepare metadata for online load balancing. During execution, with the support of these metadata, load balance can be achieved with a negligible overhead.

2.4.3 Redundancy Optimizations for Graph Processing

Compared to other topics, redundancy optimization is a relatively new one for graph processing. Thus, there are fewer prior art focusing on reducing redundancies. Vora et al. [113] optimized GraphChi [68] 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 the shared memory environment. Kusum et al. [66] proposed a graph reduction method to improve computational efficiency of Galois [80]. 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 pre-processing is the most expensive process in the distributed systems [72, 105, 110, 132]. Other than these, Wang et al. [115] leverages articulation points to split the graph into subgraphs and optimize the betweenness centrality (BC) application to avoid redundant computations on the same subgraph. By contrast, solution proposed in this dissertation does not rely on any specific partitioning strategies or incur any extra partitioning effort, which is a system-level optimization for most graph applications in the distributed setup.

2.4.4 Other Related Optimizations for Graph Processing

Orthogonal to the load balance and redundancy optimization, graph library, algorithm, and language designs are another related approaches. CombBLAS [26] offers a distributed parallel graph library with a set of linear algebra primitives. Parallel Boost Graph Library (PBGL) [5] provides graph data structures and message passing mechanisms (MPI) to parallelize applications. These distributed libraries can deploy the techniques proposed in this dissertation to improve load balance as well as computation efficiency. As for the graph domain-specific languages (DSL), Green-Marl [53] allows developers to describe graph algorithms intuitively and expose the data-level parallelism inherent in the algorithms. Sevenich et al. [95] adopt two DSLs to enable high-level optimizations from the compiler and skip the API invocation overheads. Optimization techniques proposed in this dissertation are orthogonal to these works.

Chapter 3

Methodology

This chapter provides an outline of distributed computing clusters that are used to study the state-of-the-art graph processing systems as well as to perform the evaluations of proposed schemes of this dissertation. Moreover, this chapter also descripes the graph workloads, graph data sets, performance metrics and corresponding measurement tools used for this dissertation.

3.1 Distributed Cluster

Table 3.1 illustrates the machine configurations of computing clusters used in the dissertation to evaluate proposed techniques. The Amazon EC2 machines form the heterogeneous environment described in Chapter 4. However, such virtual machines provide a limited number of ”manipulating knobs” and do not support the energy measurement. Hence, the local Dell PowerEdge R320 servers are utilized to form a local physical cluster in the evaluation. Since the grant on Amazon EC2 expires and the Dell PowerEdge R320 servers are not up-to-the-minute anymore, the experiments in Chapter 5 and Chapter 6 are performed on TACC Stampede2 cluster. Experiments of Chapter 6 are performed before the work proposed in Chapter 5, thus, these experiments are evaluated on the Stampede2 Xeon-Phi

Table 3.1: Machine configurations of Amazon EC2 cluster, local cluster, and TACC Stampede2 Xeon-Phi cluster.

Amazon EC2 Cluster

Name	HW Threads	Memory	Network
c4.xlarge	4	7.5GB	100 Mbps to 1.86 Gbps
c4.2xlarge	8	15GB	100 Mbps to 1.86 Gbps
m4.2xlarge	8	32GB	100 Mbps to 1.86 Gbps
c4.4xlarge	16	30GB	100 Mbps to 1.86 Gbps
c4.8xlarge	36	60GB	up to 8.86Gbps

Dell PowerEdge R320 Cluster

Name	HW Threads	Memory	Network
Xeon_Sever_S	4	64GB	up to 10Mbps
Xeon_Sever_L	12	64GB	up to 10Mbps

TACC Stampede2 Cluster

Name	HW Threads	Memory	Network
Xeon-Phi	68	96GB of DDR4 RAM + 16GB MCDRAM	up to 100Gbps
Xeon-Skylake	48	192GB of DDR4 RAM	up to 100Gbps

cluster (only Xeon-Phi cluster is available at the moment). Later, the Stampede2 cluster equips the powerful Xeon-Skylake server with a high DRAM capacity and a high network bandwidth. Therefore, evaluations in Chapter 5 are moved to the Skylake server cluster. To sum up, the philosophy of forming experimental computer clusters is to use state-of-the-art machines at the moment.

3.2 Graph Workloads

The techniques presented in this dissertation are evaluated across a number of graph processing workloads. These graph applications are implemented atop the graph processing system. This section provides a basic overview of these selected algorithms.

PageRank (PR): The PageRank algorithm [83] computes the relative rating of a node based on the weights of all the connected nodes. It's main use is in ranking the importance of web pages on the internet and is defined as follows:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v)}{L(v)} \tag{3.1}$$

This equation states that the PageRank for a page u depends on the ranking values for each page v contained in the set B_u, the set containing all the pages linking to v divided by the number of links from page v. The damping factor, d, reduces the impact of any one page on another page's rank. The total number of pages is represented by N.

TunkRank (TR): TunkRank [116] extends PageRank algorithm and is used to calculate the influence of `twitterer` recursively as:

$$Influence(X) = \sum_{Y \in Followers(X)} \frac{1 + p + Influence(Y)}{Friends(Y)}, \tag{3.2}$$

where `p` is the constant probability that twitterers retweet a tweet. TunkRank measures twitterer `X`'s influence as the expected number of twitterers who will read a tweet that he/she publishes.

Single-Source-Shortest-Path (SSSP): Given a graph and a root vertex, Single-Source-Shortest-Path algorithm can find the shorest paths from root to all vertices in the graph. The distance of each vertices (except root) are initialized to ∞, while the root is initialized to 0. The implementation of this algorithm follows the propagation manner, where the updated value of all active vertices are iteratively sent to neighboring vertices until no active vertex exists anymore.

Single-Source-Widest-Path (WP): Similar to SSSP, Single-Source-Widest-Path algorithm attempts to find the widest path between a give root vertex and other vertex in the graph. Different, the edge weight of a given graph is considered as the width rather than the distance used in SSSP.

Connected Components (CC): The connected components algorithm attempts to determine the number of components that are connected in a graph and the number of vertices and edges in each connected component. CC is implemented as a simple label propagation algorithm that iterates until the vertex label identifiers are no longer changing.

Triangle Count (TC): Triangle count counts the total number of triangles in a graph, and also counts the number of triangles associated with each vertex. For every edge (u,v) in the graph, this application [94] counts the number of intersections of the neighbor set on u and the neighbor set on v. This counts every triangle 3 times, so the final answer can be obtained by summing across all the edges and dividing by 3.

3.3 Graph Data Sets

Table 3.2 shows a number of different real-world graph and large-scale synthetic data sets used in experiments. The data sets range in size from a few million edges (e.g., amazon) to 10 billions (e.g., RMAT3). The algorithms applied on these graphs are listed in the last column of Table 3.2. Since the computer cluster size used in Chapter 4 is relatively small, only four data sets (i.e., amazon, s-wiki, citation, and livejournal) are used in the experiments. The experiments of Chapter 5 utilizes four real-world graphs (i.e. from livejournal to twitter) and three large-scale synthetic graphs (i.e., RMAT1-RMAT3) in the evaluation. The experiments

Table 3.2: The graph Data Sets [63, 65, 70] used in experiments

Real graph	**Vertices**	**Edges**	**Size**	**Algorithms**
amazon	403,394	3,387,388	46MB	PR,CC,TC
s-wiki	2,394,385	5,021,410	64MB	PR,CC,TC
citation	3,774,768	16,518,948	268MB	PR,CC,TC
pokec	1,632,803	30,622,564	405MB	PR,TR,CC,SSSP,WP
orkut	3,072,411	117,184,899	1.7GB	PR,TR,CC,SSSP,WP
delicious	33,801,712	301,183,605	3.7GB	PR,TR,CC,SSSP,WP
wiki	12,150,976	378,142,420	5.8GB	PR,TR,CC,SSSP,WP
s-twitter	11,316,812	85,331,845	1.3GB	PR,TR,CC,SSSP,WP
livejournal	4,847,571	68,993,773	1.1GB	PR,TR,CC,TC,SSSP,WP
friendster	65,608,366	1,806,067,135	31GB	PR,TR,CC,SSSP,WP
weibo	58,655,849	261,321,071	3.8GB	PR,TR,CC,SSSP
twitter	52,579,682	1,963,263,823	28GB	PR,TR,CC,SSSP
Synthetic graph	**Vertices**	**Edges**	**Size**	**Algorithms**
RMAT1	100,000,000	2,000,000,000	33GB	PR,TR,CC,SSSP,WP
RMAT2	300,000,000	6,000,000,000	104GB	PR,TR,CC,SSSP,WP
RMAT3	500,000,000	10,000,000,000	150GB	PR,TR,CC,SSSP,WP

of Chapter 6 includes seven real-world graphs (i.e., from pokec to friendster) and three large-scale synthetic graphs (i.e., RMAT1-RMAT3).

3.4 Performance Metrics and Measurement Tools

Runtime and Energy Runtime is the main performance metric of a distributed graph processing system. In this dissertation, the runtime of graph pre-processing phase and graph workload execution phase are reported to demonstrate the effectiveness and overhead of proposed solutions. The runtime metric in the experiments is measured by *gettimeofday()* system function. The unit of this metric is generally *Second*. Other than the runtime metric, energy is also measured as a metric to compare proposed solutions with prior works. This metric is measured via Intel RAPL counters [6], and the unit is *Joule*.

Hardware Metrics Hardware metrics are deployed in this dissertation to further study and understand the performance gain brought by proposed schemes. Among all the hardware metrics, the number of instructions and memory accesses are selected to be profiled by *Perf* [11]. Since the experiments are performed in a distributed cluster, *ibrun* was utilized to launch the *Perf* measurement along with experiments on each machine. With the same graph algorithm and data set, a lower number of instructions indicates the proposed solution accomplish the job in a more efficient manner. Due to the fact that most graph algorithms are memory intensive, less memory accesses generally leads to a higher performance.

System Metrics Other than the metrics discussed above, some system metrics are also recorded in the experiments. Network traffic is one of the most important metrics that can heavily affect the performance of a distributed system. To quantify this, the number of bytes that have been sent/received are measured. In addition, memory footprint is profiled via the Valgrind tool [79] to indicate the peak memory usage of the system. In order to examine the load balance situation, [16] is utilized to record and visualize the detailed execution procedure of each machine. The visualized idle time of each machine before the synchronization barrier is considered as the wasted time due to workload imbalance.

Chapter 4

HAP: Heterogeneous-aware Partitioning for Distributed Graph Processing[2] [3]

Large-scale distributed graph processing is an increasingly important computational problem, spurred on by the popularity of cloud and big data computing. The ever reducing price of storage has enabled servers and data farms to collect and retain massive data sets, much of which can be expressed as graphs. As such, researchers have developed many computational frameworks to address the specific needs of distributed graph algorithms, such as GraphLab [71], PowerGraph [45], Pregel [76], Giraph [48], and Gemini [132].

Novel computational frameworks, however, are not the only change induced by big data and cloud computing. To cheaply provide scale-out performance, data

[2]M. LeBeane, S. Song, R. Panda, J. Ryoo, and L. K. John, "Data Partitioning Strategies for Graph Workloads on Heterogeneous Clusters", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015. Shuang Song contributed to the idea development and implementation. Michael LeBeane was involved in implementation. Reena Panda and Jee Ho Ryoo were involved in experimental setup. Lizy K. John supervised the overall project.

[3]S. Song, M. Li, X. Zheng, J. Ryoo, R. Panda, M. LeBeane, A. Gerstlauer, and L. K. John, "Proxy-Guided Load Balancing of Graph Processing Workloads on Heterogeneous Clusters", in *Proceedings of 45th International Conference on Parallel Processing*, 2016. Shuang Song contributed to the idea development and implementation. Meng Li, Xinnian Zheng, Reena Panda, and Jee Ho Ryoo were involved in performance evaluation and technical writing. Andreas Gerstlauer and Lizy K. John supervised the overall project.

centers are evolving from expensive enterprise servers to networks of off-the-shelf commodity parts. This trend has opened the door to data centers populated with *heterogeneous compute units* for a variety of economic and performance related reasons. For example, many data centers with commodity components will upgrade in a piecemeal fashion according to their needs, leaving a variety of compute units available. Heterogeneity can also be deliberately introduced to service the needs of different types of workloads. Most data centers serve the needs of a variety of users and clients, and will provide different system configurations to target various performance and power points [34][44]. Virtualized environments (e.g. Amazon EC2 [1]) can also impose heterogeneity by partitioning clusters of homogeneous machines into a variety of configurations. Such partitionings of otherwise homogeneous machines resemble actual physical heterogeneous clusters. Virtualization also inadvertently introduces heterogeneous performance due to shared multi-tenancy nodes. Finally, heterogeneity can be introduced by means of attaching accelerators (e.g. GPGPUs or Intel Phi [2] accelerators) to existing CPU clusters, although dedicated accelerators are beyond the scope of the current work.

Despite the ever growing prevalence of heterogeneous clusters, most graph analytics frameworks operate under the assumption that all compute nodes are balanced in performance. As discussed in Section 1.1, this assumption leads to an imbalance of data distribution. For graph analytics applications which require synchronization between iterations of an algorithm, the slow nodes result in low average processor occupancy and huge processing inefficiencies. The motivational data has shown an 8 core machine spends more than 40% of its execution time waiting at a barrier for a 4 core processor to complete an iteration over its local graph. To solve this problem, ideally, the graph partitioning algorithm should correctly skew the data in accordance with the processing capability of the computing nodes so that all machines reach the barrier at approximately the same time.

This chapter describes Heterogeneity-aware Partitioning (*HAP*) that aims to balance the data distribution of distributed graph processing in heterogeneous environments. *HAP* enhances several state of the art static partitioning algorithms to account for the affects of heterogeneous data centers on graph analytics. These partitioning algorithms are implemented as pluggable edge/vertex cut methods in PowerGraph's streaming graph partitioner. These partitioners are guided by a metric that describes the desired data load on each node, which is defined as the skew factor. *HAP* also proposes a number of ways that the skew factor of a cluster can be estimated and used to guide partitioning decisions. Specifically, the contributions of *HAP* to the state of the art include the following:

1. *HAP* proposes a number of simple heterogeneity estimation methodologies. These estimations are used to develop the skew factor and to guide the graph splitting decisions offered by the partitioning algorithms.

2. *HAP* offers a number of graph partitioning strategies to improve data-ingress on heterogeneous clusters. While dynamic load balancing techniques can account for data center heterogeneity [61] [93], *HAP* shows that a simple partitioning can perform very well. In addition, the proposed static partitioning algorithms in *HAP* are orthogonal to the dynamic load balancing techniques.

Table 4.1: Amazon c4-type virtual node configurations.

Amazon EC2 Configuration

Name	HW Threads	Memory	Network [1]
c4.xlarge	4	7.5GB	100 Mbps to 1.86 Gbps
c4.2xlarge	8	15GB	100 Mbps to 1.86 Gbps
c4.4xlarge	16	30GB	100 Mbps to 1.86 Gbps
c4.8xlarge	36	60GB	up to 8.86Gbps

3. An in-depth evaluation on heterogeneous clusters built from the Amazon EC2 virtualized cloud environment and local computing machines are performed to evaluate *HAP*. Compared to the standard online data partitioning algorithms, *HAP* improves application execution time over a variety of real-world data sets and algorithms.

4.1 Estimating Heterogeneity in a Cluster

The main idea in data partitioning for heterogeneous clusters is simple: dividing the input data set into shards such that each machine receives an amount of data in accordance with a metric. The skew factor is defined as the desired data partitioning ratio of the cluster. In this chapter, the skew factor is always written relative to other nodes, with the least powerful node receiving the value of 1. This section provides two simple and tractable methods of representing the skew factor of a heterogeneous cluster. To ground the discussion, a number of Amazon EC2 machines will be used to demonstrate the calculation of skew factor. Table 4.1 shows the machine configurations.

4.1.1 Thread-based Estimation

For graph data sets where the graph can easily fit into the memory of any single node, a reasonable proxy of performance among machines in the cluster can be derived by looking at the relative performance of the CPUs. If the CPU type is the same and only the number of hardware threads assigned to each node (e.g., virtual machine) varies across configurations, the number of worker threads can be utilized to define the skew factor of a heterogeneous cluster. For instance, in PowerGraph, the number of logical cores reserved for computation $(num\_logical\_cpus - 2)$ is used to calculate relative throughput. Two logical cores are reserved for communication, hence, not used to compute the skew factor. As shown in Figure 4.1, the skew factors of c4-type machines estimated based on the number of threads are 1:3:7:17. This estimation method is referred as thread-based method.

4.1.2 Profiling-based Estimation

Besides the thread-based method, a lightweight profiling-based method is also proposed to develop the skew factor. Figure 4.1 shows, compared to thread-based estimation, the skew factors of c4-machines guided by profiling behave differently and vary across applications. Moreover, profile can cover some cases that thread-based method is not applicable. For instance, when the heterogeneous machines in the cluster have same thread count. Thread-based one will not be able to reveal the heterogeneity. However, profiling-based estimation can still indicate the difference between machines.

Synthetic proxy graphs are utilized to capture a machine's computation

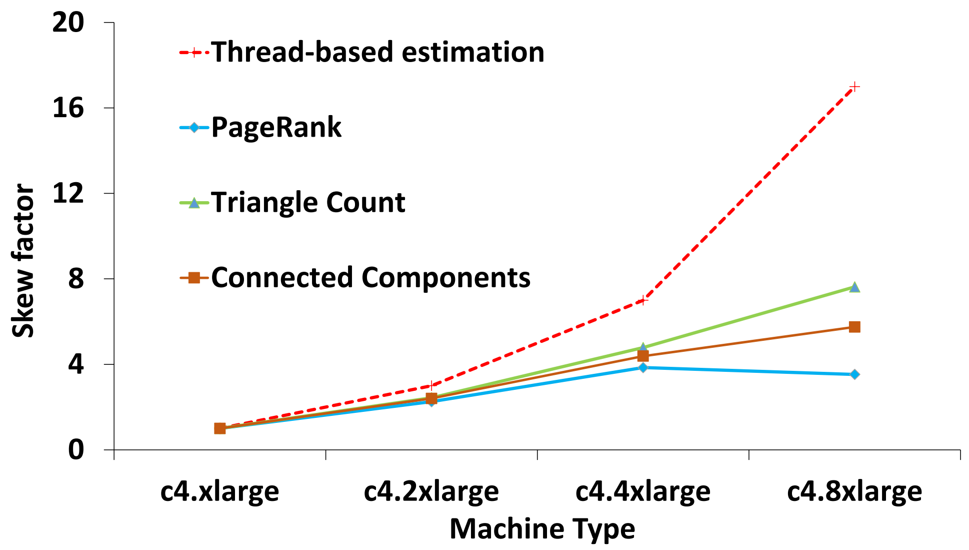


Figure 4.1: Thread-based estimation vs. profiling-based estimation (PageRank, Triangle Count, and Connected Components are estimated by the profiling method.

power on a graph workload. To do so, the synthetic graphs with diverse distributions are generated. Algorithm 1 shows the pseudo-code to generate the synthetic proxy graphs following the power-law distribution. It takes the number of vertices N and α parameter as inputs. Based on distribution factor α, the probability of each vertex is calculated and associated with the number of degrees that will be generated later. Then, the probability density function (pdf) will be transformed into a cumulative density function (cdf). The total number of degrees of any vertex is generated by the cdf function. All the connected vertices are produced by a random hash. If directional edges are needed, the order of $edge(u,v)$ could be understood as the graph having an edge from u to v and vice verse. To omit self-loops, a condition check on vertex u being unequal to vertex v is added in the process, if necessary.

Algorithm 1 Synthetitc Graph Generator

```
procedure GRAPH GENERATION
    for $i \leq N$ do
        $pdf[i] = i^{-\alpha}$
    end for
    $cdf = transform(pdf)$
    $hash = constant\_value$
    for $u \leq N$ do
        $degree = multinomial(cdf)$
        for $d \leq degree$ do
            $v = (u + hash) \bmod N$
            $output\_edge(u, v)$
        end for
    end for
end procedure
```

The overhead of generating synthetic graphs depends on the graph size and distribution. To cover a broad range of real-world graphs [45], three proxy graphs (with distribution factor αs ranging from 1.95 to 2.4) are generated. Generating these deployed proxies took 67 seconds in total.

These generated synthetic graph will be combined with each graph application to form an independent profiling set. For a give heterogeneous cluster, machines are classified into different groups and only one machine from each group needs to be profiled. After profiling, the obtained performance of each machine is used to estimate the skew factor for each application on a given cluster. As the modern cluster's heterogeneity is increasing (e.g., operating frequency of machine also varies), profiling can more accurately reflect the heterogeneity. Overall, compared to the thread-based method, profiling-based estimation incurs a higher overhead (i.e., proxy graph generation and profiling cost) with more accurate estimation and corner cases covered.

4.2 Problem Formulation

Now that with a method of estimating the skew factor for a given cluster, the requirements of a graph partitioning algorithm can be formulated as a vertex cut problem. A similar formulation can be derived for an edge cut objective as well. Let E and V be the set of all edges and vertices contained in the graph, respectively. The problem of partitioning edges onto heterogeneous nodes can be expressed as a n-way **vertex-cut** that assigns each edge $e \in E$ to a machine $A(e) \in P$ where P is the set of all machines. Each vertex then spans the set of machines $A(v) \subseteq P$ that contain its adjacent edges. I also formally define Sk such that $Sk(p) = \frac{skewfactor(p)}{sum(skewfactor)}$ where the skew factor can be calculated as discussed in Section 4.1. Therefore, $Sk(p)$ is the relative throughput associated with machine p expressed between $[0,1]$. Formally, the vertex cut objective can be expressed similarly to [45]:

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)| \tag{4.1}$$

$$\text{s.t. } \forall p \in P, \textbf{abs}(|\{e \in E : A(e) = p\}| - Sk(p) * |E|) < \lambda \tag{4.2}$$

where λ is defined as the **imbalance factor**. The number of **replicas** of a vertex v is defined as the $|A(v)|$ copies of the vertex v. Therefore, Equation 4.1 attempts to place edges for a given vertex v on a small number of machines, minimizing the communication and memory overhead. Equation 4.2, on the other hand, attempts to balance the distribution of edges over all available machines according to the relative throughput of each node, expressed by *Sk*. Any partitioning algorithm should strive to account for both equations to achieve a quality *n*-way heterogeneous vertex cut.

Algorithm 2 Heterogeneity-aware Random Hash

```
1: procedure RANDOM
2:     for $e \in E$ do
3:         $s \leftarrow Src(e)$                          ▷ s is source vertex of edge e
4:         $d \leftarrow Dest(e)$                         ▷ d is destination vertex of edge e
5:         $p \leftarrow iSk[\frac{HashE(s,d)}{maxHash}]$     ▷ compute skewed hash
6:         $e.owner \leftarrow p$                         ▷ assign owning node for e
7:     end for
8: end procedure
```

4.3 Heterogeneity-aware Partitioning Algorithms

This section presents five online static graph partitioning algorithms [32, 45, 76] that have be modified to support optimized heterogeneous data placement using the skew factor. These five algorithms (three vertex cut algorithms and two mixed cut ones) contain a mix of the most commonly used streaming split algorithms, as well as newly proposed research techniques. The skew factor can be constructed using either the thread-based or profiling-based approach as illustrated in Section 4.1. For the explanation of the algorithms, the *Sk* is used as defined in Section 4.2. Additionally, *iSk* is used and defined as the inverse cumulative density function of the skew factors. Indexing into *iSk* with a probability $[0,1]$ returns a node id based on the cumulative density of the skew factor. The inverse cumulative density

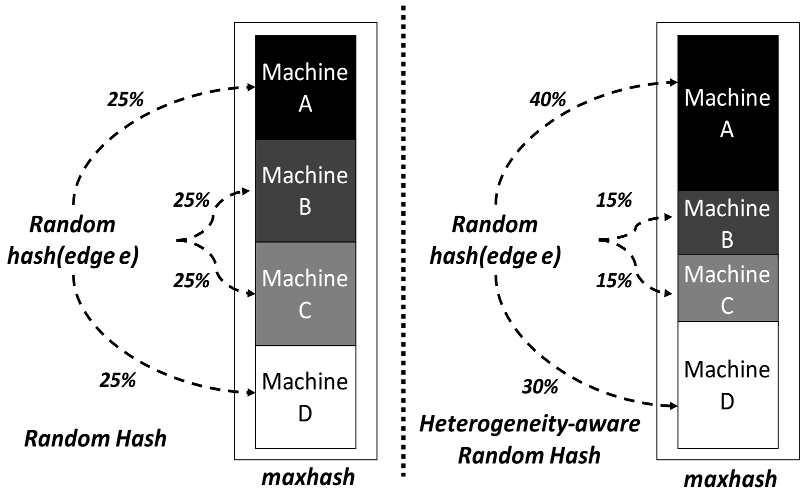


Figure 4.2: Random Hash vs. heterogeneity-aware Random Hash.

function is trivially constructed from the skew factor or *Sk*.

4.3.1 Heterogeneity-aware Random Hash

The random vertex cut (Random) was originally proposed in the PowerGraph [45] framework as a baseline method for extremely fast partitioning. It attempts to assign an edge to a node based on a random hash of the source and destination vertices. Algorithm 2 describes the formulation of Random for a heterogeneous environment.

The formulation for a heterogeneous environment is quite simple. Skewed Random is expressed as a probability and now indexes into *iSk* to produce a weighted assignment of edges to machines based on the skew factor. The effect of the weighted edge assignment is shown in Figure 4.2, where the machine with a higher skew factor value (higher computational power) will receive more edges in the heterogeneity-aware random hash partitioning.

4.3.2 Heterogeneity-aware Oblivious

The oblivious vertex cut algorithm was originally proposed in PowerGraph [45] as an improvement over the Random algorithm. It attempts to factor locality into the graph partitioning decision by assigning edges to nodes based on prior scheduling decisions at the expense of a longer partitioning phase. Algorithm 3 describes the formulation of oblivious for a heterogeneous environment.

Algorithm 3 Heterogeneity-aware Oblivious

```
1:  procedure OBLIVIOUS
2:      for $e \in E$ do
3:          $s \leftarrow Src(e)$
4:          $d \leftarrow Dest(e)$
5:          for $p \in P$ do
6:              $srcPresence \leftarrow (Edges(s,p) > 0)$
7:              $destPresence \leftarrow (Edges(d,p) > 0)$
8:              $bal \leftarrow Balance(Sk,p)$
9:              $score[p] \leftarrow bal + srcPresence + destPresence$
10:         end for
11:         $e.owner \leftarrow argmax_p(score)$
12:     end for
13: end procedure
```

The heterogeneous formulation of the algorithm attempts to assign edges to nodes that already contain either the source or the destination vertices. In the pseudo-code, the $Balance()$ function assigns a $[0-1]$ score based on how the current distribution of edges deviates from the ratio of edges suggested by $Sk[p]$. Let u and v be the vertices associated with an edge e, and let $A(u)$ be defined as in Section 4.2. Heterogeneous-aware Oblivious cut uses the following modified heuristics from [45], applied in this order:

1. If $A(u)$ and $A(v)$ intersect, then e should be assigned to a machine in the intersection biased by the skew factor.
2. If $A(u)$ and $A(v)$ are not empty, then e should be assigned to the machine containing one of the vertices biased by the skew factor.
3. If only one of the two vertices has been assigned to a machine, then choose a machine for e from the assigned vertex biased by the skew factor.
4. If neither vertex has been assigned, then assign e to the least loaded machine biased by the skew factor.

Note that since this algorithm is a heuristic, it does not guarantee an exact balance in accordance with the skew factor, as the Random Hash does.

4.3.3 Heterogeneity-aware Grid

The Grid vertex cut partitioner is designed to limit the communication overheads by constraining the number of candidate machines for each assignment. The number of machines in the cluster has to be a square number, as they are used to form a square matrix grid as displayed in Figure 4.3. A shard is defined as a row or column of machines in this context. Similar to the concept of heterogeneous Random Hash, each shard has its weight, which is determined from the weights of machines in the shard. Differently, every vertex is hashed to a shard instead of

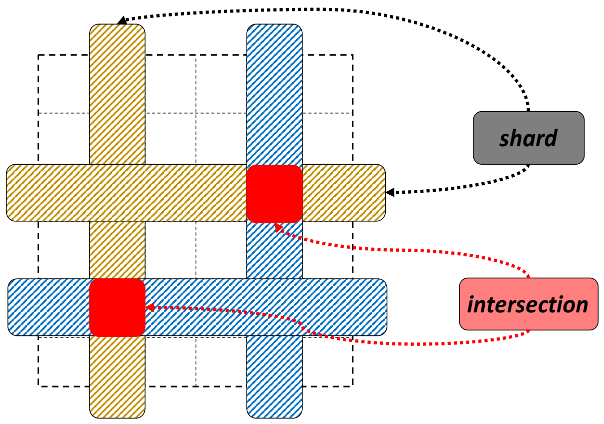


Figure 4.3: Illustration of Machine Grid and Shards

single machine. For each edge, two selected shards corresponding to the source and target vertices generate an intersection. Considering the current edge distribution and the edge placements suggested by skew factor, each machine in the intersection receives a score. The edge will be allocated to the machine with the maximum score.

Heterogeneity-aware Hybrid The Hybrid cut algorithm [32] is similar to Random with one important difference. Hybrid attempts to perform both vertex and edge cuts depending on the average degree of the vertex in question. It employs a two pass approach to accomplish this objective. The first approach assigns all edges to nodes based on a hash of the destination vertex, essentially performing a random edge cut. More importantly, however, the first pass allows for the easy calculation of the total degree of each vertex. The second pass finds all the vertices with an in-degree higher than a threshold and reassigns them similarly to the random hash

Algorithm 4 Heterogeneity-aware Hybrid

```
1: procedure HYBRID
2:     for $e \in E$ do
3:         $d \leftarrow Dest(e)$
4:         $p \leftarrow iSk[\frac{HashV(d)}{maxHash}]$
5:         $e.owner \leftarrow p$
6:     end for
7:     for $v \in V$ do
8:         if $inDegree(v) > Threshold$ then
9:             for $e \in Edges(v)$ do
10:                $s \leftarrow Src(e)$
11:                $p \leftarrow iSk[\frac{HashV(s)}{maxHash}]$
12:                $e.owner \leftarrow p$
13:            end for
14:        end if
15:    end for
16: end procedure
```

methodology. Algorithm 4 describes the heterogeneity-aware Hybrid cut algorithm.

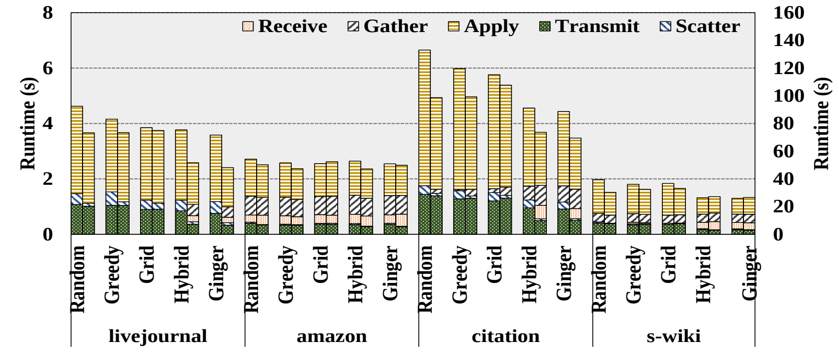
Heterogeneity-aware Hybrid modifies both phases of Hybrid cut assignment in a similar manner as heterogeneity-aware Random. Both random hashes are modified to index into *iSk* to produce a weighted assignment of edges to machines based on the relative throughput of the nodes.

Heterogeneity-aware Ginger The Ginger cut partitioning algorithm [32] is an extension of Hybrid cut enhanced with a locality heuristic called Fennel [107]. For high degree vertices, it operates like Hybrid cut. For low degree vertices, Ginger minimizes the expected value of the replication factor. Let V_p represent the set of vertices that are assigned to node p. Formally, a low-degree vertex v is assigned

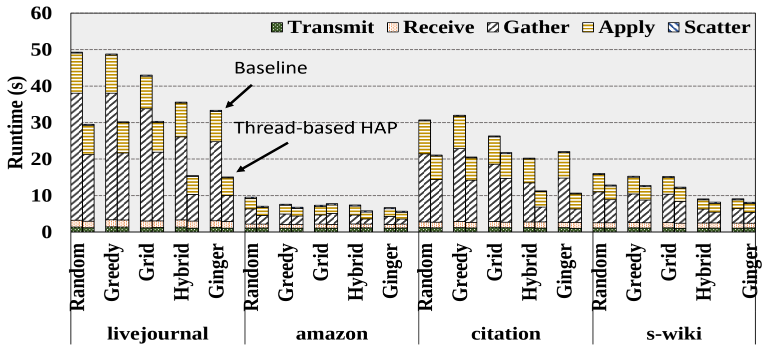
Algorithm 5 Heterogeneity-aware Ginger

```
1:  procedure GINGER
2:      for $e \in E$ do
3:          $d \leftarrow Dest(e)$
4:          $p \leftarrow iSk[\frac{HashV(d)}{maxHash}]$
5:          $e.owner \leftarrow p$
6:      end for
7:      for $v \in V$ do
8:          if $inDegree(v) > Threshold$ then
9:              for $e \in Edges(v)$ do
10:                 $s \leftarrow Src(e)$
11:                 $p \leftarrow iSk[\frac{HashV(s)}{maxHash}]$
12:                 $e.owner \leftarrow p$
13:             end for
14:         else
15:             for $p \in P$ do
16:                 $V_p \leftarrow Verts(p)$
17:                 $cost[p] \leftarrow \left|N(v) \cap V_p\right| - (1 - Sk[p]) * b(p)$
18:             end for
19:             $e.owner \leftarrow argmax_p(cost)$
20:         end if
21:     end for
22: end procedure
```

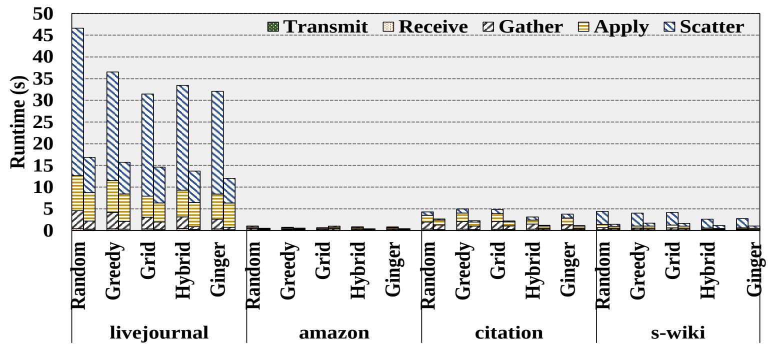
to node i such that $c(v,p) > c(v,j), forall\, j \in P$, where $c(v,p)$ is the cost function. The cost function is defined as $c(v,p) = \left|N(v) \cap V_p\right| - b(p)$, where $N(v)$ denotes the set of neighboring vertices along the in-edges of v. The first term, $\left|N(v) \cap V_p\right|$ represents the degree of vertex v in the candidate partition p. The balance formula $b(p)$ represents the marginal balancing cost of adding vertex v to node p and is represented by a normalized factor considering both the number of edges and vertices assigned to a node: $\frac{1}{2}(\left|V_p\right| + \frac{|V|}{|E|} * \left|E_p\right|)$. Algorithm 5 describes the formulation of Ginger cut for a heterogeneous environment.



(a) Connected Components runtime



(b) PageRank runtime



(c) Triangle Count runtime

Figure 4.4: Runtime analysis for graph applications and data sets under thread-based *HAP* and default partitionings (Heterogeneous cluster is formed by Amazon EC2 c4.xlarge, c4.2xlarge, c4.4xlarge, and c4.8xlarge). The thread-based *HAP* decreases application runtime by as much as 69%, and on average 31%.

The heterogeneity-aware Ginger algorithm modifies the first pass similarly to Skewed Hybrid by using a weighted hash on the destination vertex to assign an edge. The second pass for high degree vertices is also the same as Hybrid cut, using a weighted hash on the source vertex to assign an edge. The primary difference is for the second pass on low degree vertices, which now uses a modified version of the Fennel heuristic. The balance heuristic $b(p)$ is multiplied by $1 - Sk[p]$ to favor node assignments more in line with the relative throughput of each node. Similarly to the Obilivous cut algorithm, this algorithm employs a heuristic which can generate partitions that slightly deviate from the requested skew factor.

4.4 Evaluation

The proposed *HAP* methodologies are evaluated on the heterogeneous cluster formed by Amazon EC2 c4 virtual machines and Dell PowerEdge R320 servers (detailed machine configuration is shown in Section 3.1). The five proposed heterogeneity-aware partitioning algorithms are implemented on top of the PowerGraph [45] system. Hence, the baseline used in this section is the default performance of PowerGraph. Three popular graph applications (PageRank, Triangle Count, and Connected Components) and four real-world graphs (livejournal, amazon, citation, and s-wiki) are deployed to comprehensively evaluate *HAP*.

Experiment Outline Section 4.4.1 compares the performance of thread-based *HAP* (skew factor generated by the thread-based estimation) and default partitioning scheme when applied to three graph applications. Meanwhile, it also presents the overhead of incorporating the heterogeneity-aware principal in the ingress phase. Section 4.4.2 demonstrates the static edge distribution after applying the HAP algorithms. Section 4.4.3 compares performance of graph applications guided by the thread-based and profiling-based *HAP*. Lastly, Section 4.4.4 shows the cost efficiency of various Amazon EC2 machines on graph analytical workloads, which is the extra benefit brought by profiling the synthetic proxy graphs.

4.4.1 End-to-end Performance Evaluation

Figure 4.4 shows the runtime of each *HAP* partitioner and the unmodified version when applied to several graph applications and data sets. The height of the bar illustrates runtime, which is broken down into the time spent in each phase of the algorithm. These phases are the gather/apply/scatter phases that compose the majority of graph computation, and the transmit and receive phases, which represent the amount of time that the compute threads are putting/getting data into its RX/TX buffers for the network threads.

The PageRank algorithm illustrated in Figure 4.4b shows an average improvement in runtime of 25%, with the Hybrid and Ginger demonstrating the best performance. The Triangle Count application, presented in Figure 4.4c, has an average improvement in runtime of 48%. This application also displays a strong amount of data dependent performance, with the livejournal graph taking much longer than the others. Livejournal is a power-law graph with a small number of extremely high degree vertices, which stresses the triangle counting algorithm. Finally, Connected Components (Figure 4.4a) shows an average improvement in runtime of

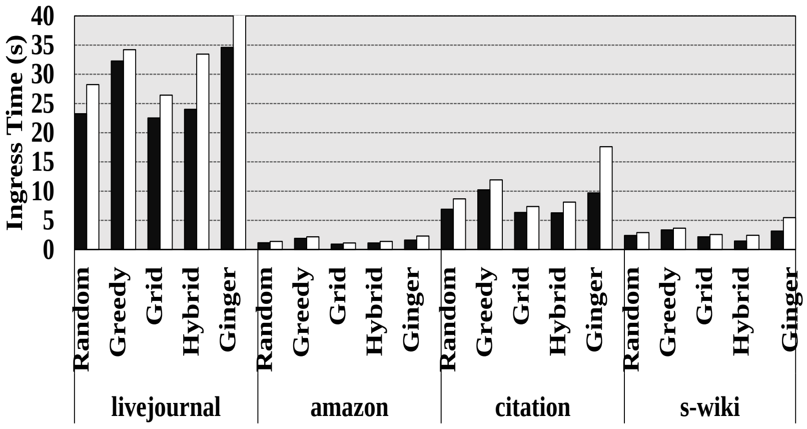


Figure 4.5: Overhead of the ingress partitioning techniques. *HAP* incurs approximately 20% ingress overhead, which will amortize over the execution of most non-trivial applications.

25%. The road map data set takes significantly longer to process due to the unique nature of road topology, which differs significantly from the other graphs. Overall, the thread-based *HAP* algorithms net an improvement of 31% over their respective baseline across all algorithms and data sets.

Besides the performance improvement, it is also important to measure the amount of time spending on partitioning/ingressing. Higher quality partitions usually take more time than naive algorithms. Figure 4.5 shows how data ingress is affected by heterogeneity across data sets. In general, Random and Grid partitioning schemes are the quickest, and Greedy and Ginger are the slowest. Not counting Ginger, the heterogeneous algorithms increase the data ingress time over their respective baseline by approximately 22%. Almost all of the runtime overhead added by the heterogeneous variants arises from the cost of computing a weighted hash. While this is done efficiently in code, the calculation is performed at least once for

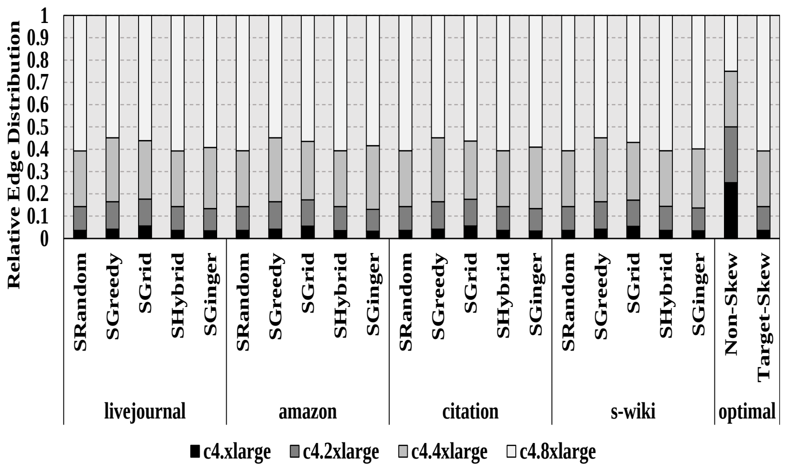


Figure 4.6: Relative distribution of edges to nodes for graphs data sets.

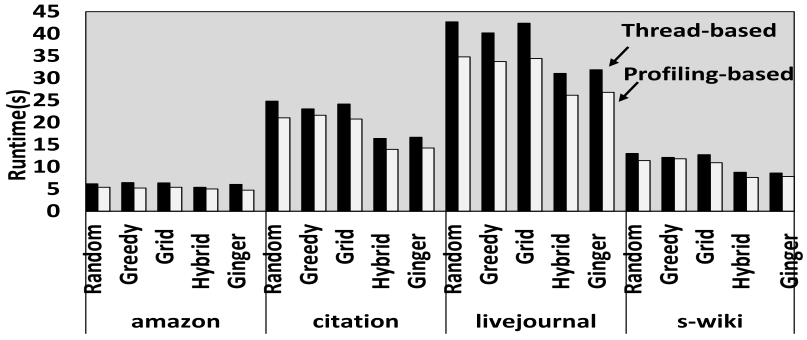
every new edge processed. The heterogeneous formulation of Ginger, however, has a relatively high overhead and increases the ingress time by 66%, due to more calls into the weighted hash function attributed to its multi-pass approach. While these numbers might seem high, the real cost of data ingress must be measured with respect to the application program. If the application program runtime is greater than the data ingress phase, then the cost of ingress will be amortized over the application program run, and the benefits during application phase from the proposed approach will exceed the extra computation needed to load the data.

4.4.2 Load Balance

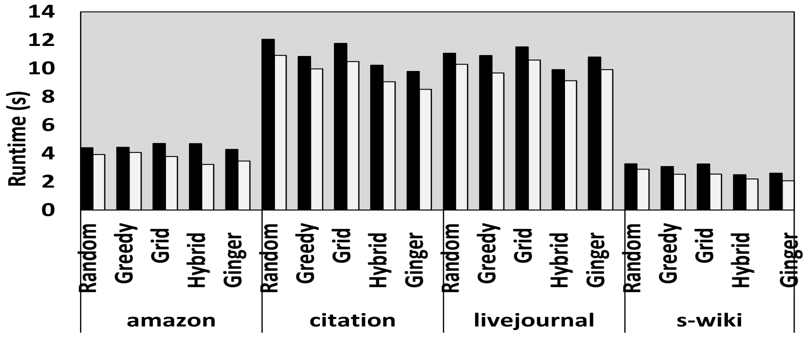
To obtain the optimal benefit in a heterogeneous cluster, it is important that the partitioning strategies achieve the desired load balance as mentioned in Section 4.1. However, many of these strategies are heuristics and do not guarantee a perfect data distribution. Therefore, it is important to study how the choice of partitioning algorithm and data set can affect the load balance. Figure 4.6 shows the balance of edge distributions among different data sets and partitioning strategies when compared to the targeted edge distribution and the default homogeneous partitioning approaches. The two bars on the right side of the graph illustrate the original homogeneous load balance objective and the optimal/target load balance objective. Algorithms that are based on a random hash of the edges (such as Random and Hybrid) achieve a near-perfect load balance of edges in accordance with estimation. Algorithms based on heuristics (such as Greedy, Grid, and Ginger), do not perfectly achieve the target skew, but still maintain a fairly reasonable load balance. There is not a significant amount of dependence between the choice of data set and the load balance, with consistent trends across all data sets. Overall, most algorithms illustrate good adherence to the estimated heterogeneity.

4.4.3 Thread-based vs Profiling-based *HAP*

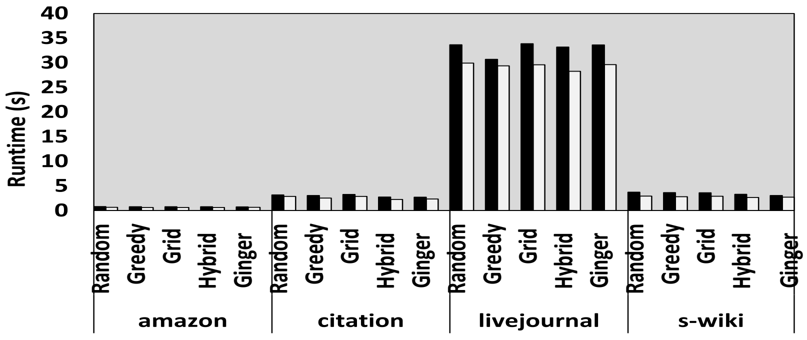
To compare the quality of thread-based and profiling-based *HAP*, a small heterogeneous cluster is formed by an Amazon EC2 c4.2xlarge and m4.2xlarge machine. The Amazon c4.2xlarge and m4.2xlarge have the same thread count, however, profiling-based estimation indicates that the skew factor of this cluster is approximately 1:1.21, where m4.2xlarge has a slightly weaker computational power. Figure 4.7 shows, compared to thread-based *HAP*, profiling-based partitioning results in an average of 1.17$\times$ speedup across three applications with four real-world graphs. Moreover, to mimic a more complex heterogeneous cluster, a small



(a) PageRank runtime

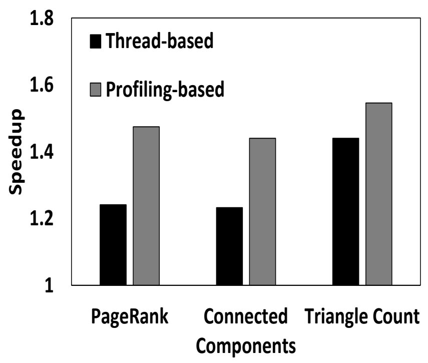


(b) Connected Components runtime

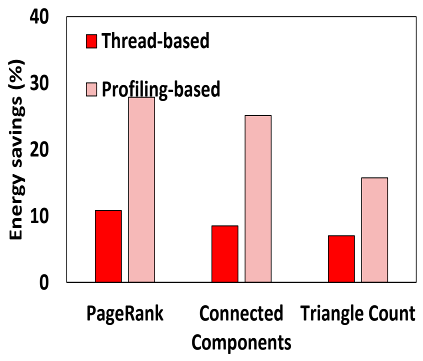


(c) Triangle Count runtime

Figure 4.7: Performance comparison of partitioning algorithms guided by thread-based and profiling-based estimations on a heterogeneous cluster formed by Amazon EC2 c4.2xlarge and m4.2xlarge machines. Baseline system uses the default partitioning.



(a) Performance comparison.



(b) Energy cost comparison.

Figure 4.8: Performance and energy improvements of thread-based and profiling-based *HAP* on Dell PowerEdge R320 clusters. Baseline system uses the default partitioning.

local cluster (configurations are shown as "Dell PowerEdge R320 Cluster" in Table 3.1) is formed to measure both the performance and energy cost of thread-based and profiling-based *HAP* partitionings (Amazon EC2 clusters do not support energy measurement and frequency manipulation). The fast machine has 12 cores running at 2.5Ghz, while the little machine only has 4 cores with a maximum frequency of 1.8Ghz. Not surprisingly, the profiling-based skew factor estimation changes substantially. The PageRank and Connect Components become more than 1 : 6. Different from the significant changes in these two applications, Triangle Count's skew factor ratio increases from 1 : 3.1 to 1 : 4.5, and it becomes quite similar to the partition ratio suggested by thread-based method. Therefore, Figure 4.8 shows that both the runtime improvement and energy reduction of this application is similar to what thread-based one achieves. On average, the profiling-based *HAP* achieves 1.49x speedup and 22.9% energy savings over the default homogeneous partitionings. This presents an average of 15% speedup and 14.1% energy reduction over

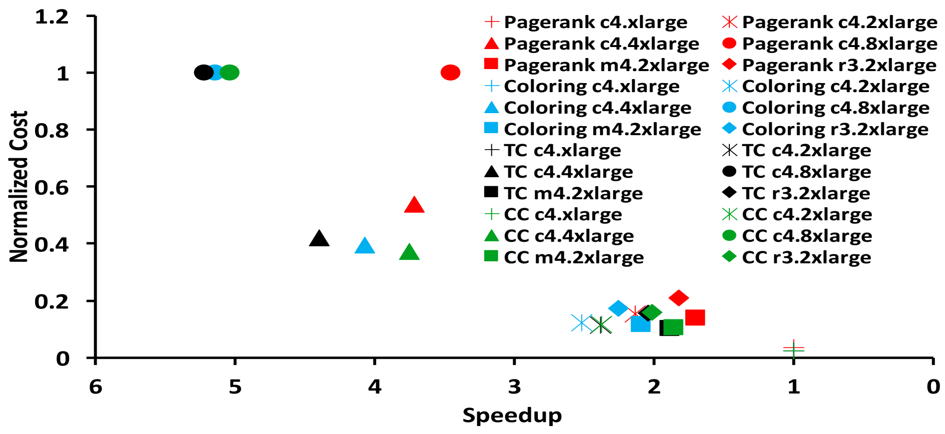


Figure 4.9: Cost and performance pareto graph of different computing nodes and different graph applications.

the thread-based *HAP*.

4.4.4 Cost Efficiency Projection

For users of cloud computing services, cost is a primary consideration. Other than the performance and energy improvements achieved in a heterogeneous cluster, profiling the synthetic graphs can also offer an accurate overview of the cost efficiency of different machines. Figure 4.9 plots the Pareto space of each individual machine's performance and cost on four applications. All cost and speedup information is generated by profiling synthetic graphs.

There are many metrics that can be used to evaluate cost efficiency, such as total cost of ownership (TCO) and cost per throughput/performance. Similarly, the cost per task is used to define a machine's efficiency. The cost per task is defined as the product of task runtimes and a machine's hourly rate (hourly cost is shown in [1]). As Figure 4.9 shows that machines of similar type are clustered. All

2xlarge machines (from three different domains) are grouped together with around 2x around speedup and 0.2x cost, which means none of them demonstrate their “advertised” specialty for graph applications. Within the computation-optimized domain, *8xlarge* is the most expensive machine for graph workloads. This is a result of the high charge rate and relatively low performance. The *4xlarge* and *2xlarge* saves 60% and 80% cost and provides 4x and 2x speedup, which should be considered as reasonable candidates for graph applications to satisfy both aspects. Without profiling using synthetic graphs, users would have no insights about the machines provided by cloud services or the machines they may have already deployed.

4.5 Summary

Graph analytics workloads have emerged as an extremely important class of problem during the age of big data. As the amount of heterogeneity in data centers continue to increase due to virtualization and the explicit introduction of heterogeneous compute units, it becomes more and more critical for graph processing frameworks to evolve as well. Towards this end, this chapter provides a number of graph partitioning strategies that attempt to account for heterogeneity in the data ingress phase of the popular PowerGraph framework. Moreover, this chapter shows that the skew factor derived by thread-based and profiling-based estimations can drive huge performance wins using heterogeneity-aware partitioning strategies. Experiments illustrate that *HAP* reduces application runtime by as much as 64% with an average of 32% on a small 4-node heterogeneous cluster. Additionally, evaluation shows that profiling-based *HAP* can offer an average of 15% speedup and 14.1% energy reduction over the thread-based one across a variety of applications and data sets with an amortizable profiling overhead.

Chapter 5

Hula: Auto-balancing Distributed Graph Processing On the Fly

The magnitude of data is growing exponentially in the big data era [21, 100]. A significant amount of this data is stored as graphs in many domains, such as social networks [67, 73], online retail, and bio-informatics [19]. To efficiently process the large-scale graphs, both academic and industrial communities have invested significant efforts on designing and optimizing graph processing systems.

Existing graph processing engines employ distributed-memory models to accelerate the graph computation with massive parallelism available in multi-machine clusters. However, distributed graph processing suffers performance inefficiencies due to workload imbalance [32, 61, 69, 93, 100]. This is mainly caused by the irregularity of graph analytics. For instance, irregular execution patterns of graph algorithms result in different amounts of dynamic operations (e.g., memory accesses, atomic operations, arithmetic operations, network transmissions, and many others) on each machine in the cluster. More severely, for non-stationary graph applications [61] like single source shortest path (SSSP), the workload intensity can dynamically move among machines. These significantly increase the difficulty of achieving work balance in distributed graph processing.

To balance the workloads, existing graph processing systems rely on various graph partitioning schemes [32, 45, 69, 76, 100, 128, 132] which decompose the workloads at the pre-processing stage. However, it is difficult to design a common static partitioning algorithm that avoids load imbalance for any given graph and application due to the unpredictable dynamic behaviors. To alleviate this issue, prior works [61, 93] adopt dynamic load balancing. During the execution, these approaches identify the source of imbalance at the vertex level and migrate the "heavy" vertices across machines. Moreover, attributes associated with these vertices, such as vertex states, properties, and adjacency lists are also migrated. After migration, a cluster-wide vertex mapping needs to be updated for the computational correctness. Although these schemes are promising, the large overhead of balancing workloads and remapping the vertices can easily surpass the performance gains [17].

This chapter presents *Hula*, a distributed graph processing system that auto-balances graph analytics on the fly with low overhead. It is observed that only a few machines (graph partitions) incur intensive workloads, and these machines are often surrounded by the ones with light workloads under the chunk-based partitioning scheme. Based on this key observation, *Hula* splits a graph into multiple chunks and contiguously distributes them in the multi-machine cluster (referred to as coarse-grained partitioning). This transforms a graph into a "hula hoop" shape, which naturally exposes a neighboring relationship for machines/partitions and preserves locality existing in real-world graphs [103, 132].

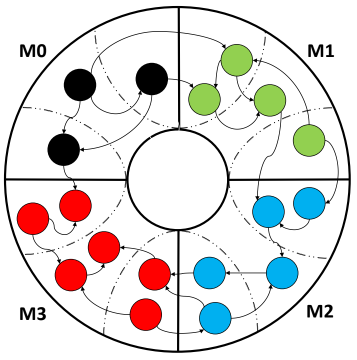
In the "hula hoop", *Hula* allocates a subset of vertices to be shared by neighboring machines. *Hula* further splits this subset in a finer-granularity and generates structural metadata (i.e., connection information) for fine-grained partitions that are stored on both neighboring machines. This forms *Hula*'s hybrid partitioner, which aims to help achieve flexible migrations and reduce the data movement during the migrations between neighbors. The migrating workload in *Hula* mimics the action of rolling a “hula hoop”.

Hula utilizes the lightweight hardware timer to monitor the workload across machines. Once imbalance is identified, *Hula*'s decentralized scheduler re-partitions the vertices and migrates minimum vertex properties (e.g., distance in SSSP). Overall, unlike the prior works, *Hula* 1) reduces the online monitoring overhead for workload imbalance, 2) lowers the data migration volume, and 3) simplifies the scheduling, migration, and synchronization logic. The contributions of *Hula* are summarized as follows:

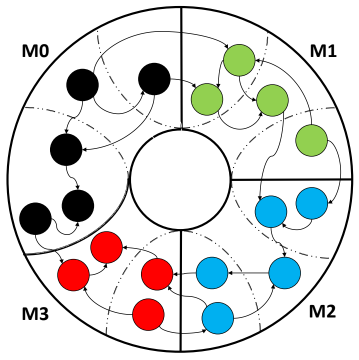
1. *Hula* proposes a novel hybrid partitioning scheme to maintain a graph's natural locality as well as generating metadata for lightweight online workload migrations.

2. *Hula* utilizes the hardware timer to monitor balance status and support workload scheduler to arrange work migration. The proposed migration scheme incurs minimal data movement to reduce workload imbalance in distributed graph engines.

3. *Hula*, a distributed graph processing system, is integrated with the proposed dynamic workload balancing strategy with various state-of-the-art techniques,



(a) Partitions before workload balancing.



(b) Partitions after workload balancing.

Figure 5.1: Overview of *Hula*'s workload balancing methodology. Note that M0 and M3 are used as examples to show the workload migration process.

such as a hybrid computation model and an intra-machine work-stealing scheme. With the support of *Hula*'s APIs, the load balancing process is transparent to users.

4. *Hula* is evaluated with extensive experiments and compared with two state-of-the-art distributed graph processing systems. Experiments of four popular graph algorithms with four real-world graphs running on a 16-machine cluster (close to a thousand computing cores) show that *Hula* outperforms existing systems, yielding speedups up to 48.7× (4.4× on average).

5.1 Design of Hula

This section first overviews *Hula*'s load balancing methodology, and then discusses its key components such as its hybrid partitioner, workload monitor, and load balancing scheme in detail.

5.1.1 Methodology Overview

Like other distributed graph processing frameworks, *Hula* needs to pre-process a given graph, and then apply the graph application on it. In the pre-processing stage, *Hula* partitions the graph and duplicates some vertices as well as their connection information (e.g., adjacency list, referred to as metadata) across neighboring partitions. As shown in Figure 5.1a, multiple vertices and their metadata are shared between a pair of neighboring partitions (e.g., the black and red vertices and associated edges surrounded by dot lines between M0 and M3).

During the execution phase, *Hula* can migrate work between neighboring machines for load balance. For instance, as shown in Figure 5.1b, if M3 has more work than M0, the shared components are migrated from M3 to M0. Since M0 already owns the metadata of the migrated components, only a small amount of information (e.g., up-to-date vertex properties) needs to be transferred via the network. After the migration, a partitioning boundary between M3 and M0 is re-drawn to update the ownership of the vertices. As can be seen, *Hula* restricts the migration network/topology and utilizes the metadata generated in the pre-processing phase to achieve lightweight load balancing at runtime.

5.1.2 Hybrid Partitioner

Graph systems typically utilize graph partitioners to reduce communication overhead and balance workloads. However, as aforementioned, static partitioning strategies alone cannot eliminate imbalance caused by applications' unpredictable activities and dynamic behaviors. Thus, *Hula* employs a hybrid partitioning scheme

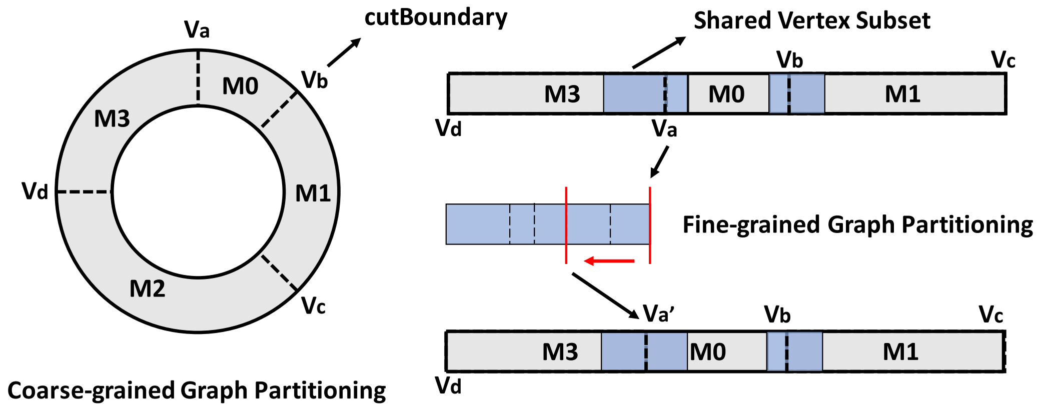

cutBoundary
Shared Vertex Subset
Va
Vb
Vc
Vd
M0
M1
M2
M3
Va'
Fine-grained Graph Partitioning
Coarse-grained Graph Partitioning

Figure 5.2: Example of *Hula*'s hybrid graph partitioning scheme for a 4-machine cluster.

that partitions a graph and prepares the metadata for online workload balancing.

In a cluster with p machines, *Hula* splits the graph to multiple partitions in an edge-centric manner and contiguously distributes it to each machine. As shown in Figure 5.2, this scheme naturally exposes a neighboring relationship among machines (referred to as **Coarse-grained Graph Partitioning**). After splitting, each machine owns one partition. Each partition is defined by two **cutBoundaries** (e.g., V_a and V_b). For example, a vertex with index ranging in between V_a and V_b is owned by $M0$. The next chunk ranging from V_b and V_c is owned by the neighbor $M1$. To roughly balance the workload, all partitions have an equal number of edges. According to the prior studies [25, 40, 108], graphs often have locality in adjacent vertices that are likely to be stored close to each other, as most edges connect two vertices with close vertex indices. *Hula*'s coarse-grained graph partition scheme can yield good data locality because vertices with close indices are likely connected with edges.

However, similar to prior partitioning schemes [32, 45, 107, 132], the coarse-grained partitioner is insufficient to balance a graph application's dynamic workload. Thus, *Hula* implements a second level partitioning to support dynamic load balancing with minimal cost. As shown in Figure 5.2, *Hula* selects a subset of vertices that is shared by two neighboring partitions in the "Hula hoop". This is marked as a **Shared Vertex Subset** (e.g., the blue shade between $M3$ and $M0$), which consists of $\alpha\%$ of edges originally owned by each partition (Section 5.3.3 discusses how to select an appropriate size of shared vertex subset). Then, each shared vertex subset is further partitioned at a finer granularity (referred as **Fine-grained Graph Partitioning**). Similar to the coarse-grained partitioner, each fine-grained partition has a similar amount of edges, and edges associated with a vertex reside in one partition. The number of mini-chunks produced by fine-grained partition is equal to the number of physical cores on a machine. This utilizes all computing units to parallelize the pre-processing work. Like the red lines shown in Figure 5.2, moving the "Hula hoop" transfers the ownership of the fine-grained chunks, which can help migrate the workload of a machine on the fly.

After the hybrid cutting, *Hula* generates the desired formats: compressed sparse row (CSR) and compressed sparse column (CSC) for the `push`/`pull` computation. This connection information is generated for both coarse-grained and fine-grained partitions. Pre-computing these for fine-grained chunks trades off the pre-preocessing time for light-weight data migration during execution. Moreover, *Hula* restricts the amount of data that can be transferred and only allows a migration to happen between a pair of neighboring machines (like the action of play-

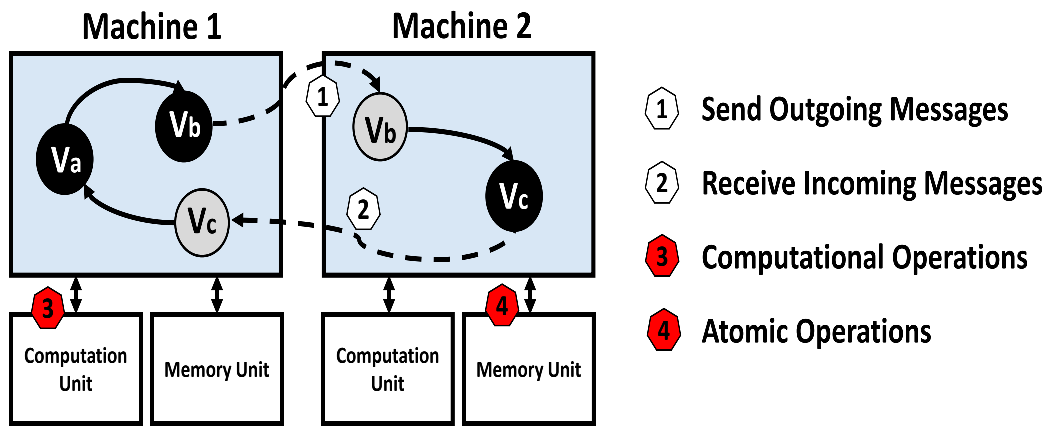


Figure 5.3: *Hula*'s workload monitoring system.

ing a "Hula hoop"). Compared to the fully connected migration topology in prior works [61, 93], *Hula*'s design simplifies the migration topology, minimizes the data movement in migration, and avoids possible network flooding during workload balancing.

5.1.3 Workload Monitoring

To balance distributed graph processing, *Hula* needs to monitor machines' workload intensity in each execution iteration (a.k.a., superstep [76]). As shown in Figure 5.3, *Hula* has four types of operations. However, unlike prior works [61, 93], *Hula*'s imbalance is mainly caused by the uneven distribution of computational operations and atomic operations. This is because 1) *Hula* dedicates one thread to send outgoing messages, one thread to receive incoming messages, and the rest of the threads to work on computation and memory access, and 2) Ethernet technology has been improved (e.g., Infiniband network) and will be further advanced. This

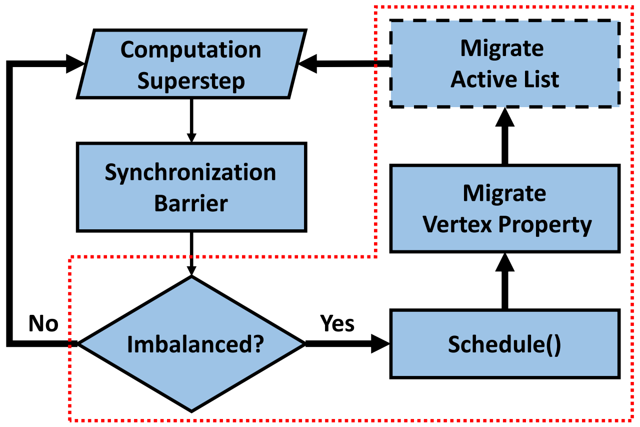


Figure 5.4: *Hula*'s workflow with the workload scheduling and migration support.

hides the impact of sending and receiving data in graph processing. *Hula* utilizes the hardware timer (i.e., $rdtsc()$) to profile the CPU cycles spent on the computation unit and the memory unit. Compared to a software timer (e.g., $gettimeofday()$) used in prior work [61], this method provides an extremely low overhead, where each profiling only costs tens of nanoseconds on modern chips [12]. In order to measure the "true" workload situation, the timers are placed before every lock or barrier to squeeze out the idle/waiting time across machines. This information is summarized as a score of balance (later referred as *bScore*). The usage of *bScore* is discussed in the next section.

5.1.4 Dynamic Load Balancing

Leveraging the information recorded by the monitoring system, *Hula* can decide the balance situation of a distributed cluster at the moment and take corresponding actions. Figure 5.4 overviews the workflow of *Hula* to achieve the dynamic load balancing; the core balancing steps are marked in the red-dotted box.

Within *Hula*, all machines perform computation supersteps concurrently until reaching a synchronization barrier. After the barrier, the neighboring machines exchange the balancing information. Workload migration is based on the satisfaction of two conditions. One is the number of active edges, which is equal to the number of active vertices $\times$ associated edges. If the number of active edges is smaller than 10% of the total number of edges, *Hula* does not proceed to the migration procedure. This is because the migration cost can surpass the benefit of balancing a small amount of workload. The other condition is comparing the balancing information between neighboring machines. If a machine owns more workloads than its neighbor without a significant difference (e.g., $>5\%$ used in experiments), no migration occurs. As shown in Figure 5.4, *Hula* continues to the next round of computation without the need of load balancing (i.e., the "No" path) in these two cases.

Otherwise, *Hula* goes to the load balancing path. *Hula* utilizes a lightweight work migration strategy, which consists of workload scheduling and migration. The design is further discussed in the following sections.

5.1.4.1 Workload Scheduling

When load imbalance between two neighboring machines is detected, they proceed to the scheduling phase as shown in Figure 5.4. *Hula* moves the partition boundaries between a machine and its left neighbor only to migrate workloads, which reduces synchronization overhead and parallelizes the scheduling process.

Algorithm 6 Workload Scheduler.

```
1: $leftMachine$ = ($machineId$ - 1 + $machines$) % $machines$
2:
3: //if this machine has a heavier workload than its left neighbor
4: $leftOverlap_r$ = 2 $\times$ $machineId$ + 1
5: if $bScore[machineId] > 2 \times bScore[leftMachine]$
6:     $newCutBoundary[machineId]$ = $overlapCutBoundary[leftOverlap_r]$
7: else
8:     $space$ = $overlapCutBoundary[leftOverlap_r]$ - $CutBoundary[machineId]$
9:     $percent$ = ($bScore[machineId]$ - $bScore[leftMachine]$) / $bScore[machineId]$ / 2
10:    $move$ = $percent \times space$ / $leftStep[machineId] \times leftStep[machineId]$
11:    $newCutBoundary[machineId]$ = $cutBoundary[machineId]$ + $move$
12:
13: //if left neighboring machine has a heavier workload than this machine
14: $leftOverlap_l$ = 2 $\times$ $machineId$
15: if $bScore[leftMachine] > 2 \times bScore[machineId]$
16:    $newCutBoundary[machineId]$ = $overlapCutBoundary[leftOverlap_l]$
17: else
18:    $space$ = $cutBoundary[machineId]$ - $overlapCutBoundary[leftOverlap_l]$
19:    $percent$ = ($bScore[leftMachine]$ - $bScore[machineId]$) / $bScore[leftMachine]$ / 2
20:    $move$ = $percent \times space$ / $leftStep[machineId] \times leftStep[machineId]$
21:    $newCutBoundary[machineId]$ = $cutBoundary[machineId]$ - $move$
22:
23: $Synchronize(newCutBoundary)$
24: $newCutBoundary[machines]$ = $vertices$ + $newCutBoundary[0]$
```

Algorithm 6 describes how to leverage *bScore* (i.e., balance score) to determine the new cut/partition boundary for the next computation iteration.

An example is used to illustrate the algorithm with two machines, which can be generalized to multiple machines in the cluster. Given a machine $M1$, if its left neighbor $M0$'s *bScore* is smaller than 50% of $M1$'s, then *newCutBoundary* between the two machines will be shifted to the rightmost side (*overlapCutBoundary* in line 6). It gives the entire shared vertex subset (shown in Figure 5.2) to $M0$. Otherwise, the scheduler first calculates the distance between the current partition and the rightmost boundary (line 8). Based on the difference between the two machines' *bScore*s, the new boundary then moves to the right side proportionally (line 9-11). Shifting the partition boundary to the right side helps reduce the workload for $M1$, as $M1$ owns fewer vertices/edges. Migrating work from the left machine (line 13-

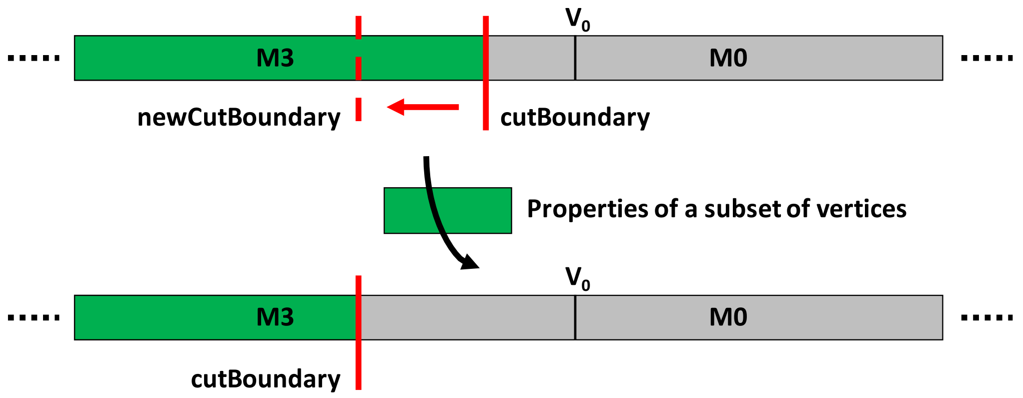


Figure 5.5: Examples of *Hula*'s workload migration between M3 and M0.

21) follows the similar strategy. The boundary movement mimics a "Hula hoop" shifting action. After computing the new left boundary, all machines synchronize this information and use it to migrate the necessary data before the next-round computation (line 23-25).

5.1.4.2 Workload Migration

After the scheduler calculates the new partition boundaries, the vertex property migration is initiated. *Hula* migrates vertex properties (e.g., a vertex's distance and activeness in SSSP algorithm) and updates the partition boundaries for the next round of computation.

To initiate a workload migration, *Hula* needs three pieces of information including starting pointer/address and size of migrating data, and direction of this migration. All of these are determined by the positions of current and new partition boundaries. For instance, Figure 5.5 shows the examples of workload migration between machine $M0$ and its left neighbor $M3$, where the *newCutBoundary* is on the left side of *cutBoundary*. This indicates the direction of a migration, where M3 needs to transfer work to M0. After confirming the direction, a subset of data in a vertex property array (starting from index *newCutBoundary*) is transmitted from M3 to M0. The size of this transmission is equal to the value difference between *newCutBoundary* and *cutBoundary*.

Although Figure 5.5 shows the example of $M0$ and its left neighbor, migration between a machine and its right neighbor follows the same logic. Other than the vertex property, applications like SSSP need to migrate active list before the next superstep (as shown in Figure 5.4 in block-dotted box). Migrating the active list and vertex property follow a similar logic. Differently, the active list is more condensed data format, as it only uses a single bit to represent a vertex's activeness. Compared to the data movements in prior workload balancing schemes [61, 93], *Hula* migrates a minimal amount of data to reduce workload imbalance on the fly.

5.2 Implementation Details of Hula

This section elaborates on *Hula*'s important system implementation details including the computation model, the programming model, and the intra-machine work stealing scheme.

5.2.1 Graph Computation Model

During the graph processing, the number of active edges in each iteration may vary dramatically. As studied in [22], the traditional top-down method (i.e.,

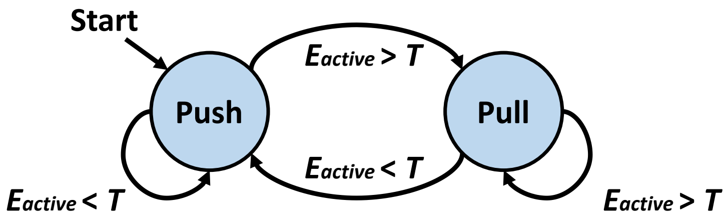


Figure 5.6: *Hula*'s computation model (E_{active} represents the number of active edges and T stands for the threshold used to switch between two models.)

push) cannot efficiently handle the graph computation all the time. Thus, modern graph algorithms and systems [22, 99, 103, 132] prefer a hybrid graph computation model that can be adapted according to the number of active edges. *Hula* is inspired to implement such a hybrid model, which combines the top-down computation manner with the bottom-up one.

As shown in Figure 5.6, there are two key parameters that determine the computation mode at the beginning of each iteration. E_{active} represents the number of active edges at the moment, and T is the threshold for mode switching. If E_{active} is less than T, the graph vertices in this iteration are executed in a *push* way, where an active vertex v_{src} and its value are broadcast to the destination machines in the cluster. Upon receiving this, all v_{dst}s in destination machines are traversed and updated. If *pull* mode (a.k.a, bottom-up) is selected, a v_{dst}'s source vertices are traversed locally, and an intermediate result is generated and sent to the machine owing v_{dst}. This can dramatically reduce the network traffic.

Algorithm 7 PageRank algorithm on *Hula*.

```
1:  $currRank = alloc\_vertex\_property$¡double¿()
2:  $nextRank = alloc\_vertex\_property$¡double¿()
3:  //active list records the activeness of each vertex
4:  $active = alloc\_active\_list()$
5:  $active \rightarrow active\_all()$
6:  $activeListMigration = false$
7:  // two user-defined utility function
8:  $commit(v, val)$
9:      $atomic\_add(\&nextRank[v], val)$
10: $accumulate(val_1, val_2)$
11:     return $val_1 + val_2$
12: // iterative superstep computation
13: $hula \rightarrow superstep$ (
14:     $push(v_{src}, outgoingEdges)\{$
15:         for $v_{dst} \in outgoingEdges$
16:             $commit(v_{dst}, currRank[v_{src}])$
17:     $\}$, //spread out $v_{src}$ value via outgoing edges
18:     $pull(v_{dst}, incomingEdges)\{$
19:         double $sum = 0$
20:         for $v_{src} \in incomingEdges$
21:             $sum$ += $currRank[v_{src}]$
22:         return $sum$
23:     $\}$, //locally summarize $v_{src}$ values for a $v_{dst}$
24:     $active$, $activeListMigration$ //$false$ indicates no need to migrate active list
25: )
```

5.2.2 Programming with Hula

Hula's APIs are illustrated with a programming example—PageRank. The PageRank algorithm [83] 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. Algorithm 7 demonstrates the PageRank implemented atop *Hula*. Lines 1-4 leverage *Hula*'s allocation APIs to allocate the vertex property arrays (*currRank* and *nextRank*) and active list (*active*). After these, users need to define two functions in lines 8-11. The $commit()$ function helps perform the user-defined atomic operation on a vertex. The $accumulate()$ function applies a user-defined operation to accumulate two intermediate results. With this support, PageRank can execute *Hula*'s superstep API function to compute vertices' rankings.

In a computation superstep (lines 13-25), users need to provide their own `push` and `pull` functions to traverse edges in either top-down or bottom-up fashion [22]. Upon receiving v_{src}, the `push` function adds its current value to each v_{dst} via a set of outgoing edges (lines 14-16). The add operation here leverages the user-defined *commit* to achieve atomicity. Differently, for a given v_{dst}, *pull* summarizes v_{dst}'s local sum via its *incomingEdges* and returns this local result back to *Hula* (lines 18-22). The local result is sent to the destination machine owning v_{dst}. Besides the `push` and `pull` functions, the active list and a Boolean value need to be passed to *Hula* as well. For PageRank, as all vertices during computation are active, *activeListMigration* is set to false to avoid migrating active list. For other applications such as SSSP, *activeListMigration* needs to be turned on. Other than these APIs, *Hula*, like other program systems [32, 45, 48, 103, 132], also provides an *update_vertices*() API for an application to apply a certain operation on all vertices (e.g., PageRank's ranking equation), which facilitates the algorithm implementation.

This PageRank example shows that the implementation of graph applications on *Hula* is straightforward. Additionally, *Hula*'s dynamic load balancing philosophy is transparent to users and does not incur heavy programming burdens.

5.2.3 Intra-machine Balance

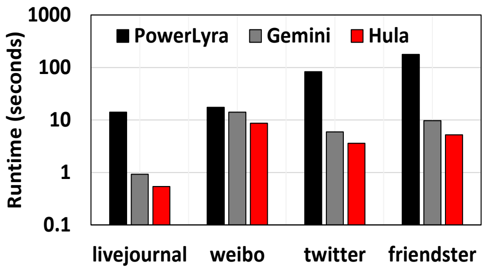
Other than the imbalance on the inter-machine level, the load imbalance within a machine can also severely impact the overall performance of distributed graph processing. To address this, *Hula* follows the idea proposed in prior approaches [24, 35, 41, 103, 132] to implement a fine-grained work stealing mechanism. *Hula* splits each partition into multiple chunks (i.e., the number of chunks equals the number of physical cores). To achieve work stealing, each chunk is further cut into multiple mini-chunks. As recommended in [103], each mini-chunk owns 256 contiguous vertices with associated metadata. Such contiguous vertex chunks with good access patterns are friendly to the existing hardware prefetcher.

In each execution superstep, all working threads attempt to finish up the assigned mini-chunks in order. Once a thread completes its job, it steals the remaining chunks from other "busy" threads. To avoid multiple threads working on the same mini-chunk, *Hula* uses a pointer to keep track of the progress. An atomic operation (e.g., $\_sync\_fetch\_and\_add$) is used to update the progress pointer that is visible to all threads. With this design, all threads on a machine are "eager" to finish the local execution to boost the single-machine performance.

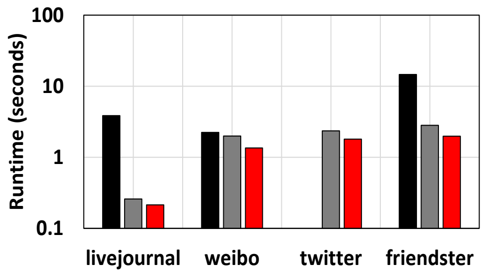
5.3 Evaluation

Hula is evaluated on a 16-machine cluster with Intel Xeon Platinum 8160 processors (Skylake) and InfiniBand network switches (up to 100Gb/s). Each machine has 48 physical cores with 32KB pivate L1 I/D caches, 1MB private L2 caches, and shared 33MB L3 caches. Each machine has 192GB DDR4 DRAM.

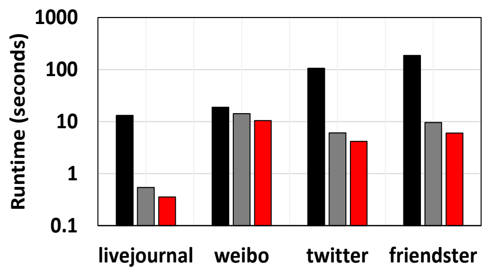
Hula is compared with two popular distributed graph processing systems—PowerLyra [32] and Gemini [132]. Some other prior load balancing work for distributed graph processing, such as Mizan [61] and GPS [93], cannot be setup due to the unsupported version of Hadoop (v1.x) and software bugs [32]. Power-



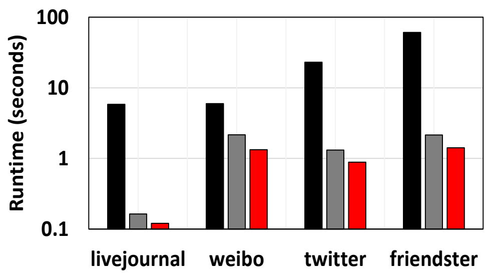
(a) PageRank



(b) SSSP



(c) TunkRank



(d) ConnComp

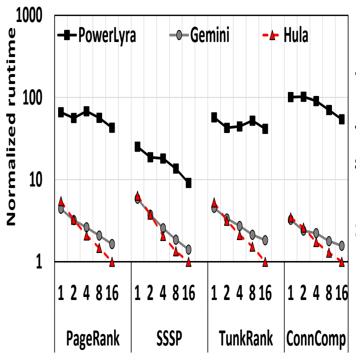
Figure 5.7: 16-machine runtime comparison of PageRank, SSSP, TunkRank, and ConnComp implemented atop PowerLyra, Gemini and Hula. Note that runtime is in log-scale and PowerLyra (uses Ginger partitioner to offer the best performance [32]) failed to process twitter graph [51, 96, 128, 132].

Lyra is chosen as the baseline system, as its hybrid-cut graph partitioning scheme has already shown a significant performance improvement in [32] over these prior works. Gemini is a computation-centric distributed graph processing system that employs chunk partitioning, hybrid computation model, and many other advanced techniques. Since Gemini and *Hula* have many similar system components, Gemini is selected as the baseline to show the performance improvement brought by *Hula*'s dynamic workload balancing design. The experiments include four popular graph applications from both stationary and non-stationary categories (non-stationary: Single Source Shortest Path (SSSP), Connected Components(ConnComp); stationary: PageRank and TunkRank) [61]. The average results of three repeated executions (standard deviation $< 5\%$) are reported. The input for these applications include four real-world graphs (i.e., livejournal-twitter) and three large synthetic RMAT graphs [63] (i.e., RMAT1-RMAT3) with the number of vertices and edges ranging from 4.8 to 500 million and from 69 million to 10 billion, respectively, as shown in Table 3.2.

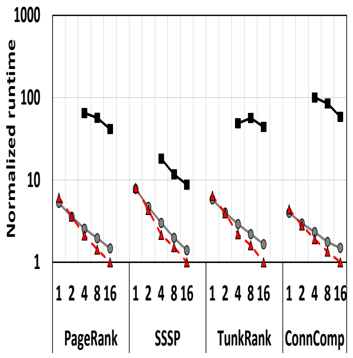
Experiment Outline Section 5.3.1 compares *Hula*'s performance with Gemini and PowerLyra on a 16-machine cluster (with close to a thousand computing units). This experiment uses the real-world graphs as the input to show the performance in practical usage. Section 5.3.2 uses large synthetic graphs to evaluate *Hula*'s inter-machine scalability. Due to the size limit of real-world graphs, the synthetic (RMAT) graphs are used as the input to make sure each machine and each core has enough computation under a large scale. Section 5.3.3 further discusses *Hula*'s performance on the various parameter settings and demonstrates the key design trade-offs.

5.3.1 Overall Performance

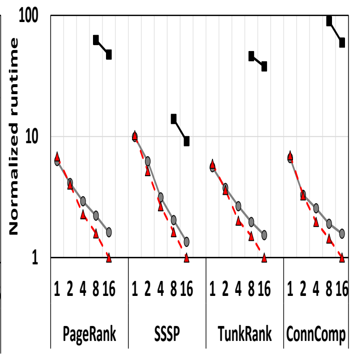
As *Hula* aims to improve the distributed graph processing performance via a lightweight load balancing technique, comparing it to other state-of-the-art distributed graph processing systems can help quantify *Hula*'s high performance improvement. Figure 5.7 reports the 16-machine performance of PowerLyra, Gemini and *Hula*, executing four popular graph applications on four real-world graphs.



(a) RMAT1



(b) RMAT2



(c) RMAT3

Figure 5.8: Inter-machine scalability of the application execution of PowerLyra [32], Gemini [132] and *Hula* (1-16 machines) on three synthetic RMAT graphs. Note: missing points are due to the failure of exceeding memory capacity.

Runtime is reported in logarithmic scale for readability.

PowerLyra is a general distributed graph platform that provides many options for researchers to test their ideas. Even though PowerLyra only deploys static partitioners (e.g., Ginger), it still outperforms many state-of-the-art systems [61, 93] with a dynamic load balancer [32]. As shown in Figure 5.7, *Hula* outperforms PowerLyra in all cases significantly (13.7× on average), with up to 48.7× for the ConnComp application on the livejournal graph (in Figure 5.7d). For the large-scale graphs such as twitter and friendster, *Hula* can yield an average of 24.5× speedup across four applications. This explicitly indicates *Hula*'s superior performance in comparison to prior works on dynamic re-balancing for distributed graph processing.

Unlike PowerLyra, Gemini is a dedicated computation-centric system that utilizes most of the state-of-the-art optimization techniques. As reported in [132], it outperforms PowerGraph, GraphX, and PowerLyra in the distributed setup, and provides a competitive (sometimes even better) performance as Galois [80] and Ligra [99] on a single machine. For PageRank and TunkRank, Figure 5.7a and 5.7c show that *Hula* achieves 1.6× speedup on average, while Figure 5.7b and 5.7d indicate that *Hula* provides an average of 1.4× speedup on SSSP and ConnComp, respectively. The speedup difference is mainly due to SSSP's and ConnComp's non-stationary runtime behaviors, which incur more workload migrations. Since Gemini's chunk partitioner is similar to *Hula*'s coarse-grained partitioner, the performance improvement is purely brought by *Hula*'s lightweight dynamic load balancing design.

5.3.2 Scalability

To demonstrate *Hula*'s inter-machine scalability, three large-scale synthetic graphs generated by PaRMAT [63] are used to provide enough work for the 16-machine cluster. Due to inefficient memory usage [51, 96, 128, 132], PowerLyra cannot handle the RMAT2 graph with fewer than 4 machines, and it cannot handle the RMAT3 graph with fewer than 8 machines. However, Gemini and *Hula* can operate these RMAT graphs in all cases.

Figure 5.8 demonstrates the inter-machine scalability of the execution phase of the three systems. For all RMAT graphs, *Hula* can achieve a significant speedup (up to 102.2× on ConnComp executing the RMAT1 graph) over PowerLyra in the scale-out experiments. Among all applications, *Hula* shows the smallest speedup over PowerLyra on SSSP (an average of 13.8×). This is because PowerLyra provides relatively better scalability on SSSP compared to the other three graph workloads.

Unlike PowerLyra, Gemini has competitive performance and scalability compared to *Hula*. However, as the number of machines increases, Gemini suffers from scaling losses because of the workload imbalance and the increasing communication overhead. More machines in a cluster generally provide more computational power. Yet, it enlarges the difficulty of balancing the graph workload and introduces more network costs. This surpasses the benefits obtained from adding more computation resources. Such behavior is clearly observed in Figure 5.8a and 5.8b. With a worse single-node performance (e.g., SSSP executing RMAT3 in Figure 5.8c), *Hula* can increasingly outperform Gemini on four applications for scaling out. On average, compared to the 1-machine setup, *Hula* running on a 16-machine cluster yields an average of 6.3$\times$ speedup across the four applications with the three RMAT graphs. Among all the configurations (1 to 16 machines), *Hula* outperforms Gemini by up to a 1.8$\times$ (1.3$\times$ on average) speedup.

5.3.3 Further Discussions

Some micro metrics and key design parameters are analyzed to further evaluate *Hula*.

Shared Vertex Subset In order to achieve lightweight workload migrations, *Hula* prepares a **Shared Vertex Subset** for each pair of neighboring partitions in the "Hula Hoop". The size of this subset has a large impact on *Hula*'s performance,

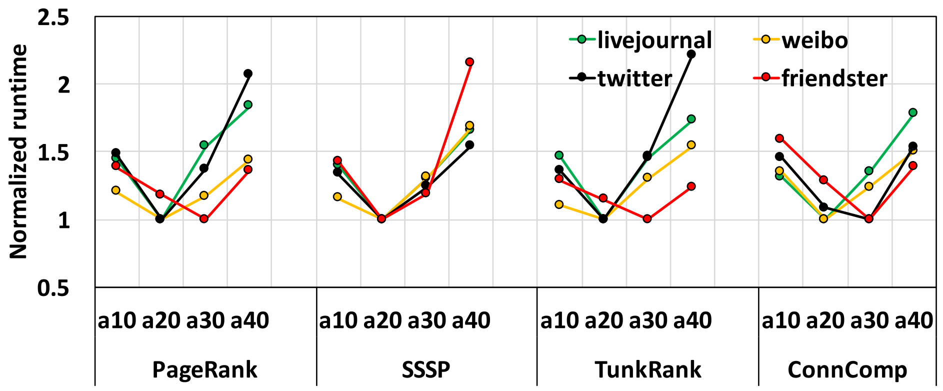


Figure 5.9: Sensitivity study on the size of **Shared Vertex Subset**. The size of **Shared Vertex Subset** is represented by α and α10 denotes α =10%.

as it explicitly determines the granularity and total amount of data of workload migration. Thus, a sensitivity study is performed on the size of the shared subset on the 16-machine cluster. In this study, α represents the size of a shared subset, which is the percentage of edges in a partition that belongs to this shared subset. *Hula* scales α from 10% to 40%, which means 20% to 80% of edges in a partition can be transferred to nearby workers (each partition contributes to two shared vertex subsets with its left and right neighbors). As Figure 5.9 shows, α10 always provides a sub-optimal performance due to its limited amount of data that can be migrated. At α20 or α30, *Hula* reaches the best performance on all applications. Since the number of partitions of various αs are the same (i.e., equal to the number of physical cores on a machine), increasing the size of the shared vertex subset (i.e., α) enlarges the size of each partition. If an individual partition is too large to resolve a minor workload imbalance (like α40), it leads to continuous occurrences of data migrations in remaining computations. This is the main cause of the

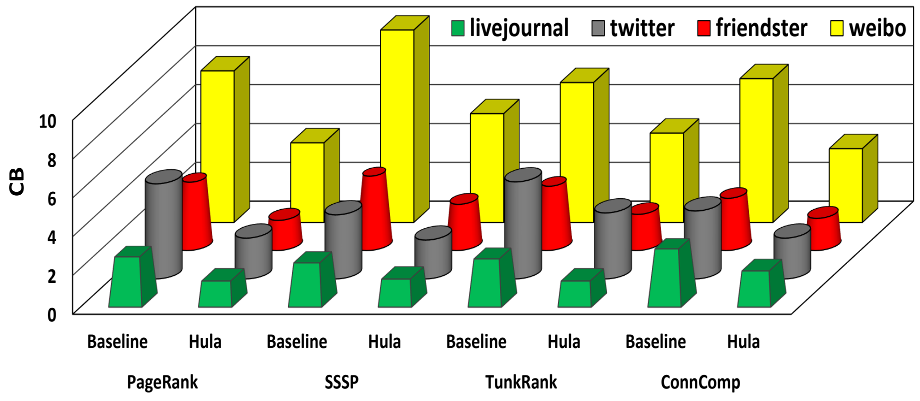


Figure 5.10: The comparison between baseline (*Hula* with dynamic workload balancing disabled) and *Hula* in terms of CB (balance of a cluster) metric on 16-machine cluster.

performance inflections in Figure 5.9. Increasing the number of partitions results in a finer-grained partitioning, which can solve this issue on a large shared subset. However, it can significantly aggravate the pre-processing cost.

Overall, $\alpha 20$ outperforms other configurations in most cases, as it provides enough data partitioned at the selected granularity (i.e., number of partitions equals number of cores) to optimize workload imbalance. *Hula* uses $\alpha 20$ as the default setting in the experiments.

Workload Balance To quantify workload balance of processing a graph application on a p-machine cluster, *Hula* defines a metric *CB*, which stands for the cluster balance. *CB* is the average time difference between the “slowest” and “fastest” machines per iteration. Formally, it is defined as follows:

$$CB = \frac{1}{n}\sum_{1}^{n} \frac{\max(bScore_i)}{\min(bScore_i)} \forall i, \tag{5.1}$$

In this equation, i stands for a machine in a cluster and n denotes the number of iterations to finish an application's execution. Figure 5.10 shows the CB comparison between baseline (*Hula* with dynamic workload balancing disabled) and *Hula*. For PageRank and TunkRank applications, *Hula* can reduce up to 57% (46% on average) of work imbalance on four real graphs. Workload migration starts between neighbors at the beginning of these two stationary applications until it reaches the boundary limits of the shared vertex subset. In contrast, workload migration keeps happening during the execution of non-stationary applications such as SSSP and ConnComp due to the dynamic workload changes across iterations. This certainly increases the difficulty of balancing the workload. Thus, *Hula* achieves a smaller imbalance reduction on these applications. On average, *Hula*'s proposed balancing scheme is able to reduce the workload imbalance by 43% on four popular graph applications.

Hybrid Partitioning To achieve lightweight data migration on the fly, *Hula* leverages the fine-grained partitioning to further split the shared vertex subset and generate the corresponding metadata. This could increase the pre-processing cost. *Hula*'s pre-processing time is compared with six state-of-the-art pre-processing designs to quantify the overhead of *Hula*'s hybrid partitioning scheme. As shown in Table 5.1, *Hula*'s pre-processing cost is much lower than the average cost of all seven pre-processing procedures, as *Hula* outperforms Random-Ginger in most cases. Even

Table 5.1: Pre-processing cost (seconds) comparison of partitioning four real-world graphs on 16-machine cluster. Random, Grid, Oblivious are implemented atop PowerGraph [45]. Hybrid and Ginger are implemented on top of PowerLyra [32]. Chunk is deployed by Gemini [132]. Compared to most of state-of-the-art pre-processing schemes, *Hula*-Hybrid has a lower cost.

Partitioning Algorithm	livejournal	weibo	twitter	friendster
Random	5.5	32.6	100.1	93.6
Grid	3.6	25.1	65.1	69.7
Oblivious	9.4	61.8	226.1	244.1
Hybrid	3.9	13.1	85.1	94.9
Ginger	8.2	57.1	186.6	223.8
Chunk	2.5	52.6	52.9	80.1
Hula-Hybrid	2.6	53.4	57.1	88.6
GMEAN	4.6	37.6	95.1	113.3

Table 5.2: Memory footprint (GB) comparison of PowerLyra, Gemini, and *Hula* on a 16-machine cluster.

Systems	livejournal	weibo	twitter	friendster
PowerLyra	21.0	91.5	351.3	380.5
Gemini	8.2	23.1	67.4	73.5
Hula	9.1	25.1	83.6	87.9

though *Hula*'s hybrid partitioning algorithm is more complicated as compared to Chunk, it only incurs an extra 2.5% overhead on average. This is because *Hula* parallelizes all the pre-processing work on the available cores. Such negligible pre-processing overhead can be easily amortized with the repeated usage of metadata by different applications on the same graph [129].

Memory Footprint After *Hula*'s pre-processing stage, the graph's structural information (e.g., CSR and CSC data structures) is stored, and then utilized in the execution of graph applications. Other than the data stored for a static partition, *Hula*'s hybrid partitioner generates the structural metadata for the shared vertex subsets. This could incur a larger memory footprint. This section profiles the memory footprint of PowerLyra, Gemini and *Hula* on a 16-machine cluster, and compares them in Table 5.2. The results shown in Table 5.2 are the average memory footprint of four applications, where SSSP implemented atop all systems consumes approximately 10%-20% higher memory usage compared to other applications. This is due to the storage of edge weights. Compared to PowerLyra, *Hula* reduces the memory footprint by an average of 70.1% across four real-world graphs. Compared to Gemini, *Hula*'s footprint is 14.9% larger on average. It is clear to see that the metadata generated by *Hula*'s partitioner trades off a small amount of memory footprint to the dynamic lightweight workload migration. Since modern data center machines are trending towards a higher memory capacity to accommodate in-memory systems (e.g., Spark [10] and RocksDB [13]), such a small footprint increment is tolerable.

5.4 Summary

This chapter presents *Hula*, a distributed graph processing system that auto-balances graph computation on the fly. With the support of *Hula*'s hybrid partitioner, *Hula* combines the hardware timer and a decentralized scheduler to achieve lightweight workload balancing, which results in better performance and scalability in distributed graph processing. Experimental results on a 16-machine cluster show that *Hula* outperforms state-of-the-art distributed graph processing systems, yielding speedups up to 48.7$\times$.

Chapter 6

SLFE: A Distributed Graph Processing System with Redundancy Reduction [4]

To achieve high performance, existing graph systems exploit massive parallelism using either distributed [31, 45, 46, 71, 76, 80, 88, 93, 119, 132] or shared memory models [68, 81, 89, 99]. Such systems process graphs in a repeated-relaxing manner (e.g., using Bellman-Ford algorithm variants [23] to iteratively process a vertex with its active neighbors) rather than in a sequential but work-optimal order [18, 74, 75, 132]. This introduces a fundamental trade-off between available parallelism and redundant computations [74, 75]. Several popular graph processing systems [32, 45, 132] have been analyzed. This dissertation observed that redundant computations pervasively exist.

As discussed in Section 2.3, due to the nature (i.e., the core aggregation function) of different graph algorithms, the root causes of computational redundancies in graph analytics vary across applications. For example, applications such as Single Source Shortest Path (SSSP) employ *min*() as their core aggregation func-

[4]S. Song, X. Liu, Q. Wu, A. Gerstlauer, T. Li, and L. K. John, "Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction", in *Proceedings of VLDB Endowment*, 2019. Shuang Song contributed to the idea development and implementation. Qinzhe Wu was involved in performance evaluation. Xu Liu, Andreas Gerstlauer, Tao Li, and Lizy K. John supervised the overall project.

tion. 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, the redundancies in these applications are definied as the computations triggered by the updates with intermediate (not final min/max) values. This dissertation proposes 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, the 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. A "finish early" approach is proposed to terminate the subsequent computations on these vertices to eliminate such redundancies.

SLFE (pronounced as "Selfie"), a distributed graph processing system, reduces redundancies to achieve high-performance graph analytics. In contrast to the prior works that leverage dynamic re-sharding/partitioning [113] or multi-round partitioning [66] to reduce redundancy in out-of-core graph systems, *SLFE* has the following benefits:

- *SLFE* does not incur any extra partitioning effort [5].
- It does not rely on any specific ingress methodology, so it can be easily adopted by other systems.
- It has extremely low pre-processing overhead, which is suitable for online optimization.
- *SLFE* 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. *SLFE* deploys a set of redundancy-reduction aware `push`/`pull` functions to make use of guidance produced in pre-processing. Moreover, *SLFE* also provides a set of system APIs to enable redundancy reductions as well as programming simplicity/flexibility for different graph applications.

SLFE is the first distributed graph processing system that is designed with the redundancy reduction principal. Compared to state-of-the-art graph procesisng systems (including three state-of-the-art distributed and two shared-memory graph processing systems), *SLFE* yields speedups up to 74.8$\times$ (16.3$\times$ on average) and

[5]The partitioning phase in distributed graph systems is expensive [72, 105, 110, 132].

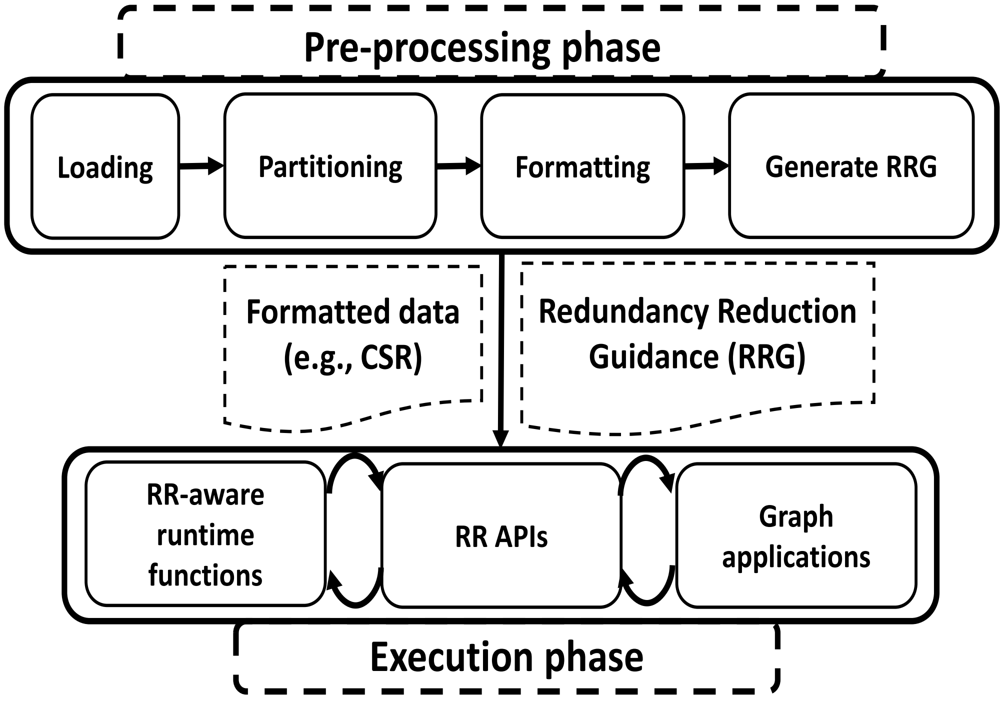


Figure 6.1: System overview of *SLFE*.

1644× (56× on average) over existing distributed and shared memory systems, respectively.

6.1 System Design of SLFE—Start Late or Finish Early

This section first overviews *SLFE*'s approach and then discusses its key system designs in detail.

6.1.1 SLFE Methodology and System 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 [45, 71, 76, 132] typically consist of two phases: pre-processing and execution; *SLFE* follows the same design principle. As Figure 6.1 shows, *SLFE* loads, partitions, formats the entire graph in the pre-processing phase, and then generates the RRG for redundancy optimization. The subsequent execution phase accepts the pre-processing outputs (i.e., formatted graph and RRG). Section 6.1.2 and 6.1.3 elaborate on *SELF*'s partitioner and RRG generation, respectively.

SLFE's execution phase consists of three components as shown in Figure 6.1: 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.

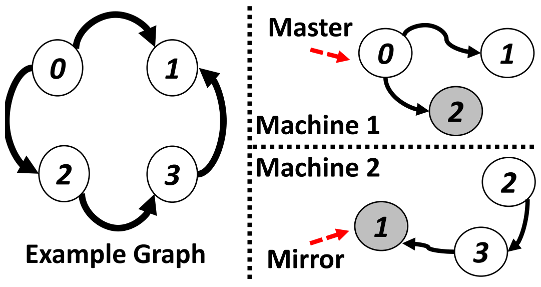
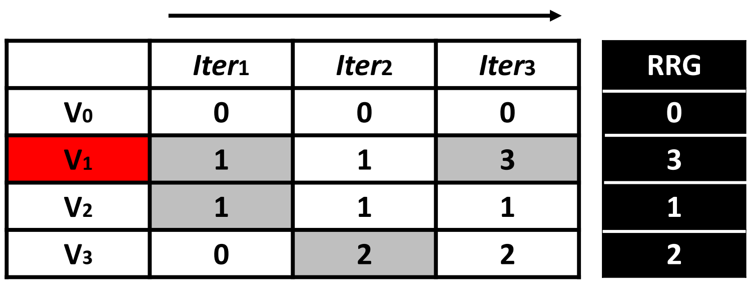


Figure 6.2: Example of the chunking partitioning.

Meanwhile, these functions 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 6.1.4-6.1.6 discuss each component of *SLFE*'s execution phase.

6.1.2 Chunking Partitioner

At the beginning of *SLFE*'s pre-processing 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 [25, 108] with a minimal partitioning cost. An efficient chunking partitioning scheme [132] is employed to evenly partition a large-scale graph into contiguous chunks and assign a chunk to each machine in a cluster. Figure 6.2 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 [32, 45, 76], some vertices are duplicated across different



	*Iter*1	*Iter*2	*Iter*3	RRG
V0	0	0	0	0
V1	1	1	3	3
V2	1	1	1	1
V3	0	2	2	2

Figure 6.3: RRG for the example graph shown in Figure 6.2.

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 [20, 32, 45, 46, 68, 71, 99, 106, 132]. Thus, it dramatically enhances the applicability of *SLFE*’s techniques to other graph systems. The next step of the pre-processing phase is to generate RRG.

6.1.3 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 [28]) to record the iteration number of a vertex’s update. Figure 6.3 shows the RRG generation with an exam-

Algorithm 8 Pre-processing 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 $v_{dst} \in V$
 7:         for $v_{src} \in v_{dst}.incomingNeighbors$
 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;
```

ple graph. 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 6.3 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 8 shows the pseudo-code of the proposed pre-processing 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 only the topological knowledge of the graph needs to be obtained. Moreover, *visited* is used 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 pre-processing overhead. Once v_{dst}'s *dist* is updated, it becomes active to propagate its value to the succeeding vertices.

After Algorithm 8, 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 instance, the V_1's RRG in Figure 6.3 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").

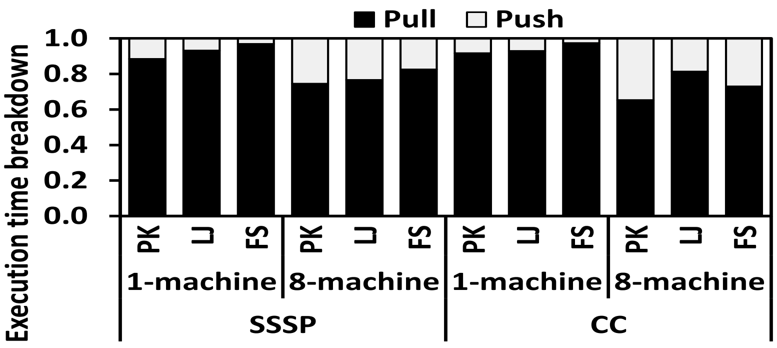


Figure 6.4: SSSP and CC execution time breakdown of `pull` and `push` mode, which are measured in 1-machine and 8-machine setup with pokec (PK), liveJournal (LJ), and friendster (FS) graphs.

Overall, the proposed pre-processing 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 [18, 32, 45, 46, 68, 76, 119, 132] can easily adopt it. The overhead of this scheme is low and is thoroughly evaluated in Section 6.2. The next section discusses how *SLFE* applies RRG in the execution phase.

6.1.4 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 [22, 81, 99, 132] 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, the `push`/`pull` propagation model is analyzed.

The execution time of `pull`/`push` mode in SSSP and CC [6] is measured 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 6.4 shows, SSSP and CC on a single machine spend more than 92.8% and 94% of their execution time in the `pull` mode. When run 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 [59]. Eliminating redundancies in `push` is also attempted. 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 [7]. 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

[6]Applications with arithmetic aggregation functions are excluded, as they always execute in the `pull` mode to iteratively compute all vertices [7].

Algorithm 9 Pull Mode Computation.

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

Algorithm 10 Push Mode Computation.

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

incoming edges, and then apply a user defined *pullFunc* on the destination vertex. Algorithm 9 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 (the usage of this variable is explained 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 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 10), since `push` does not employ redundancy optimization, but only guarantees the result correctness. The details of Algorithm 10 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 [76] is commonly deployed by modern distributed systems [32, 45, 71, 80, 119, 132] 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, all the vertices in the transition phase (i.e., `pull`$\rightarrow$`push`) are reactivated. 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.

Table 6.1: RR APIs provided by *SLFE*.

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

6.1.5 RR-APIs

SLFE defines a set of application programming interfaces (APIs) to transparently optimize redundancy, as shown in Table 6.1. 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.

6.1.6 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

Algorithm 11 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 $v_{dst} \in v_{src}.outgoingNeighbors$
 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 $v_{src} \in v_{dst}.incomingNeighbors$
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
```

SLFE's RR APIs, optimizing redundant computation requires minimum programming efforts.

6.1.6.1 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 11 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

Algorithm 12 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 $V$s
11: uint32_t* $stableCnt$ = new uint32_t[$numV$]; //RulerS
12: float* $stableValue$ = new float[$numV$];
13: vertexUpdate(vOp)
14:     for $v_x \in V$
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; $stableValue[v_x]$ = $rank[v_x]$;
```

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) [43]. 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, *SLFE*'s SSSP does not incur any extra effort from the programming perspective.

6.1.6.2 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, only the difference is demonstrated here. Algorithm 12 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 12 (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*, it is considered 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.

6.1.7 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, the idea of [24, 35, 41, 132] is utilized 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 [132].

6.1.8 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 $\{d_{n_1}^{(t)}, \dots, d_{n_k}^{(t)}\}$ 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 $min(d_{n_1}^{(t)} + e_{n_1 v}, \dots, d_{n_k}^{(t)} + e_{n_k v})$, 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 $t \to \infty$. Thus, by the monotone convergence theorem [52], 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 $\{f_v(\mathcal{V}^{(0)}), \dots, f_v(\mathcal{V}^{(t)})\}$ produced by the function f_v converges as $t \to \infty$ 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 $t \to \infty$. 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. *SLFE*'s accuracy was experimentally verified by comparing *SLFE*'s results (e.g., vertices' properties) with the ones produced by other systems [32, 45, 132].

6.1.9 SLFE's Generality

The idea of redundancy reduction in *SLFE* is applicable to other graph frameworks. The reason a new framework *SLFE* is proposed for the 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 the RR optimization.

The Gather-Apply-Scatter (GAS) computation model [45] has been widely adopted by many popular graph systems [45, 46, 82, 106, 110, 131]. The RRG provided by *SLFE*'s unique pre-processing 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 [45] as an example, one can adapt *SLFE*'s 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.

6.2 Evaluation

SLFE is evaluated on a 8-machine cluster with 2nd generation of Xeon-Phi processor (detailed machine configuration is shown in Table 3.1) and InfiniBand network switch (up to 100Gb/s).

This sections compares *SLFE* with three distributed graph processing systems—PowerGraph [45], PowerLyra [32], and Gemini [132]. In addition, this section compares *SLFE*'s performance in a single machine with two shared-memory systems, GraphChi [68] and Ligra [99]. All the experiments include five popular graph applications from the two categories according to Table 2.1 (*min/max*: Single Source Shortest Path (SSSP), Connected Components(CC), WidestPath (WP); *arithmetic*: PageRank (PR) and TunkRank(TR)). The average results of five repeated executions (standard deviation $< 5\%$) are reported. The input for these applications include seven real-world graphs (i.e., pokec-friendster[7]) and three large synthetic RMAT graphs (i.e., RMAT1-RMAT3) 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 3.2.

[7]In this chapter, abbreviations PK, LJ, WK, DI, OK, ST, and FS are used for pokec, livejournal, wiki, delicious, orkut, s-twitter and friendster graphs, respectively.

Table 6.2: 8-machine end-to-end runtime and improvement over the state-of-the-art distributed systems.

Pre-processing 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×						

Experiment Outline Section 6.2.1 compares *SLFE*'s end-to-end performance (pre-processing and execution time) with other three distributed systems running with the largest scale: 8 machines and 68 cores per machine. This experiment use the real-world graphs as the input to show the performance in practical usage. Section 6.2.2.1 evaluates *SLFE*'s intra-machine scalability (scale-up), with the comparison with the two shared-memory systems. As limited to the memory size, only the real-world graphs are used as the input. Section 6.2.2.2 evaluates *SLFE*'s inter-machine scalability (scale-out). The RMAT graphs are used as the input to make sure each machine and each core has enough computation under large scale. Section 6.2.3 shows some micro metrics (e.g., number of computations and network traffic) to verify that the performance gains are due to the optimization of redundant computation.

6.2.1 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 6.2 reports the 8-machine performance of PowerGraph, PowerLyra and *SLFE*, running five popular applications on seven real graphs. The first part of Table 6.2 reports the pre-processing time of the three systems, which includes the graph loading, partitioning, and formatting steps. Since *SLFE* extends the general pre-processing phase to generate RRG information, such time cost is reported as well. For applications' execution time, the results show that *SLFE* outperforms these two

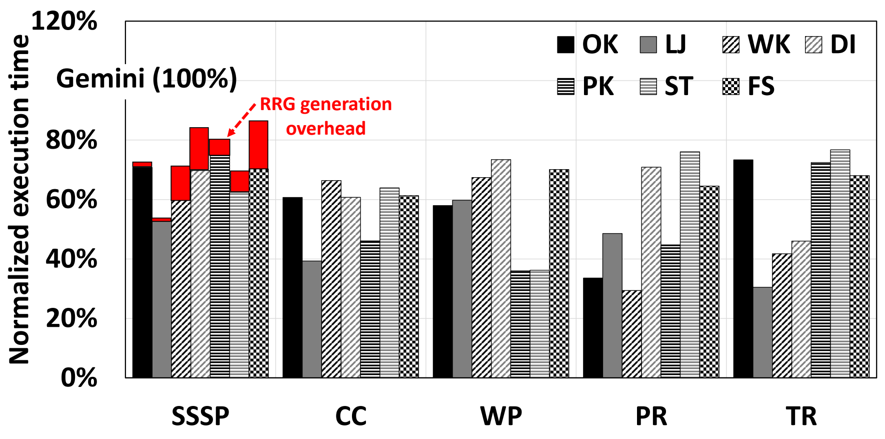


Figure 6.5: *SLFE*'s time of execution phase and RRG generation time normalized to Gemini [132] on a 8-machine cluster.

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.

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 [132], Gemini is reported to outperform PowerGraph, GraphX [46], and PowerLyra by 19.1× on average. *SLFE*'s performance is compared with Gemini in Figure 6.5. Since *SLFE* and Gemini utilize the same pre-processing method (pre-processing time is already reported in Table 6.2), 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 pre-processing 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 6.5, 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.

6.2.2 Scalability Evaluation

6.2.2.1 Intra-machine Scalability

Next, the intra-machine scalability of *SLFE* is evaluated by using 1 to 68 cores to run five applications with all real graphs that fit in a single machine's memory. Overall, Figure 6.6 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

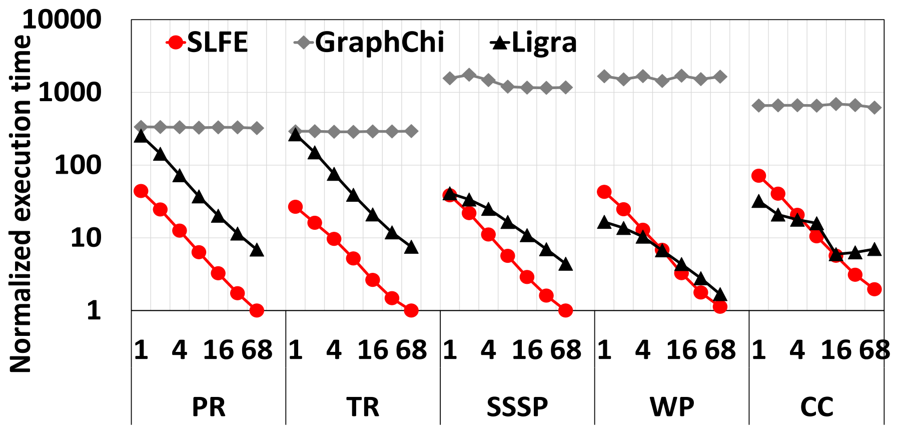


Figure 6.6: Intra-machine scalability (1-68 cores) of *SLFE*, GraphChi [68], and Ligra [99] on a single-machine setup (average of seven real world graphs).

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, *SLFE*'s scalability can be compared 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 6.6, 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.

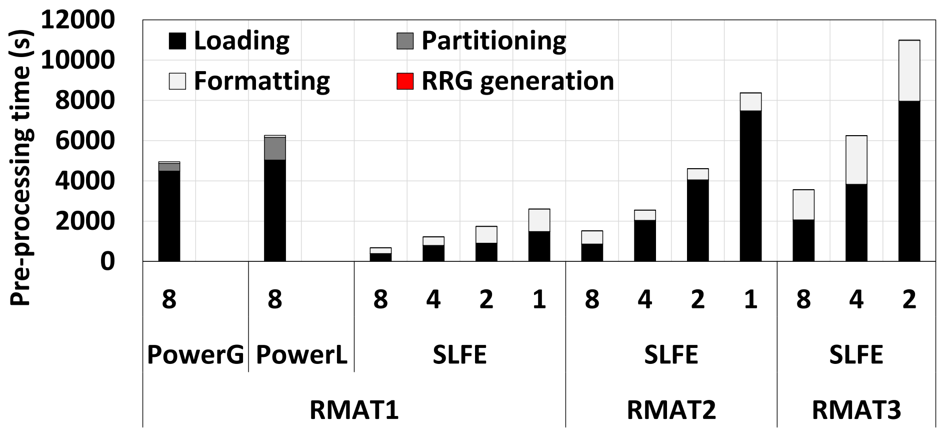
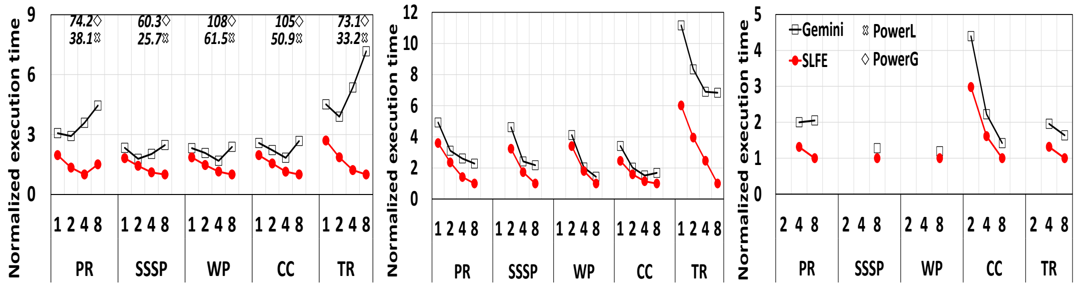


Figure 6.7: Inter-machine scalability of the pre-processing phase of PowerGraph [45], PowerLyra [32], and *SLFE* on three synthetic graphs. *SLFE* and Gemini [132] use the same pre-processing methodology except for the RRG generation. PowerGraph and PowerLyra can only execute the smallest RMAT1 graph in a 8-machine setup because of their inefficient memory usage [51, 96, 128, 132], while *SLFE* fails to process RMAT3 in a single machine. For all the cases, RRG generation is invisible because it incurs very small overhead.

6.2.2.2 Inter-machine Scalability

To demonstrate *SLFE*'s inter-machine scalability, PaRMAT [63] is used to generate three large-scale synthetic RMAT graphs. Due to the inefficient memory usage [51, 96, 128, 132], PowerGraph and PowerLyra can only handle the smallest RMAT1 graph in the 8-machine cluster. However, Gemini and *SLFE* can operate these RMAT graphs in most cases. Figure 6.7 summarizes these four systems' pre-processing time. For RMAT1 graph on a 8-machine setup, *SLFE* finishes the pre-processing procedure much faster than the PowerGraph and PowerLyra. Moreover, as shown in Figure 6.7, *SLFE*'s pre-processing phase also provides a good scale-out scalability for all the three RMAT graphs. Compared to the original preprocessing

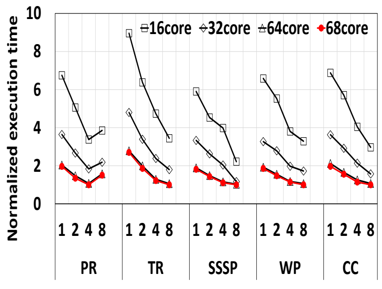


(a) RMAT 1 (b) RMAT 2 (c) RMAT 3

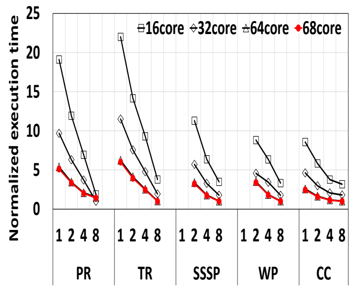
Figure 6.8: Inter-machine scalability of the execution phase of Gemini [132], PowerGraph [45], PowerLyra [32], and *SLFE* (1-8 machines) on three synthetic RMAT graphs. Note: missing points are due to the failure of exceeding memory capacity.

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

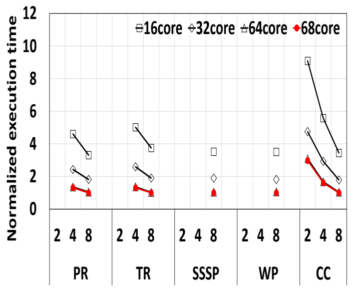
After comparing the pre-processing cost, the inter-machine scalability of the execution phase of the four systems is demonstrated in Figure 6.8. This also includes the sum of *SLFE*'s execution time and RRG generation time to further verify the feasibility of the proposed approach in various scale-out configurations. As shown in Figure 6.8a, *SLFE* can achieve a significant speedup (up to 108$\times$ and 51.5$\times$) over PowerGraph and PowerLyra for all the five applications with the RMAT1 graph on the 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



(a) RMAT 1



(b) RMAT 2



(c) RMAT 3

Figure 6.9: 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.

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, *SLFE*'s trend-lines are reported by varying the number of cores per server as well as the number of server machines in the cluster. Figure 6.9 shows such analysis on the three synthetic graphs. 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. 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, 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.

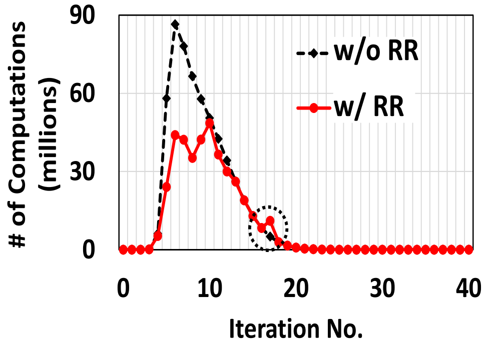
6.2.3 Further Discussions

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

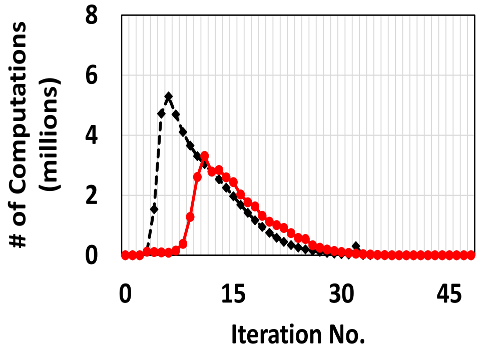
6.2.3.1 Number of Computations

The computation here is defined as an update on a vertex, which includes a *min*/*max* or arithmetic operation and the corresponding synchronization operations. The number of computations during the execution phase of SSSP, CC, and PR is shown in Figure 6.10. 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 6.10, 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.

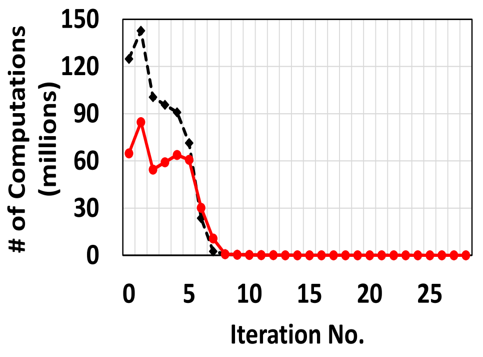
The SSSP initiates from a given root, and its number of computation dramatically increases when more vertices are involved in the computation (Figure 6.10a and 6.10b). Redundant computations are reduced in the `pull` mode. Hence, com-



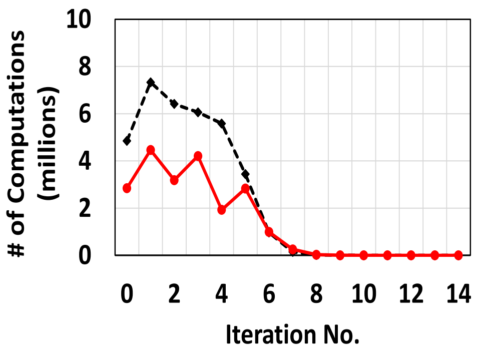
(a) SSSP-FS



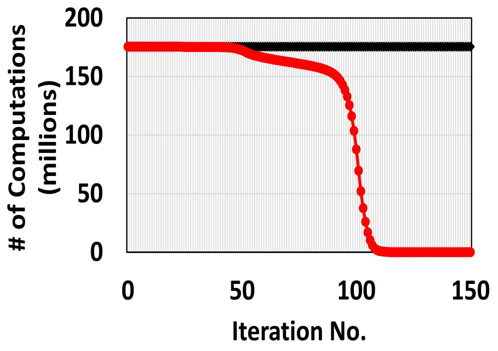
(b) SSSP-LJ



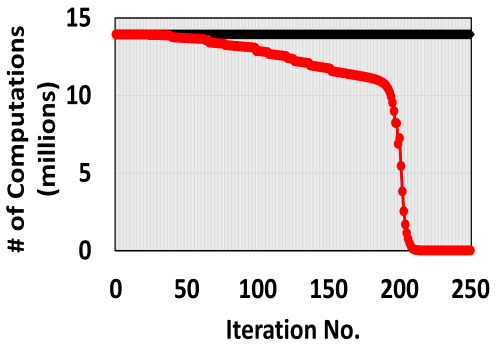
(c) CC-FS



(d) CC-LJ



(e) PageRank-FS



(f) PageRank-LJ

Figure 6.10: *SLFE*'s no. of computations per iteration.

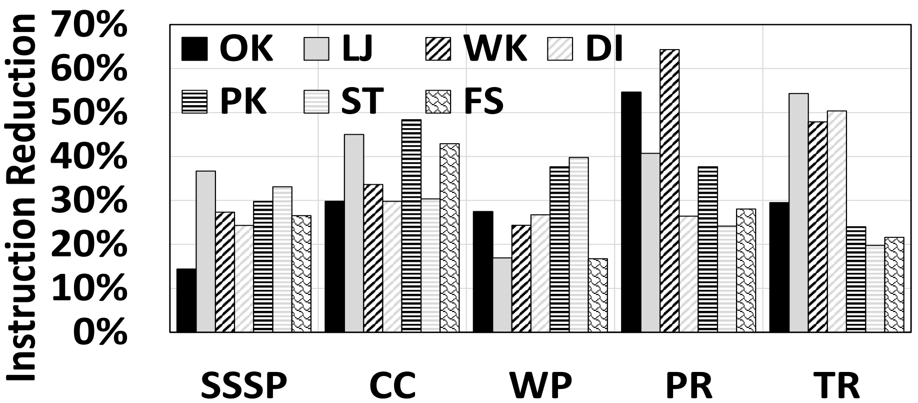


Figure 6.11: *SLFE*'s reductions on the number of instructions.

pared 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 6.1.4, the `push` function activates all vertices to deliver "unseen" updates of inactive vertices in the `pull`→`push` transition phase. One such event is circled in Figure 6.10a), which only incurs a small amount of immediate computations to guarantee the correctness. Figure 6.10c and 6.10d 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 [32, 45, 68, 132] 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 6.10e and 6.10f).

Table 6.3: *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%

6.2.3.2 Hardware and System Metrics

Instructions and Memory Acceses. The instruction reduction when *SLFE* enables redundancy reduction (RR) is quantified via counting the retired instructions using performance monitoring units [6, 11]. 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, the number of memory accesses is quantified via counting hardware load and store events. As shown in Table 6.3, *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 [69, 100, 101]. For simplicity, *SLFE* is compared 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

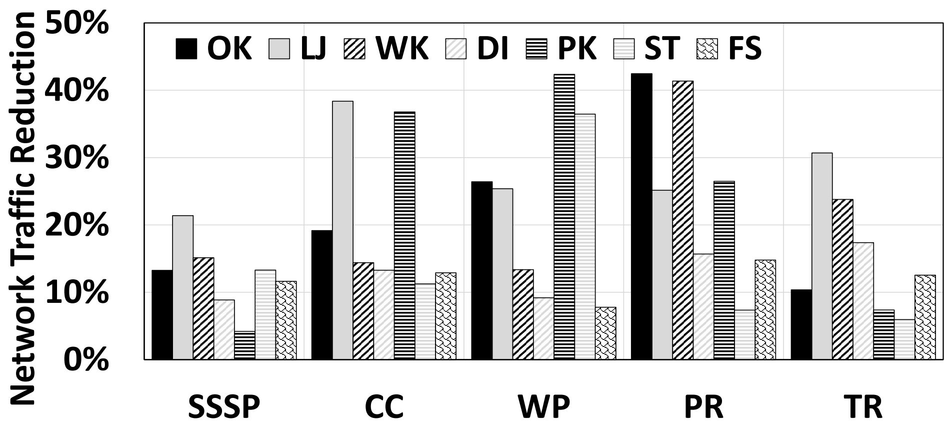


Figure 6.12: *SLFE*'s network traffic reduction on a 8-machine cluster.

the same amount of network traffic as Gemini. Figure 6.12 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.

6.3 Limitations

SLFE has two limitations. First, redundancy reduction guidance needs to be generated in the pre-processing 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. The pre-processing overhead could be further optimized in the future. Second, although not observed in the 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, work stealing addresses this issue. However, it is challenging to address the potential inter-machine load imbalance due to costly communication via network. Thus, a load balancing scheme can be developed to prevent this.

6.4 Summary

This chapter presents *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 pre-processing 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.

Chapter 7

Conclusion

A significant amount of data from various fields is stored as vertices and edges to explicitly expresses connections between data segments. To quickly obtain valuable insights, researchers and data scientist employ powerful computing clusters to apply graph algorithms atop these large-scale graphs. However, processing graphs in a distributed setup suffers performance degradation caused by workload imbalance and redundant computations. In order to improve the performance of distributed graph processing, this work aims to optimize the imbalance and redundancy via novel pre-processing techniques and customized computation units.

7.1 Summary

The dissertation provides three contributions to enhance the workload balance and computational efficiency of distributed graph analytics. First, Heterogeneity-aware Partitioning (*HAP*) [58, 69, 100] balances the data distribution of graph processing in heterogeneous environments. *HAP* provides two estimation methodologies to quantify the heterogeneity of a cluster. It also enhances five partitioning schemes to utilize the estimations of heterogeneity for data splitting. Balancing the data distribution reduces the workload imbalance and communication cost during execution, which speeds up distributed graph analytics significantly.

Another contribution is the *Hula*, which auto-balances the distributed graph workload with minimal overhead. *Hula* creates a hybrid partitioning scheme to maintain graph locality, as well as to generate metatdata for lightweight workload migration. The hardware timer is utilized by *Hula* to keep track of the work intensity of each machine in the cluster. Based on this information, *Hula*'s decentralized scheduler arranges the workload migrations between two iterations of computation. With the support of metadata, only a minimal amount of data is needed for work migration. These techniques provided by *Hula* achieve a lightweight balance optimization for distributed graph processing.

Finally, this dissertation defines the redundant computations existing in distributed graph processing, and reveals the root causes of the redundancy. Moreover, it provides *SLFE* [103]—a system solution to reduce redundancy for distributed graph analytics. *SLFE* develops a lightweight pre-processing technique to capture the maximum propagation level of vertices in a given graph. This topological information is defined as Redundancy Reduction Guidance (RRG) and is further utilized for redundancy reduction. *SLFE* designs Redundancy Reduction (RR)-aware runtime functions to prune redundant operations on the fly and support regular graph traversals. To maintain high programmability, *SLFE* implements a set of RR-aware APIs and provides them to users. These techniques provided in *SLFE* allow distributed graph processing to perform fewer redundant computations.

7.2 Future Work

There are a number of future research directions that can enhance and utilize the techniques described in this dissertation. This section explores a number of promising next-steps to further improve load balancing and computational efficiency for distributed graph processing.

To improve the load balance of distributed graph processing on heterogeneous clusters, heterogeneity-aware partitioning techniques need an accurate estimation of underlying machines' graph processing capability. Due to the rapid development of cloud computing, cutting edge hardware are merged in the service line in a fast manner. In this case, the accurate profiling-based estimation of machines' graph processing capability can be costly. The thread-based estimation method incurs no extra cost, but it is not accurate enough. Thus, an estimation methodology is needed to precisely reflect the computational power of machines in a heterogeneous cluster with minimal overhead. Graph applications are generally composed of operations like memory accesses, atomic locks, arithmetic operations, etc. Prior works [77, 84, 86, 90–92, 101, 102, 117, 118, 120, 130] show that thorough evaluations can help understand the correlations between emerging applications and hardware resources. Thus, detailed evaluations need to be performed in order to reveal the correlations between graph algorithms and fundamental hardware components. This information can be further utilized to estimate a machine's graph processing power in an accurate and lightweight manner. Building such estimation methodology is a promising direction to balance the data distribution of large-scale graphs in a heterogeneous cluster.

Another interesting future work is to apply the dynamic workload balancing scheme proposed in this dissertation on various graph partitioning algorithms. As observed in the experiments of this dissertation (i.e., Table 5.1), depending on the graph structures, various partitioning schemes result in different pre-processing efficiencies. Moreover, recent work [36] shows that there is no common partitioner favoring all graph applications. These factors motivate the further effort of adopting the lightweight dynamic balancing design principal for various graph partitioners. The first challenge here is to select an appropriate partitioner for a given graph and application. The second challenge is how to form the neighboring relationship between machines once a proper partitioner is selected. Some partitioners (like Grid partitioner [45]) may result in a complex relationship for work migration. Consequently, research on extending the migration scheme needs to be performed. As can be seen, there are many opportunities to explore further in this area.

Random walk has recently grained immense attention as a powerful mathematical tool for extracting information for many important graph algorithms (e.g., personalized PageRank [42, 50], SimRank [57], node2vec [47]) and machine learning tasks. However, due to the dynamic nature of sophisticated walk strategies, implementing such an important graph processing task atop existing graph processing engines [71, 76, 132] encounters significant performance and scalability problems. To solve these issues, the first distributed graph random walk engine, KnightKing, has been proposed [122] lately. To improve the computation efficiency of this important class of graph tasks, studies (similar to the ones performed in Section 2.3) can be performed to analyze the type of redundancy and the amount of redundant operations in applications implemented atop the model of random walking. Due to the nature of dynamic walk, the root cause of redundant operations needs to be explored. Redundancy reduction schemes described in this dissertation can be extended and exploited for the future design of the random walk engine. Overall, there remains a significant amount of work that can be pursued to improve the performance of distributed graph random walk systems.

Bibliography

[1] Amazon EC2. http://aws.amazon.com/ec2. Accessed: 04-16-2015.

[2] Intel xeon phi coprocessor. https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html. Accessed: 04-16-2015.

[3] 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.

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

[5] Parallel boost graph library. https://www.boost.org/doc/libs/1_72_0/libs/graph_parallel/doc/html/index.html, May 2009.

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

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

[8] Facebook. https://www.facebook.com/, October 2017.

[9] Apache hadoop. http://hadoop.apache.org/, February 2018.

[10] Apache spark. https://spark.apache.org/, February 2018.

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

[12] Nanosecond-precision test. http://wiki.zeromq.org/results:more-precise-0mq-tests, 2019.

[13] Rocksdb. https://rocksdb.org/, October 2019.

[14] Build and run applications without thinking about servers. https://aws.amazon.com/serverless/, 2020.

[15] A. Abdolrashidi and L. Ramaswamy. Incremental partitioning of large time-evolving graphs. In *2015 IEEE Conference on Collaboration and Internet Computing (CIC)*, pages 19–27, Oct 2015.

[16] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, April 2010.

[17] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *Proceedings of the Seventeenth International Conference on Architectural*

Support for Programming Languages and Operating Systems, ASPLOS XVII, pages 61–74, New York, NY, USA, 2012. ACM.

[18] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin 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.

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

[20] 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.

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

[22] Scott Beamer, Krste Asanović, and David 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.

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

[24] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[25] 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.

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

[27] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM.

[28] Alan Bundy and Lincoln Wallen. Breadth-first search. In *Catalogue of Artificial Intelligence Tools*, pages 13–13. Springer, 1984.

[29] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In

2014 IEEE 28th International Parallel and Distributed Processing Symposium, pages 889–901, May 2014.

[30] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *2009 46th ACM/IEEE Design Automation Conference*, pages 927–930, July 2009.

[31] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing 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.

[32] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo 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.

[33] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015.

[34] Byung-Gon Chun, Gianluca Iannaccone, Giuseppe Iannaccone, Randy Katz, Gunho Lee, and Luca Niccolini. An energy case for hybrid datacenters. *SIGOPS Oper. Syst. Rev.*, 44(1):76–80, March 2010.

[35] 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.

[36] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 752–768, New York, NY, USA, 2018. ACM.

[37] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[38] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 293–304, New York, NY, USA, 2017. ACM.

[39] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[40] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262, August 1999.

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

[42] Dniel Fogaras, Balzs Rcz, Kroly Csalogny, and Tams Sarls. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.

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

[44] Siddharth Garg, Shreyas Sundaram, and Hiren D. Patel. Robust heterogeneous data center design: A principled approach. *SIGMETRICS Perform. Eval. Rev.*, 39(3):28–30, December 2011.

[45] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos 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.

[46] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion 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.

[47] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 16, pages 855–864, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proc. VLDB Endow.*, 8(9):950–961, May 2015.

[49] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

[50] Taher H. Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th International Conference on World Wide Web*, WWW 02, pages 517–526, New York, NY, USA, 2002. Association for Computing Machinery.

[51] Benjamin Heintz and Abhishek Chandra. Enabling scalable social group analytics via hypergraph analysis systems. In *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud15, page 14, USA, 2015. USENIX Association.

[52] Francis B Hildebrand. *Methods of applied mathematics*. Courier Corporation, 2012.

[53] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle 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.

[54] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. Pgx.d: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 58:1–58:12, New York, NY, USA, 2015. ACM.

[55] Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

[56] Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. Graphbuilder: Scalable graph etl framework. GRADES, pages 4:1–4:6, 2013.

[57] Glen Jeh and Jennifer Widom. Simrank: A measure of structural-context similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 02, pages 538–543, New York, NY, USA, 2002. Association for Computing Machinery.

[58] Lizy Kurian John, Shuang Song, and Andreas Gerstlauer. Guided load balancing of graph processing workloads on heterogeneous clusters, October 8 2019. US Patent 10,437,648.

[59] 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.

[60] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[61] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos 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.

[62] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50, Oct 2015.

[63] Farzad Khorasani, Rajiv Gupta, and Laxmi 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.

[64] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, New York, NY, USA, 2014. ACM.

[65] Jérôme 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.

[66] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian 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.

[67] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[68] Aapo Kyrola, Guy Blelloch, and Carlos 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.

[69] 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.

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

[71] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[72] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.*, 8(3):281–292, November 2014.

[73] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[74] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav 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.

[75] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav 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.

[76] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz 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.

[77] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. Csalt: Context switch aware large tlb. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 17, pages 449–462, New York, NY, USA, 2017. Association for Computing Machinery.

[78] Kamran Najeebullah, Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. Bpp: Large graph storage for efficient disk based processing. *arXiv preprint arXiv:1401.2327*, 2014.

[79] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[80] Donald Nguyen, Andrew Lenharth, and Keshav 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.

[81] Donald Nguyen, Andrew Lenharth, and Keshav 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.

[82] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan 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.

[83] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[84] R. Panda, S. Song, J. Dean, and L. K. John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, Feb 2018.

[85] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 243–252, New York, NY, USA, 2015. ACM.

[86] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*,

ISCA 07, pages 412–423, New York, NY, USA, 2007. Association for Computing Machinery.

[87] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, and Seif Haridi. Distributed vertex-cut partitioning. In *Proceedings of the 14th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 8460*, pages 186–200, Berlin, Heidelberg, 2014. Springer-Verlag.

[88] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy 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.

[89] Amitabha Roy, Ivo Mihailovic, and Willy 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.

[90] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John. Gpgpu benchmark suites: How well do they sample the performance spectrum? In *2015 44th International Conference on Parallel Processing*, pages 320–329, Sep. 2015.

[91] J. H. Ryoo, S. Song, and L. K. John. Puzzle memory: Multifractional partitioned heterogeneous memory scheme. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 310–317, Oct 2018.

[92] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA 17, pages 469–480, New York, NY, USA, 2017. Association for Computing Machinery.

[93] Semih Salihoglu and Jennifer 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.

[94] Thomas Schank. *Algorithmic aspects of triangle-based network analysis*. PhD thesis, University Karlsruhe, 2007.

[95] Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. Using domain-specific languages for analytic graph databases. *Proc. VLDB Endow.*, 9(13):1257–1268, September 2016.

[96] Zechao Shang, Feifei Li, Jeffrey Xu Yu, Zhiwei Zhang, and Hong 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.

[97] Bin Shao, Haixun Wang, and Yatao 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.

[98] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 317–332, GA, 2016. USENIX Association.

[99] Julian Shun and Guy 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.

[100] 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.

[101] 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.

[102] Shuang Song, Raj Desikan, Mohamad Barakat, Sridhar Sundaram, Andreas Gerstlauer, and Lizy K. John. Fine-grain program snippets generator for mobile core design. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI 17, pages 245–250, New York, NY, USA, 2017. Association for Computing Machinery.

[103] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy Kurian John. Start late or finish early: A distributed graph processing system with redundancy reduction. *Proc. VLDB Endow.*, 12(2), August 2019.

[104] Isabelle Stanton. Streaming balanced graph partitioning algorithms for random graphs. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 1287–1301, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[105] Isabelle Stanton and Gabriel 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.

[106] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[107] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international Conference on Web Search and Data Mining*, pages 333–342. ACM, 2014.

[108] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The

anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[109] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. Adaptive partitioning for large-scale dynamic graphs. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 144–153, June 2014.

[110] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow.*, 10(5):493–504, January 2017.

[111] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim.*, 13(4):32:1–32:27, October 2016.

[112] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 237–251, New York, NY, USA, 2017. ACM.

[113] Keval Vora, Guoqing Xu, and Rajiv 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.

[114] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement—scalable and programmable

analytics over very large graphs on a single PC. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 387–401, Santa Clara, CA, 2015. USENIX Association.

[115] Lei Wang, Fan Yang, Liangji Zhuang, Huimin Cui, Fang Lv, and Xiaobing 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.

[116] Jianshu Weng, Ee-Peng Lim, Jing Jiang, and Qi He. Twitterrank: Finding topic-sensitive influential twitterers. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, pages 261–270, New York, NY, USA, 2010. ACM.

[117] Joseph Whitehouse, Qinzhe Wu, Shuang Song, Eugene John, Andreas Gerstlauer, and Lizy K. John. A study of core utilization and residency in heterogeneous smart phone architectures. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE 19, pages 67–78, New York, NY, USA, 2019. Association for Computing Machinery.

[118] Q. Wu, S. Flolid, S. Song, J. Deng, and L. K. John. Invited paper for the hot workloads special session hot regions in spec cpu2017. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 71–77, Sep. 2018.

[119] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo 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.

[120] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. Can we trust profiling results? understanding and fixing the inaccuracy in modern profilers. In *Proceedings of the ACM International Conference on Supercomputing*, ICS 19, pages 284–295, New York, NY, USA, 2019. Association for Computing Machinery.

[121] Da Yan, James Cheng, Yi Lu, and Wilfred 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.

[122] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. Knightking: A fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP 19, page 524537, New York, NY, USA, 2019. Association for Computing Machinery.

[123] Y. Yasui, K. Fujisawa, and K. Goto. Numa-optimized parallel breadth-first search on multicore single-node system. In *2013 IEEE International Conference on Big Data*, pages 394–402, Oct 2013.

[124] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[125] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA, 2015. ACM.

[126] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA, 2015. ACM.

[127] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA, 2015. ACM.

[128] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin 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.

[129] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. Cgraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 441–452, Boston, MA, 2018. USENIX Association.

[130] X. Zheng, H. Vikalo, S. Song, L. K. John, and A. Gerstlauer. Sampling-based binary-level cross-platform performance estimation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1709–1714, March 2017.

[131] Yang Zhou, Ling Liu, Kisung Lee, and Qi Zhang. Graphtwist: Fast iterative graph computation with two-tier optimizations. *Proc. VLDB Endow.*, 8(11):1262–1273, July 2015.

[132] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong 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.

[133] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, 2015. USENIX Association.