AN EXPERIMENTAL COMPARISON OF PARTITIONING STRATEGIES IN DISTRIBUTED GRAPH PROCESSING

BY

SHIV VERMA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Associate Professor Indranil Gupta

ABSTRACT

In this thesis, we study the problem of choosing among partitioning strategies in distributed graph processing systems. To this end, we evaluate and characterize both the performance and resource usage of different partitioning strategies under various popular distributed graph processing systems, applications, input graphs, and execution environments. Through our experiments, we found that no single partitioning strategy is the best fit for all situations, and that the choice of partitioning strategy has a significant effect on resource usage and application run-time. Our experiments demonstrate that the choice of partitioning strategy depends on (1) the degree distribution of input graph, (2) the type and duration of the application, and (3) the cluster size. Based on our results, we present rules of thumb to help users pick the best partitioning strategy for their particular use cases. We present results from each system, as well as from all partitioning strategies implemented in two common systems (PowerLyra and GraphX).

To friends, family and teachers.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

There is a vast amount of information around us that can be represented in the form of graphs. These include graphs of social networks, bipartite graphs between buyers and items, graphs of road networks, dependency graphs for software, etc. Moreover, the size of these graphs has rapidly risen and can now reach up to hundreds of billions of nodes and trillions of edges [1]. Systems such as PowerGraph [2], Pregel [3], GraphX [4], Giraph [5], and GraphChi [6] are some of the plethora of graph processing systems being used to process these large graphs today. These frameworks allow users to write vertex-programs which define the computation to be performed on the input graph. Common applications including PageRank or Single Source Shortest Path can be easily expressed as these vertex-programs.

To be able to compute on large graphs, these systems are typically run in a distributed manner. However, to distribute graph computation over multiple machines in a cluster, the input graph first needs to be *partitioned* before computation starts by assigning graph elements (either edges or vertices) to individual machines.

The partitions created have a significant impact on the performance and resource usage in the computation stage. To avoid excess communication between different partitions during computation, systems typically use *vertex mirroring*, whereby some vertices may have images in multiple partitions. If a partitioning strategy results in a large number of mirrors, then it will lead to higher communication costs, memory usage, and synchronization costs. These synchronization overheads and communication costs, in turn, lead to higher job completion times. Besides reducing the number of mirrors, the partitioning strategy needs to make sure that the partitions are balanced in order to avoid overloading individual servers and creating stragglers.

Graph partitioning itself must also be fast and efficient; for some graph applications, the time it takes to load and partition the graph can be much

Table 1.1: Systems and their Partitioning Strategies.

System	Partitioning Strategies
PowerGraph (§5)	Random, Grid, Oblivious, HDRF, PDS
PowerLyra (§6)	Random, Grid, Oblivious, Hybrid, Hybrid-Ginger, PDS
GraphX (§7)	Random, Canonical Random, 1D, 2D

larger than the time it takes to do the actual computation. In particular, the authors of [7] found that when they ran PageRank for 30 iterations with PowerGraph on 10 servers, around 80% of the time was spent in the ingress and partitioning stage. Our own experiments reveal similar observations.

The characteristics of the graph also play an important role in the determining the efficiency of a partitioning technique. For example, many real world graphs, such as social networks or web graphs [8], follow a power-law distribution. Gonzalez et. al. demonstrate in [2] that the presence of very high-degree vertices in power-law graphs present unique challenges from a partitioning perspective, and motivate the use of *vertex-cuts* in such cases. A large amount of research has been done to improve graph partitioning for distributed graph processing systems, e.g., [9, 2, 10]. Current research is typically aimed at reducing the number of mirrors and thus improving graph processing performance while still keeping the graph ingress phase fast.

Today, many of the aforementioned graph processing systems [9, 2, 4] offer their own set of partitioning strategies. For instance, as shown in Table 1.1, PowerGraph [2] offers five different partitioning strategies, GraphX [4] offers four, and PowerLyra [9] six. Even after a user has decided which system to use, it is rarely clear which partitioning strategy is the best fit for any given use case. In this thesis, we aim to address this dilemma. Our first goal is to compare partitioning strategies *within* each system. This holds value for developers planning to use a given system. It is *not* our goal to compare graph processing systems against each other. They release new versions frequently, and there is sufficient literature on this topic [11]. We also implement all partitioning strategies in one common system (PowerLyra) and present experiments and observations (with caveats).

1.1 Contributions of this Thesis

The main contributions of this thesis are:

- We present experimental comparisons of the partitioning strategies present in three distributed graph processing systems (PowerGraph, GraphX, PowerLyra);
- For each system we provide rules of thumb to help developers pick the right partitioning strategy;
- We implement PowerGraph's and GraphX's partitioning strategies into PowerLyra (along with a new variant);
- We similarly implement all strategies from PowerGraph and PowerLyra into GraphX;
- We present experimental comparisons of all strategies across all systems, and discuss our conclusions.

In particular, we find that the performance of a partitioning strategy depends on: (1) the degree distribution of the input graph, (2) the characteristics of the application being run, and (3) the number of machines in the cluster. Our results demonstrate that the choice of partitioning strategy has a significant impact on the performance of the system; e.g., for PowerGraph, we found that selecting a suboptimal partitioning strategy could lead to an overall slowdown of up to 1.9× times compared to an optimal strategy, and a >3× slowdown in just computation time alone. Similarly, we have observed significant differences in resource utilization based on the partitioning strategy used, e.g., there is a 2× difference in PageRank peak memory utilization between different partitioning strategies in PowerLyra. Finally, when all partitioning strategies are implemented in one system we find that our per-system decision trees do not change, but partitioning strategies tightly integrated with the underlying engine perform better. This means that our per-system results still hold value.

1.2 Outline of this Thesis

This thesis is organized as follows.

1. In Chapter 2, we discuss the related work in the areas of graph pro-

cessing and graph partitioning.

2. In Chapter 3 we provide some background on the graph computation models (Pregel and GAS), edge-cuts vs vertex-cuts, and the applications used for the evaluation.

3. In Chapter 4, we cover the experimental methodology of our experiments.

4. In Chapter 5, we introduce PowerGraph, its partitioning strategies and discuss our results related to the system.

5. Chapters 6 and 7 similarly cover PowerLyra and GraphX respectively.

6. Additionally Chapters 8 and 9 cover these systems respectively with all partitioning strategies ported from the other systems.

7. Finally, we conclude in Chapter 10.

1.3 Summary of Results

We now provide a brief summary of the results of our experiments. These results are from experiments performed on three systems and their associated partitioning strategies: PowerGraph, PowerLyra, and GraphX, as well as multiple different applications and real-world graphs. Table 1.1 lists the individual partitioning strategies we evaluate in this thesis.

For **PowerGraph**, we found that heuristic-based strategies, i.e., HDRF and Oblivious, perform better (in terms of both ingress and computation time) with graphs that have low-degree distribution and large diameters such as road networks. Grid incurs lower replication factors as well as a lower ingress time for heavy-tailed graphs like social networks. However, for power-law-like graphs such as UK-web, the two heuristic strategies deliver higher quality partitions (i.e., lower replication factors) but have a longer ingress phase when compared to Grid. Therefore, for power-law-like graphs, Grid is more suitable for short running jobs and HDRF/Oblivious are more suitable for long running jobs.

For **PowerLyra**, we need to additionally consider if the application being run is natural or not; Hybrid is significantly more efficient when used with natural applications. Natural applications are defined as applications which Gather from one direction and Scatter in the other (terms explained later in Chapter 3). We have provided two decision trees based on these findings: PowerGraph (Figure 5.9) and PowerLyra (Figure 6.6). These decision trees and the results that build up to them have been discussed in more detail in Sections 5.4 and 6.4, respectively.

For **GraphX**, all partitioning strategies have similar partitioning speed, i.e., the partitioning phases took roughly the same amount of time. So, the choice of partitioning strategy is based primarily on computation time. Our results indicate that Canonical Random works well with low degree graphs, and 2D edge partitioning with power-law graphs. These results are discussed in Section 7.4.

When **all partitioning strategies** are implemented and run in a common system (PowerLyra), we find that decision trees do not change, asymmetric random performs worse than random, and that the engine enhances some partitioning strategies more than others. We also find that CPU utilization is not a good indicator of performance. Similarly when all partitioning strategies are implemented in GraphX, we see that the decision process again changes very little. While performing these experiments, we also look into the effects of memory pressure on GraphX.

CHAPTER 2

RELATED WORK

Graph processing as well as Graph partitioning have been widely researched. In this chapter we present related work in these fields.

2.1 Graph Processing

There are several distributed graph processing systems that were not evaluated in this thesis. They include Pregel [3], LFGraph [7], Apache Giraph [5], GPS [12], Picollo [13], Pegasus [14] and Mizan [15].

Moreover, just as GraphX is built on Spark, other dataflow frameworks like Naiad [16] can also be used for graph processing. Husky [17], which aims to present a fine-grained yet high-level abstraction for distributed computation, has also shown promising results for graph processing.

There are also works which try to perform large-scale graph processing on a single large scale machine such as Ligra [18, 19] and GraphChi [6]. The motivation for these systems is that before scalability, a system should focus on efficiency. Gemini [20] is a graph processing system that, while distributed, prioritizes efficiency over scalability. Gemini shares a lot of optimizations with Ligra (such as the hybrid push-pull mechanism and compact vertex representation) and GraphChi (work-stealing). COST [21] is another work that shows that over-prioritizing scalabity can lead to systems that just "parallelize overheads".

The aforementioned GraphChi system uses secondary storage to store edges (which vastly outnumber vertices) to fit large graphs in a single machine. X-Stream [22] builds on top of this with an added edge-centric computation model. Chaos [23] further builds on top of it and allows scaling out.

There are also systems like Medusa [24] which aim to offload graph processing to GPUs to get a performance boost.

Kineograph [25] is a distributed graph processing system that allows graph processing on continuously changing graphs. Graph databases have also recently emerged to store, update and query such ever-changing graphs. Examples of such databases include Neo4j [26], titan[27], and Weaver [28].

2.2 Graph Partitioning

The PowerGraph paper [2] contains descriptions of the Oblivious and Coordinated strategies. Similarly, PowerLyra [29] covers the strategies Hybrid and Hybrid-Ginger. Moreover, PowerLyra has also been extended with strategies specifically catering to bipartite graphs [9]. A description of Grid and other constrained strategies can be found in the Graphbuilder paper [30].

Gemini [20] also includes a chunk-based partitioning scheme that leverages the natural locality in real world graphs.

There are several offline graph partitioning algorithms such as METIS [31], Fennel [32] and Ja-be-ja [33].

GraphH [34] is a system that performs network-aware graph partitioning as well as dynamic migration of edges during computation. LeBeane et. al. [35] take the partitioning strategies from PowerGraph and PowerLyra and modify them for heterogeneous clusters.

The Leopard [36] paper presents an edge partitioning algorithm tailored for dynamically changing graphs.

Pundir et. al. [37], present algorithms for re-partitioning the graph for mid-computation scale-out.

2.3 Other Evaluations

Anwar et. al. [38] present and compare offline partitioning strategies specifically optimized for spatial road networks and for k-means clustering.

Zorro [39], a work that add zero-cost fault-tolerance to PowerGraph, also evaluates the effects of different partitioning strategies on their fault-recovery system.

Han et. al. [11] evaluate and analyze multiple pregel-like graph processing systems and suggest ways to improve all of them.

All of these works differ in that they do not analyze *online* graph partitioning strategies for *general purpose* and *distributed* graph processing.

CHAPTER 3

BACKGROUND

This Chapter provides background information on (1) the Gather-Apply-Scatter (GAS) model, (2) the difference between edge-cuts and vertex-cuts, and (3) the different graph applications used in the evaluation.

3.1 The GAS Decomposition

The Pregel [3, 40] model of vertex-centric computation involves splitting the overall computation into *supersteps*. Vertices communicate with each other by passing messages, where messages sent in one superstep are received in the by neighbors in the next. This model is also available in systems such as Giraph and GraphX.

Similarly to Pregel, Gather-Apply-Scatter (GAS) is a model for vertex-centric computation used by systems such as PowerGraph, GraphLab [41, 42], and PowerLyra. In this model, the overall vertex computation is divided into iterations, and each iteration is further divided into *Gather*, *Apply* and *Scatter* stages (also called *minor-steps*).

In the **Gather** stage, a vertex essentially collects information about adjacent edges and neighbors and aggregates it using the specified commutative associative aggregator. In the **Apply** stage, the vertex receives the gathered and aggregated data and uses it to update its local state. Finally, in the **Scatter** stage, the vertex uses its updated state to trigger updates on the neighbouring vertices' values and/or activate them for the next iteration. The vertex program written by the user specifies to which neighbouring vertices to gather or scatter. The user specifies the gather, apply and scatter methods to be executed in their corresponding stages. The user also specifies a commutative associative aggregator for the gather stage.

3.2 Edge Cuts and Vertex Cuts

There are two main partitioning and computational approaches in distributed graph processing: (1) *edge-cuts*, as used by systems such as GraphLab [41], LFGraph [7], and the original version of Pregel, and (2) *vertex-cuts*, as used by systems such as PowerGraph and GraphX. In LFGraph and Pregel, partitions themselves communicate with each other for each such cut edge. In Distributed GraphLab, partitions maintain replicas of vertices connected by cut edges.

For systems that utilize edge-cuts, vertices are assigned to partitions and thus edges can span partitions. For systems that utilize vertex-cuts, edges are assigned to partitions and thus vertices can span partitions. Unlike edges which could be cut across only two partitions, a vertex can be cut across several as its edges may be assigned to several partitions.

Edge-cuts and vertex-cuts are preferable in different scenarios as pointed out by [9]. Edge-cuts are better for graphs with many low-degree vertices since all adjacent edges of a vertex are allocated to the same machine. However, for power-law-like graphs with several very high degree nodes, vertex-cuts allow better load balance by distributing load for those vertices over multiple machines.

3.3 Graph Applications

Graph applications differ along multiple axes: initial conditions, direction of data-flow, presence of edge-mutation, and synchronization. To capture a wide swathe of this space, we have selected the following applications for use in our subsequent experimental evaluations.

3.3.1 PageRank

PageRank is an algorithm used to rank vertices in a graph, where a vertex is ranked higher if it has incoming edges from other high-rank vertices. PageRank first starts by assigning each vertex a score of 1, and then updates the vertex score $p(v)$ in each superstep using v's neighboring vertices as:

$$p(v) = (1 - d) + d \cdot \sum_{v' \in N_i(v)} \frac{p(v')}{|N_o(v')|}$$

Here, d is a dampening factor (typically set to 0.85) and $N_o(v)$ and $N_i(v)$ are the set of out- and in-neighbors, respectively, of vertex v.

3.3.2 Weakly Connected Components

This identifies all the weakly connected components of a graph using label propagation. All vertices start out with their vertex id as their label id. Upon receiving a message form a vertex with a lower label id, they update their label id and propagate that label to all of their neighbours. At the start of computation, all vertices are active and send out their label IDs. The update rule can be formalized as:

$$p(v) = \min_{v' \in N(v)} (p(v'))$$

where $N(v)$ is the set of all neighbours of v. After convergence, all vertices have the the lowest vertex ID in its weakly connected component as its value.

3.3.3 K-Core Decomposition

A graph is said to have a *k-core* if it contains a subgraph consisting entirely of nodes of degree at least k; such a subgraph is called a *k-core*. K-core decomposition is the process of finding all such k-cores, and is performed for a given k by repeatedly removing nodes of degree less than k. The PowerGraph application accepts a *kmin* and *kmax* value and finds all k-cores for all values of k in between.

3.3.4 SSSP

Single Source Shortest Path (SSSP) finds the shortest path given a source vertex to all reachable vertices. SSSP first starts by setting the distance value of the source vertex to 0 and all other vertices to ∞. Initially only the source is active. In each superstep, all active vertices send to their neighbours their current distance from the source. In the next step, if a vertex receives a distance smaller than its own, it updates its distance and propagates the new distance value. This continues until there are no more active vertices left. The update step for any active vertex is:

$$p(v) = \min_{v' \in N(v)} (p(v') + 1)$$

This update step can be easily modified for cases that involve directed or weighted edges.

3.3.5 Simple Coloring

The Simple Coloring application assigns colors to all vertices such that no two adjacent vertices have the same color. Minimal graph coloring is a well-known NP-complete problem [43]. This application, therefore, does not guarantee a minimal coloring. All the vertices initially start with the same color and, in each iteration, each active vertex assigns itself the smallest integer (color) different from all of its neighbours':

$$p(v) = \arg\min_{k} \{k | k \neq p(v') \forall v' \in N(v)\}$$

CHAPTER 4

EXPERIMENTAL METHODOLOGY

In this chapter we describe the clusters and datasets used for the experiments as well as the metrics we measure during our experiments. Several system-specific metrics and setup details are covered in their respective chapters.

4.1 Clusters

Detailed descriptions of our experimental environments are provided in Table 4.1. For both PowerGraph and PowerLyra, we performed our experiments on three different clusters: (1) a local cluster of 9 machines (to accommodate the perfect-square machine requirement for Grid partitioning), (2) an EC2 cluster consisting of 16 m4.2xlarge instances, and (3) an EC2 cluster consisting of 25 m4.2xlarge instances. For GraphX we used a local cluster of 10 machines.

4.2 Datasets

The datasets were obtained from SNAP (Stanford Network Analysis Project) [44], LAW (Laboratory for Web Algorithmics) [45] and DIMACS challenge 9 [46]. We used a mixture of low-degree and power-law-like graphs. A summary of the datasets has been provided in Table 4.2. All the datasets were stored in plain-text edge-list format.

Table 4.1: The Cluster Specifications.

Cluster	Sizes	Memory	Storage	vCPUs
Local	9 & 10	64GB	500GB SSD	16 (2 X 4-core Intel Xeon 5620 w/ hyperthreading)
EC2 (m4.2xlarge)	16 & 25	32GB	250GB EBS SSD	8 (2.4 GHz Intel Xeon E5-2676 v3 (Haswell))

Table 4.2: The graph datasets used.

Graph Dataset	Edges	Vertices	Type
road-net-CA [44]	5.5M	1.9M	Low-Degree
road-net-USA [46]	57.5M	23.6M	Low-Degree
LiveJournal [44]	68.5M	4.8M	Heavy-Tailed
Enwiki-2013 [47, 48]	101M	4.2M	Heavy-Tailed
Twitter [49]	1.46B	41.6M	Heavy-Tailed
UK-web [47, 48]	3.71B	105.1M	Power-Law

4.3 Metrics

The primary metrics used in our experiments are:

- **Ingress time**: the time it takes to load a graph to memory (how *fast* a partitioning scheme is).
- **Computation time**: the time that it takes to run any particular graph application and always excludes the ingress/partitioning time.
- **Replication factor**: the average number of images per vertex for any partitioning strategy.
- **System-wide resource usage**: we measured memory consumption, CPU utilization and network usage at 1 second intervals.

To measure the system-wide resource metrics, we used a python library called psutil[1]. The peak memory utilization (per-machine) for an application was calculated by taking the difference between the maximum and minimum memory used by the system during experiment. This allows us to filter out the background OS memory utilization and still measure memory in an system independent way. For network IO, we found the incoming and outgoing IO patterns to be similar. So, we focus only on the incoming traffic.

We launched the system monitors on all machines a few seconds before the experiment begins and terminated the monitors a few seconds after the experiment ended. This ensured that the monitoring overhead was small and constant and also helped us accurately estimate background memory utilization. This method is similar to the one used by Han et. al. in [11].

[1] https://github.com/giampaolo/psutil

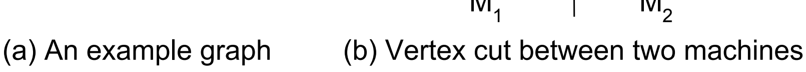
CHAPTER 5

POWERGRAPH

In this chapter we introduce the PowerGraph graph processing system, the partitioning strategies it ships with, and our experimental results.

5.1 System Introduction

PowerGraph [2] is a distributed graph processing framework written in C++ and designed to explicitly tackle the power-law degree distribution typically found in real-world graphs. The authors of PowerGraph discuss how edge-cuts perform poorly on power-law graphs and lead to load imbalance at the servers hosting the high-degree vertices. To solve the load-imbalance, they introduced vertex-cut partitioning, where edges instead of vertices were assigned to partitions.



(a) An example graph (b) Vertex cut between two machines

Figure 5.1: PowerGraph's vertex replication model.

5.1.1 Vertex Replication Model

Vertex cuts allow an even load balance but result in *replication* of the cut vertices. Whenever an edge (u, v) is assigned to a partition, the partition maintains a vertex replica for both u and v. For vertices which have images in more than one partitions, PowerGraph randomly picks one of them as the *master* and the remainder are called *mirrors*.

For a vertex, the total number of mirrors plus the master is called the vertex's *replication factor*. A common metric to measure the effectiveness of partitioning in PowerGraph is to calculate the average replication factor over all vertices [9, 2, 50]. Lower replication factors are associated with lower communication overheads and faster computation.

5.1.2 Computation Engine

PowerGraph follows the GAS model of computation, and allows the Gather and Scatter operations to be executed in parallel among machines. More specifically, all of a vertex's mirrors perform a local Gather, and then send the partially aggregated data to the master which will in turn perform another aggregation over the partial aggregates. Then in the Apply step, the master updates its local value and synchronizes all its mirrors. Thereafter, all the mirrors perform the Scatter step in parallel.

PowerGraph can be used with both *synchronous* and *asynchronous* engines. When run synchronously, the execution is divided into supersteps, each consisting of the Gather, Apply, and Scatter minor-steps. There are barriers between the minor-steps as well as the supersteps. When run asynchronously, these barriers are absent.

5.2 Partitioning Strategies

PowerGraph provides five partitioning strategies: (1) Random, (2) Oblivious, (3) Grid, (4) PDS, and (5) HDRF.

5.2.1 Random

In PowerGraph's Random hash-partitioning implementation, an edge's hash is the function of the vertices it connects. The hashing function ignores the direction of the edge, i.e., directed edges (u, v) and (v, u) hash to the same machine. Random is often appealing because it: (1) is fast, (2) distributes edges evenly, and (3) is highly parallelizable. However, Random creates a large number of mirrors.

5.2.2 Oblivious

The Oblivious graph partitioning strategy is based on a greedy heuristic with the objective of keeping the replication factor as low as possible. Oblivious incrementally and greedily places edges in a manner that keeps the replication factor low. The heuristic devolves to a few simple cases which are described in Appendix A in detail.

The heuristic requires some information about previous assignments to assign the next edge. Therefore, unlike Random, this is not a trivial strategy to parallelize and distribute. In the interest of partitioning speed, Oblivious does not make machines send each other information about previous assignments, i.e., each machine is "oblivious" to the assignments made by the other machines and thus makes decisions based on its own previous assignments.

5.2.3 Constrained

Constrained partitioning strategies hash edges, but restrict edge placement based on vertex adjacency in order to reduce the replication factor. This additional restriction is derived by assigning each vertex v a constraint set $S(v)$. An edge (u, v) is then placed in one of partitions belonging to $S(u) \cap S(v)$. As a result, Constrained partitioning imposes a tight upper bound of $|S(v)|$ on the replication factor of v. There are two popular strategies from constrained family offered by PowerGraph: Grid and PDS.

Grid [30] organizes all the machines into a square matrix. The constraint set for any vertex v is the set of all the machines in the row and column of the machine v hashes to. Thus, as shown in Figure 5.2, all edges can be stored on at least 2 machines. As a result, Grid manages to place an upper

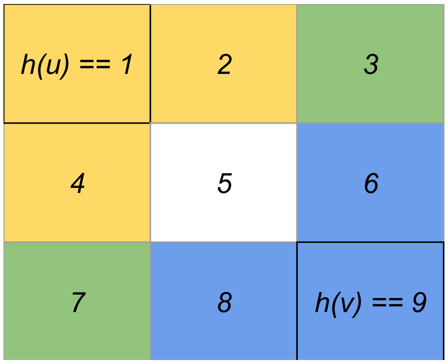


Figure 5.2: Grid Partitioning example: u hashes to 1, v hashes to 9. The edge (u, v) can be placed on machines 7 or 3.

bound of $(2\sqrt{N} - 1)$ on the replication factor where N is the total number of machines.

While Grid partitioning can generally work for any non-prime number of machines, whereby we construct an $(m \times n)$ rectangle (m and n are integers such that $m \times n = N$ and neither m nor n equals 1), the version offered by PowerGraph only works with a perfect square number of machines.

PDS uses Perfect Difference Sets [51] to generate constraint sets. However, PDS requires $(p^2 + p + 1)$ machines where p is prime. Since we were unable to satisfy the constraints of both PDS and Grid on the number of machines simultaneously, and therefore could not directly compare the two strategies, we have not included PDS in our evaluation.

5.2.4 HDRF

HDRF is a recently-introduced partitioning strategy that stands for High-Degree Replicated First [50]. HDRF is similar to oblivious, but while Oblivious breaks ties by looking at partition sizes (to ensure load-balance), HDRF looks at vertex degrees as well as partition sizes. As the name suggests, it prefers to replicate the high degree vertices when assigning edges to partitions. So, while assigning an edge (u, v), HDRF may make an assignment to a more loaded machine instead, if doing so results in less replication for the lower degree vertex between u and v.

To avoid making multiple passes, HDRF uses partial-degrees over actual degrees. HDRF updates partial-degree counters for vertices as it processes edges and uses these counters in its heuristics. The authors found no significant difference in replication factor, upon using actual degree instead of partial degree. Details can be found in Appendix B.

5.3 Experimental Setup

We ran all graph applications mentioned in the Section 3.3 with all partitioning strategies from Section 5.2. All applications were run until convergence. k-core decomposition was run with *kmin* and *kmax* set to 10 and 20 respectively. PowerGraph, by default, uses a number of threads equal to two less than the number of cores. We used the Road-net-CA, Road-net-USA, LiveJournal, Twitter, and UK-web datasets. All datasets were split into as many blocks as there are machines in the cluster to allow parallel loading.

5.4 Experimental Results

In this section, we discuss the results for PowerGraph. These results hold for PowerLyra as well.

5.4.1 Replication Factor and Performance

Figures 5.3, 5.4, and 5.5 show per-machine network IO, computation time and per-machine peak-memory usage plotted against replication factor as it is varied by picking different partitioning strategies. We see that all three performance metrics are increasing linear functions of replication factor.

All plots illustrate results with UK-Web on the EC2-25 cluster. We found similar behaviors for all graphs and cluster sizes, and we elide redundant plots. The choice of application only affects the slope of the line. We can see that this is true for all applications except Simple Coloring, which deviates from the trend due to its execution on the asynchronous engine. The asynchronous engine sometimes 'hangs' and consequently takes much longer to finish (Oblivious) and sometimes just fails (HDRF).

The linear correlation between network usage and replication factors (Figure 5.3) results from synchronization between mirrors and masters after each step. In the Gather step, $(n-1)$ replicas will send their partially aggregated

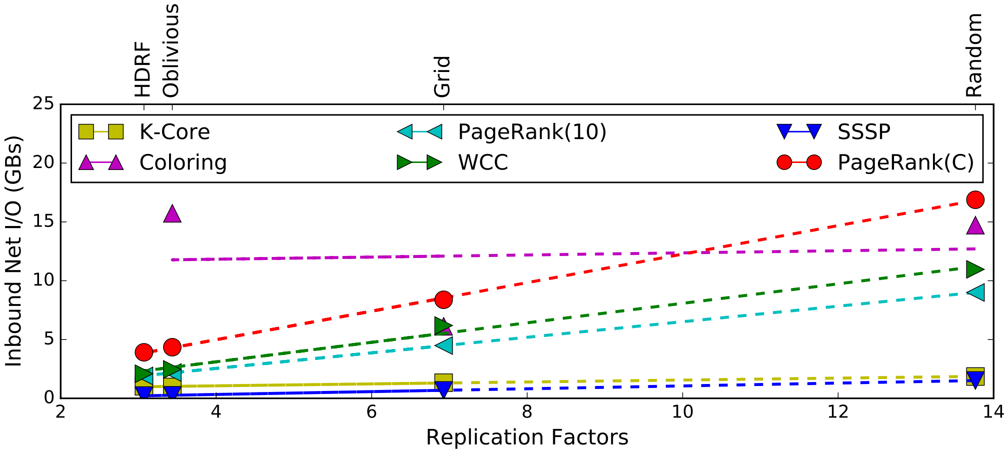


Figure 5.3: Incoming Network IO vs. Replication Factors. (PowerGraph, EC2-25, UK-Web).

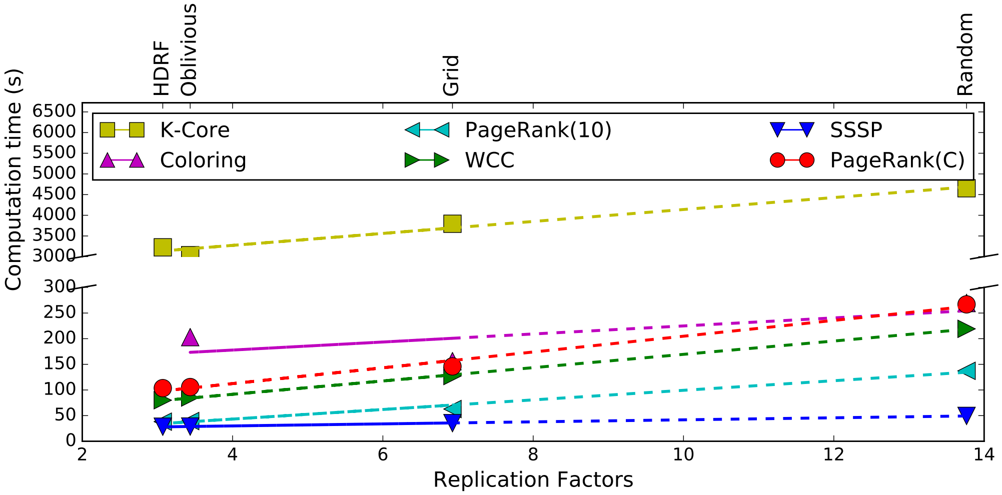


Figure 5.4: Computation Time in seconds vs. Replication Factors. (PowerGraph, EC2-25, UK-Web).

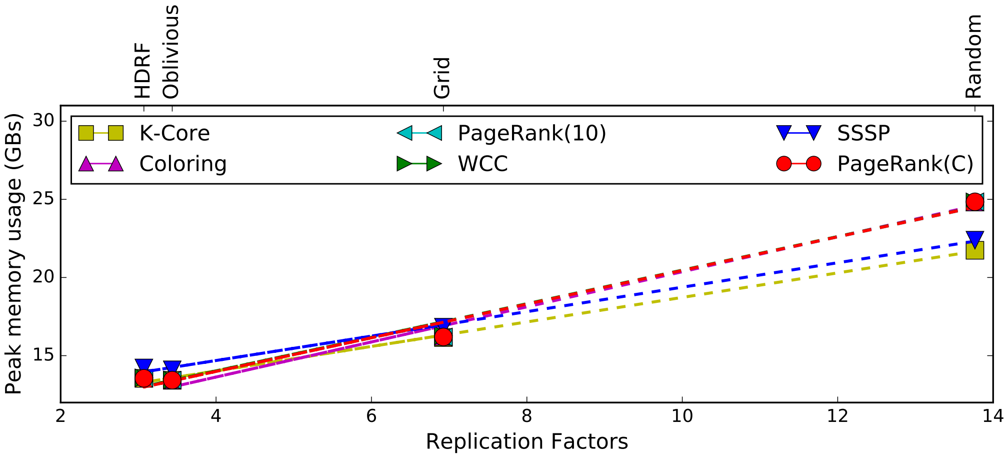


Figure 5.5: Memory usage vs Replication Factors. (PowerGraph, EC2-25, UK-Web).

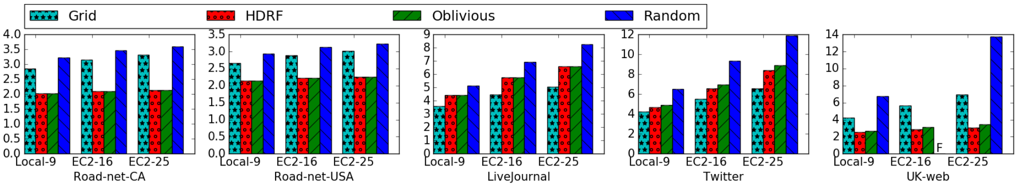


Figure 5.6: Replication Factors in Powergraph.

values to the master; after the Apply step, the master will send its $(n-1)$ replicas the updated vertex state (where n is the vertex's replication factor). The linear correlation between replication factor and computation time (Figure 5.4) can be explained by: (1) the additional computation requirements because of having more replicas, and (2) having to wait longer for network transfers to finish as the amount of data to be transferred is larger. The linear correlation between vertex replication and memory usage occurs because all vertex replicas are stored in memory (hence, having more replicas leads directly to higher memory consumption).

In light of the observation that replication factors are a reliable indicator of the resource usage due to partitioning (Figures 5.3, 5.4 and 5.5), from here on we will primarily use replication factor to compare the partitioning strategies. Figure 5.6 shows the replication factors for all of PowerGraph's partitioning strategies on all graphs and cluster sizes.

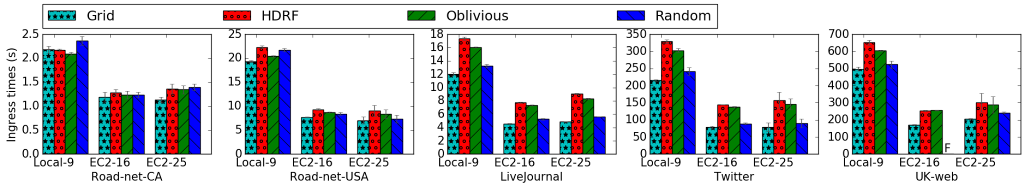
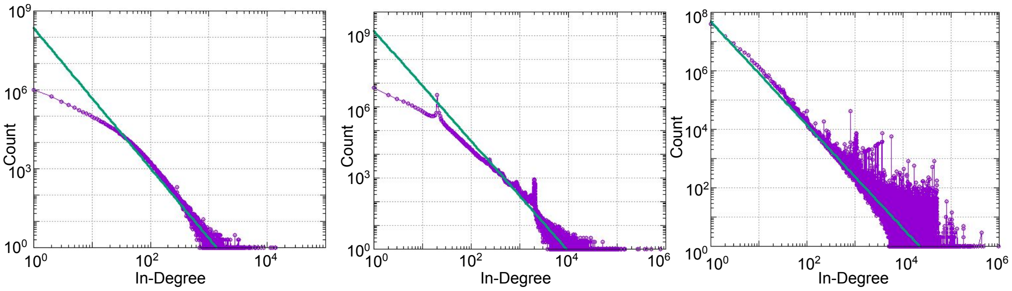


Figure 5.7: Ingress Time in seconds in PowerGraph.



(a) LiveJournal (b) Twitter (c) uk-web

Figure 5.8: In-degrees of the three powerlaw graphs used.

5.4.2 Minimizing Replication Factor

For Twitter and LiveJournal, Grid delivers the best (lowest) replication factor. For UK-web however, Grid's replication factor is worse than that of HDRF/Oblivious.[1] Although all the three graphs are skewed, heavy-tailed, and natural, they differ when it comes to low-degree nodes. In Figure 5.8, we see that relative to the power-law regression line, Twitter and LiveJournal have fewer low-degree nodes (unlike UK-web). This is why HDRF and Oblivious perform better than Grid for UK-web but not for Twitter and LiveJournal. The heuristic strategies perform better with low-degree vertices. In fact, HDRF was explicitly designed to lower the replication factor for low-degree vertices. As a result HDRF/Oblivious also deliver better replication factors for the entirely low-degree road-network graphs. Therefore, in terms of replication factor, HDRF/Oblivious are better for power-law-like graphs and low-degree graphs while Grid is preferable for heavy-tailed graphs.

[1] HDRF is a parameterized variant of Oblivious, and we use the recommended value of the parameter $\lambda = 1$. In practice, this causes HDRF and Oblivious to perform similarly.

5.4.3 Partitioning Quality vs Partitioning Speed

In Figure 5.7, we have plotted the ingress times for all the partitioning strategies. Hash-based partitioners are faster for power-law graphs in all cluster sizes, while all strategies perform similarly on low degree road network graphs. From Figure 5.7, we can see that Grid is usually the fastest in terms of ingress speed, followed by Random.

For low-degree road-network graphs, HDRF and Oblivious have the lowest replication factors (Figure 5.6) as well as fast ingress (Figure 5.7). Meanwhile, for heavy-tailed graphs like social networks, Grid delivers the lowest replication factors as well as the fastest ingress speed. However, for UK-web, Grid has the fastest ingress but HDRF has the best replication factors. Thus, for graphs like UK-web we need to look at the type of applications being run. If the application spends more time in the compute phase than in the partitioning phase, it will benefit more from lower replication factor; if it spends longer in the partitioning phase, it will benefit more from faster ingress.

Let us use the following example to demonstrate the effect of job duration on the choice of partitioning strategy: running PageRank and k-core decomposition with UK-web on the EC2-25 cluster. We show the ingress and computation times in Table 5.1. We see that for short running PageRank, the ingress phase is much longer than the computation phase. Therefore Grid, which has faster ingress, has a better total job duration, even though HDRF has a faster compute phase. On the other hand, for applications with a high compute/ingress ratio like k-core, a faster compute phase is better for the overall job duration. Therefore, when the compute/ingress ratio is lower, faster ingress is better.

When a graph may be partitioned, saved to disk, and reused later, such cases should be treated similar to the high compute/ingress ratio case (assuming that partitions will be reused enough times, compute becomes larger than ingress) and lower replication factor should be the priority.

5.4.4 Picking a Strategy

On the basis of these results, we present a decision tree to help users select a partitioning strategy (Figure 5.9). For low-degree graphs we recommend

Table 5.1: Time taken (seconds) by HDRF and Grid in the ingress and compute phases. Bold font highlights the stage which had the largest impact on the total runtime. (PowerGraph, EC2-25, UK-web).

Strategy	PageRank (Conv.)			K-Core Decomp.		
	ingress	compute	total	ingress	**compute**	total
Grid	206.4	146.0	352.4	203.6	3794.9	3998.5
HDRF	322.0	103.6	425.6	320.6	3225.1	3545.7

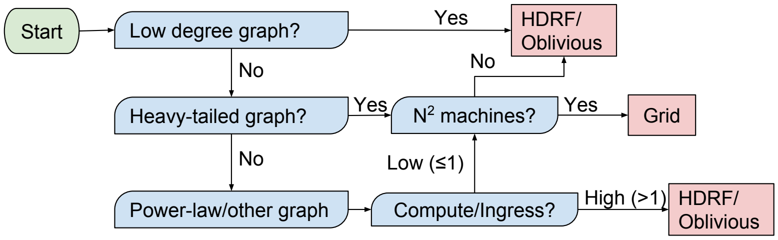


Figure 5.9: Our decision tree for picking a partitioning strategy with PowerGraph.

HDRF/Oblivious. For heavy-tailed graphs like social networks, we recommend Grid, if the cluster size permits. If Grid is not possible, then fall back on HDRF/Oblivious. For graphs that follow the power-law distribution more closely, HDRF/Oblivious are the strategy of choice. Finally, we note that because of Random's consistently high replication factor, it should generally be avoided. Even though Random has fast ingress, Grid demonstrates similar (or better) ingress times consistently, while delivering better replication factors.

CHAPTER 6

POWERLYRA

We introduce PowerLyra, its partitioning strategies, the setup used for it, and present our experimental results.

6.1 System Introduction

PowerLyra [9] is a graph analytics engine built on PowerGraph that seeks to further address the issue of skewed distribution in power-law graphs by performing differentiated processing and partitioning for high- and low-degree vertices. Its authors argue that applying vertex-cuts to low-degree vertices can lead to high communication and synchronization costs. Similarly, applying edge-cuts to high-degree vertices leads to imbalanced load and high contention. As a result, PowerLyra takes a best-of-both-worlds hybrid approach and applies edge-cuts to low-degree vertices and vertex-cuts to high-degree vertices.

PowerLyra's new partitioning strategies follow the hybrid philosophy. Two new strategies are proposed: (1) *Hybrid*, a random hash based strategy, and (2) *Hybrid-Ginger*, a heuristic-based strategy. Both partitioning strategies aim to perform vertex-cuts on high-degree vertices and edge-cuts on low-degree vertices; we discuss both in more detail in the next section. In addition, PowerLyra's new hybrid computation engine differentially processes high-degree and low-degree vertices by performing a distributed gather for high-degree vertices (as in PowerGraph), and a local gather for low-degree vertices (as in GraphLab/Pregel). PowerLyra implements both synchronous and asynchronous versions of this hybrid engine.

By performing edge-cuts on the low-degree vertices and placing them with their in-edges, PowerLyra is able to efficiently support *natural* graph applications–those algorithms that gather values in only one direction (e.g., from in-neighbors) and scatter in the other (e.g., to out-neighbors). Examples include PageRank, (directed) Single-Source Shortest Paths, etc. Since the low-degree vertices are placed with either their gather-neighbours or their scatter-neighbours, PowerLyra's approach lowers communication and synchronization costs significantly.

6.2 Partitioning Strategies

The latest version of PowerLyra, at the time of our writing, comes with PowerGraph's Random, Grid, PDS and Oblivious partitioning strategies, along with its own novel Hybrid and Hybrid-Ginger partitioning algorithms. As was the case with PowerGraph, we exclude PDS because of the reasons explained in Section 5.2.3.

6.2.1 Hybrid

Hybrid performs vertex-cuts for high-degree vertices, edge-cuts for low-degree vertices, and assigns each edge exclusively to its destination vertex. Hybrid places the edges with low-degree destinations by hashing the destination vertex, and the edges with high-degree destinations by hashing the source vertex. Using this approach, Hybrid minimizes the replication factor for low-degree vertices. Similarly to HDRF (Section 5.2.4), Hybrid also ensures that high-degree vertices have high replication factors in order to allow for better distribution and load balance for such vertices.

Unlike HDRF, Hybrid uses the actual degree of a vertex, rather than the partial degrees. Consequently, the strategy requires multiple passes over the data. During the first phase, Hybrid performs edge-cuts on all vertices and also updates the degree counters. In the second phase, called the reassignment phase, Hybrid performs vertex-cuts on the vertices whose degree is above a certain threshold. We use the default threshold value of 100.

6.2.2 Hybrid-Ginger

Hybrid-Ginger seeks to improve on Hybrid using a heuristic inspired from Fennel [32], a greedy streaming Edge-cut strategy. Hybrid-Ginger first partitions the graphs just like Hybrid but then in an additional phase, tries to further reduce the replication factors for low degree vertices through the heuristic. The heuristic is not used for high-degree vertices and is also modified to account for load-balance.

The heuristic tries to place a low degree vertex v in the partition that has more of its in-neighbours. Here, v gets assigned to a partition p that maximizes $c(v, p) = |N_i(v) \cap V_p| - b(p)$ where $N_i(v)$ is the set of v's in-edge-neighbours and V_p is the set of vertices assigned to p. The first term ($|N_i(v) \cap V_p|$) is the partition specific in-degree and the second term $b(p)$ is the load-balance factor. $b(p)$ represents the cost of adding another vertex to p by accounting for the number vertices and edges in p: $b(p) = \frac{1}{2}(|V_p| + \frac{|V|}{|E|}|E_p|)$ [9, 35].

6.3 Experimental Setup

The setup for PowerLyra was identical to that for PowerGraph (Section 5.3). We enabled PowerLyra's new hybrid computational engine and fixed a minor bug with Hybrid-Ginger that prevented it from running on UK-web.[1]

6.4 Experimental Results

In this section, we discuss the results for PowerLyra.

6.4.1 Hybrid Strategies and Natural Algorithms

We generally see a correlation between replication factors and performance for PowerLyra, similar to PowerGraph (Section 5.4.1). Unlike PowerGraph, PowerLyra is optimized for when Hybrid strategies are paired with natural algorithms which Gather from one direction and Scatter in the other (Section 6.1). Hybrid colocates the master replica of low-degree vertices with all in-edges, allowing PowerLyra to perform a local gather instead of the usual

[1] The integer type used to store the number of edges from the command-line options overflowed with UK-web.

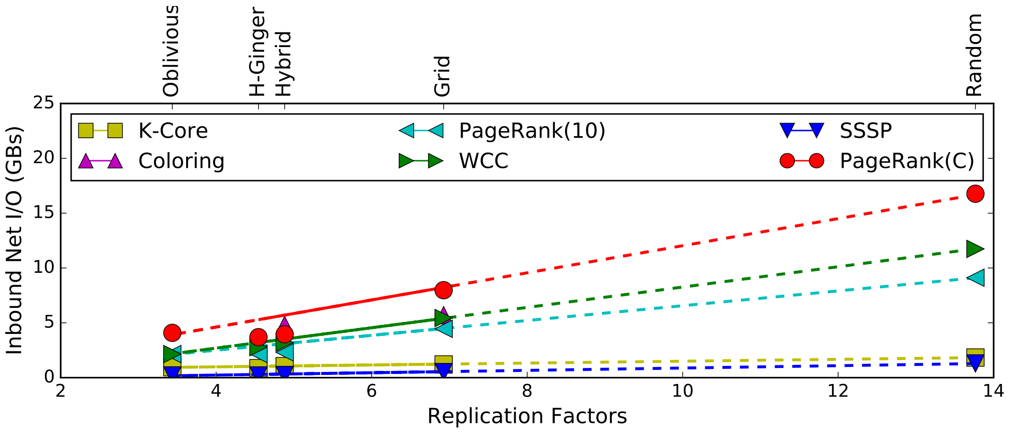


Figure 6.1: Incoming network IO vs. Replication Factor. (EC2-25, PowerLyra, UK-web).

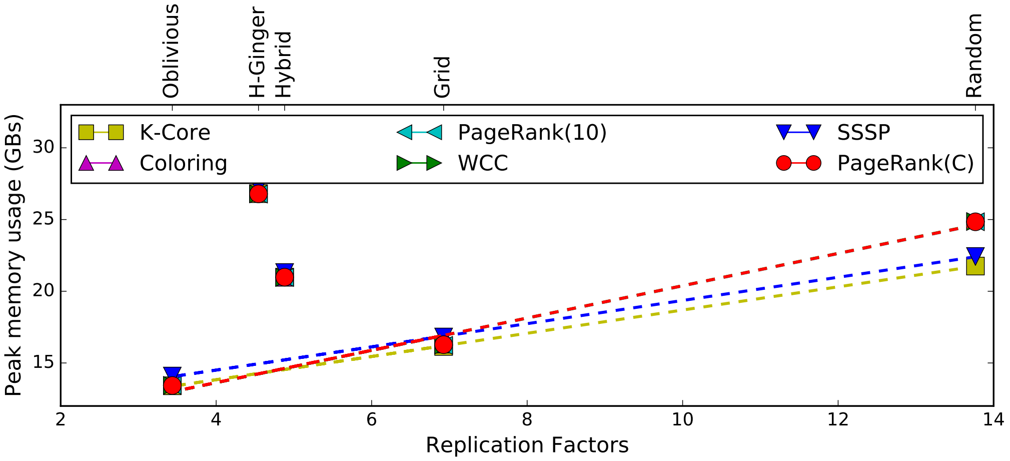


Figure 6.2: Peak memory utilization vs. Replication Factor. (EC2-25, PowerLyra, UK-web).

distributed gather. As a result, PowerLyra eliminates associated network and synchronization costs for low-degree vertices.

We can see the effect of this optimization when we look at compute-phase network usage plotted against replication factors (Figure 6.1). The Hybrid and Hybrid-Ginger datapoints have been intentionally ignored by the regression line to better highlight the effect of the optimization. We can see that Hybrid and Hybrid-Ginger use less network IO than Oblivious while running PageRank (a natural application), even though their replication factors are higher. Therefore, Hybrid strategies perform well when paired with natural algorithms. Since we used the undirected version of SSSP (which is not a natural algorithm) for the PowerGraph and PowerLyra experiments, we are

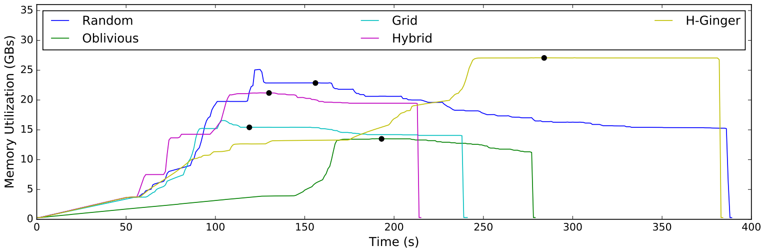


Figure 6.3: Average memory utilization over time. The black dots mark the end of ingress phase for each partitioning strategy. (EC2-25, PowerLyra, UK-web, PageRank).

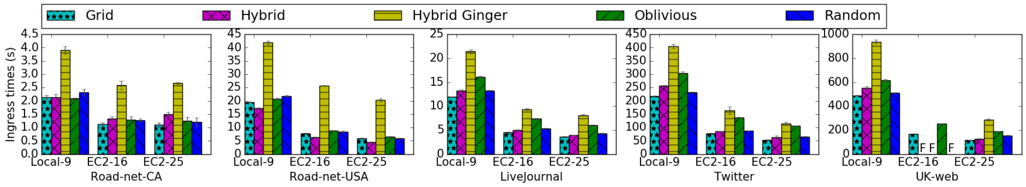


Figure 6.4: Ingress Times for PowerLyra.

unable to see network savings of similar magnitude.

6.4.2 Hybrid Strategies and Memory Overheads

From Figure 6.2, we can see that Hybrid and Hybrid-Ginger have a higher peak memory utilization than expected from their replication factor. We have again ignored the hybrid data points while drawing the regression line to highlight how much they deviate from the trend. In the timeline plot of memory utilization (Figure 6.3), we see that peak memory utilization is reached during the ingress phase (before the black dot) for each partitioning strategy. Therefore, we attribute Hybrid and Hybrid-Ginger's higher peak-memory usage to their partitioning overheads from additional phases. Unlike the other partitioning strategies (Section 5.2) which are all streaming single-pass strategies, Hybrid and Hybrid-Ginger have multiple phases. Hybrid

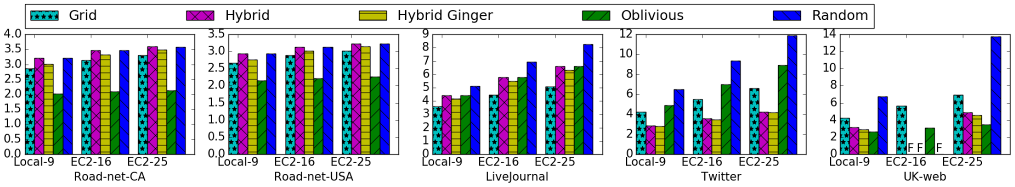


Figure 6.5: Replication Factors for PowerLyra.

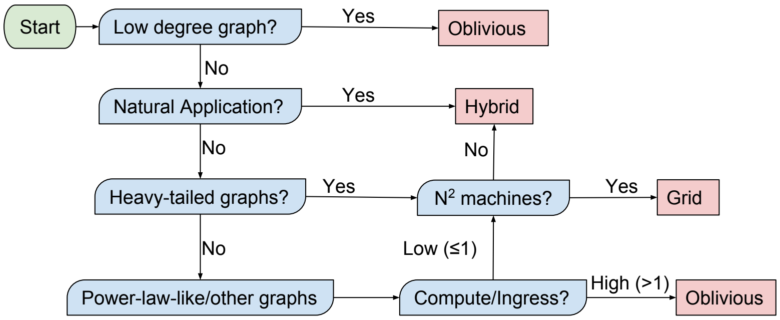


Figure 6.6: Our decision tree for PowerLyra's partitioning strategies.

reassigns high-degree vertices in its second-phase and Hybrid-Ginger has an additional phase on top of Hybrid to perform low-degree vertex reassignments on the basis of the Ginger heuristic. These additional phases contribute to the memory overhead. We can see the Hybrid-Ginger which has more phases also has a higher overhead.

6.4.3 Minimizing Replication Factor

Barring the above exceptions, replication factors are still a good indicator of performance in terms of network usage, memory and computation time. We have therefore provided ingress times and replication factors for PowerLyra's strategies on all graphs and cluster sizes in Figures 6.4 and 6.5.

Here, as in PowerGraph, Oblivious delivers the best replication factors for the low-degree road networks and UK-web graph. On the other hand, Grid and Hybrid both have low replication factors for LiveJournal and Twitter graphs. Thus, for heavy-tailed graphs, Grid would be preferable when possible as it has lower memory consumption even when it has a higher replication factor.

6.4.4 Picking a Strategy

We have provided a decision tree for PowerLyra in Figure 6.6. Most of the tree is similar to that for PowerGraph, but for PowerLyra, we also take into account if the application is natural as Hybrid synergizes well with such applications. Even so, Oblivious is a better choice for low-degree graphs because of the lower replication factors. Thus, we place the "Natural Application?"

decision node after the “Low degree graph?” node. For heavy-tailed graphs we again pick Grid if the cluster size allows it. When the cluster size is not a perfect square, we choose to fall back on Hybrid because of its similar performance (except for the higher memory usage). We also note that Hybrid-Ginger should generally be avoided in favor of Hybrid. Unlike [9], we do not find Hybrid-Ginger to be an improvement over Hybrid. Our results demonstrate that Hybrid-Ginger has significantly slower ingress (Figure 6.4), has a much higher memory footprint (Figure 6.3), and, in return, delivers only slightly better replication factor than Hybrid (Figure 6.5).

CHAPTER 7

GRAPHX

In this chapter, we introduce GraphX and its partitioning strategies, discuss the experimental setup and present the experimental results.

7.1 System Introduction

GraphX [4] is a distributed graph processing framework built on top of Apache Spark that enables users to perform graph processing while taking advantage of Spark's data flow functionality. The GraphX project was motivated by the fact that using general dataflow systems to directly perform graph computation is difficult, can involve several complex joins, and can miss optimization opportunities. Meanwhile, using a specialized graph processing tool in addition to a general dataflow framework leads to data-migration costs and additional system complexity in the overall data pipeline. GraphX addresses these challenges by providing graph processing APIs embedded into Spark.

GraphX leverages Spark's Resilient Distributed Datasets (RDDs) [52] to store the vertex and edge data in memory. RDDs are a distributed, in-memory, lazily-evaluated and fault-tolerant data structure provided by Spark. RDDs are connected via lineage-graphs (logs recording which operations on which old RDDs created the new RDD) and, through them, support lazy computation as well as fault-tolerance (based on checkpointing and re-computation). Therefore, unlike PowerGraph/PowerLyra which only support slow checkpointing, GraphX benefits from the fault-tolerance inherent to RDDs. Thus GraphX is structurally significantly different from PowerGraph/PowerLyra. GraphX also utilizes vertex-cuts to divide graph data into partitions.

7.2 Partitioning Strategies

GraphX comes with a variety of graph partitioning strategies: (1) Random, (2) Canonical Random, (3) 1D, and (4) 2D partitioning. These strategies are hash-based and stateless (they assign each edge independent of previous assignments), making them highly parallelizable streaming graph partitioning strategies. Moreover, as opposed to PowerGraph and PowerLyra, which typically assign one partition to each machine, GraphX allows for an arbitrary number of partitions per machine. A recommended rule of thumb is to use one partition per core in order to maximize parallelism.

7.2.1 Random and Canonical Random

GraphX's Random partitioning strategy assigns edges to partitions by hashing the source and vertex IDs. The Canonical Random strategy is similar, except that it hashes the source and vertex IDs in a canonical direction, e.g., edges (u, v) and (v, u) hash to the same partition under Canonical Random, but not necessarily under Random. Therefore, the Canonical Random strategy from GraphX is similar to PowerGraph's Random partitioning strategy (Section 5.2.1).

7.2.2 1D Edge Partitioning

1D Edge partitioning hashes all edges by their source vertex. As a result, this partitioning strategy ensures that all edges with the same source are collocated in the same partition. This strategy is similar to how PowerLyra's Hybrid strategy (Section 6.2.1) partitions its low-degree vertices.

7.2.3 2D Edge Partitioning

2D Edge partitioning is similar to PowerGraph's Grid (Section 5.2.3). This strategy also arranges all the partitions into a square matrix, and picks the column on the basis of the source vertex's hash and the row on the basis of the destination vertex's hash. As with the Grid partitioning strategy, this ensures a replication upper bound of $(2\sqrt{N} - 1)$ where N is the number of partitions. Moreover, the strategy works best if the number of partitions is a

Table 7.1: Computation time-based rankings for GraphX.

Application	Road-net-ca	Road-net-usa	LiveJournal	Enwiki-2013
PageRank	(1D,CR),(2D,R)	(1D,CR),(2D,R)	2D,1D,(CR,R)	(2D,1D),(CR,R)
SSSP	(CR,1D),(2D,R)	CR, (1D,R,2D)	CR,2D,1D,R	(1D,2D),(CR,R)
WCC	CR, 1D, 2D, R	CR, 1D, 2D, R	(2D,1D,CR),R	(1D,2D),(CR,R)

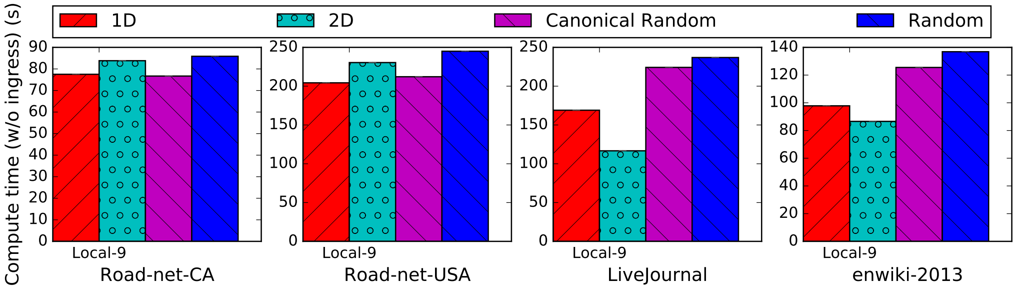


Figure 7.1: Computation times for PageRank on GraphX.

perfect square otherwise the next largest square number is used to build the grid and then the assignments are mapped back down to the correct number of partitions (potentially leading to imbalanced load).

7.3 Experimental Setup

For GraphX, we used SSSP, PageRank and WCC with 10 iterations.We ran our on experiments on a local cluster of 10 machines. GraphX ran out of memory while trying to load Twitter and UK-web. So we used Enwiki-2013 instead (Table 4.2). In GraphX, the partitioning phase is separate from, and follows after, the ingress phase. As a result partitioning time is measured separately from ingress and computation.

7.4 Experimental Results

Unlike PowerGraph/PowerLyra, GraphX only has hash-based partitioning schemes. Since all of GraphX's partitioning strategies are stateless and hash-based, they all run at similar speeds. The differences in peak memory utilizations were also not found to be noticeably large. Thus, computation time becomes the only metric on which to base the choice of partitioning strategy especially because, in GraphX, computation time was always found to be much larger than partitioning time. We show in Figure 7.1 the compute times for PageRank with all graphs used. The plots for other applications have been elided. We arrange, for each combination of graph application and input graph, the partitioning strategies in ascending order of computation time in Table 7.1. Partitioning strategies with performance close to each other parenthesized. We see that for road-network graphs, Canonical Random is consistently the fastest or the second fastest. Similarly for power-law graphs 2D edge partitioning is fastest or close to fastest.

This can be explained by the fact that for low-degree graphs, replication factor of a vertex is naturally bounded by its degree–thus the upper bound imposed by 2D edge partitioning (25 for 160 partitions) is not effective for road-networks (max degree 12). On the other hand, for the heavy-tailed and power-law-like graphs, where the max degree is much greater than 25, the upper bound helps keep the replication factor and thus execution time low.

Thus, we recommend Canonical Random for low-degree and high-diameter graphs such as road-networks and 2D partitioning for power-law-like graphs. Due to the straightforward conclusions we do not provide a decision tree.

CHAPTER 8

POWERLYRA: ALL STRATEGIES

In order to provide a uniform platform to compare partitioning strategies from across all three systems, we implemented all strategies in PowerLyra. We wished to implement faithful versions, and PowerGraph's and GraphX's strategies were significantly simpler than PowerLyra's.

8.1 Partitioning Strategies

From GraphX, we implemented 1D, 2D and Asymmetric Random (referred to as just 'Random' in GraphX). From PowerGraph we implemented HDRF. We also implemented a new strategy called 1D-Target (Section 8.2.3).[1] For GraphX, we used the Scala implementations as reference and ported it to C++ in PowerLyra. Since PowerLyra is a fork of PowerGraph, migrating the latter's strategies required fewer changes. We did not implement GraphX's "Canonical Random" as it is equivalent to PowerLyra's "Random". Similarly, PowerGraph's Random, Grid, and Oblivious are already present in PowerLyra.

8.2 Experimental Results

We ran all experiments on Local-9 and EC2-25 clusters, with the same setup as in Section 6.3. The plots comparing all 9 strategies appear in Figures 8.1 and 8.2. We describe below our key observations.

[1] See `https://gitlab-beta.engr.illinois.edu/sverma11/powerlyra-extra-partitioners` for source code.

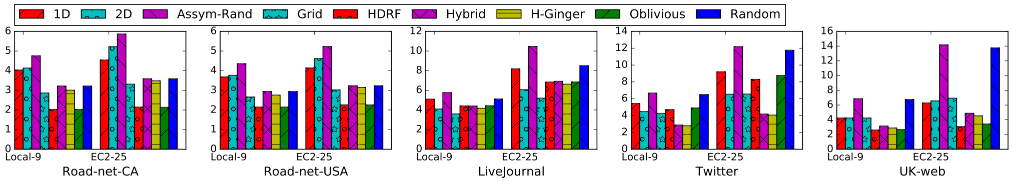


Figure 8.1: Replication Factors for PowerLyra with all Strategies.

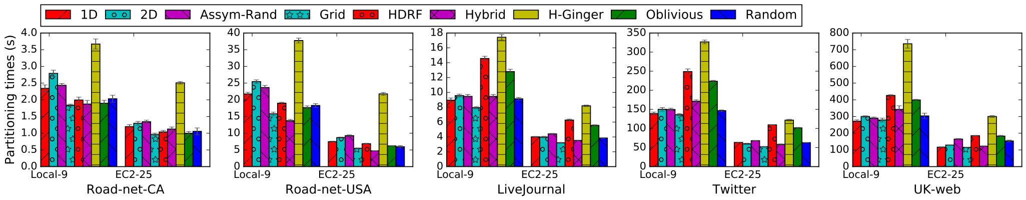


Figure 8.2: Ingress Times for PowerLyra with all Strategies.

8.2.1 No Effect on Decision Trees

We observe that a non-native strategy almost never outperform best preexisting PowerLyra strategy. The only exception is HDRF, which has similar performance to Oblivious. As a result, the decision tree for PowerLyra including all partitioning strategies is almost identical to that without (Figure 6.6), with the only difference being the replacement of 'Oblivious' with 'HDRF/Oblivious'. The relative performances of PowerGraph's strategies remained similar. The relative performances of GraphX's strategies were different after they were implemented in PowerLyra. This could have been due to multiple reasons:

(1) GraphX's use of RDDs, which are not present in the PowerLyra; and

(2) PowerLyra's Hybrid engine favoring 2D and 1D (see Section 8.2.3).

This indicates to us that the performance of a partitioning strategy in practice is correlated to how tightly it is integrated into its native system. It is possible that with further effort and optimizations the non-native strategies performance in PowerLyra could be improved (but this is beyond the scope of this thesis).

8.2.2 Asymmetric Random worse than Random

Random initially was the partitioning strategy that consistently incurred the highest replication factors in PowerLyra (Figure 6.5). However, Asymmetric Random (which doesn't guarantee that edges (u, v) and (v, u) get placed to

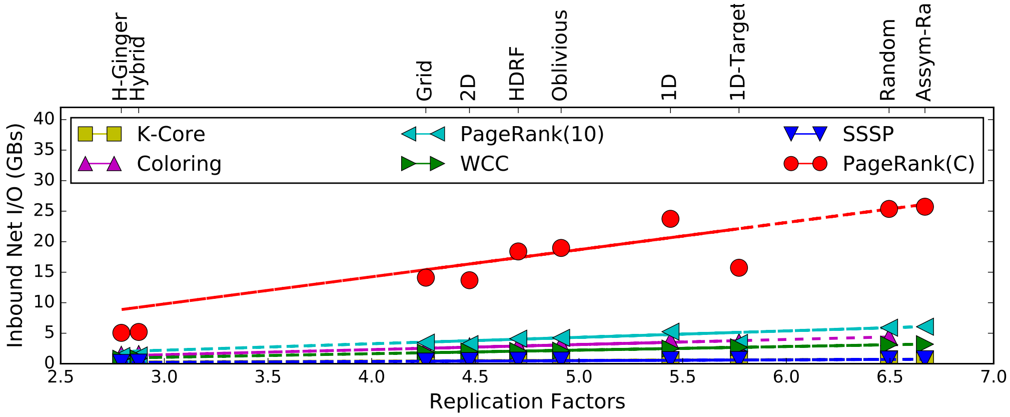


Figure 8.3: Incoming network IO vs. Replication Factor. (Local-9, PowerLyra, Twitter).

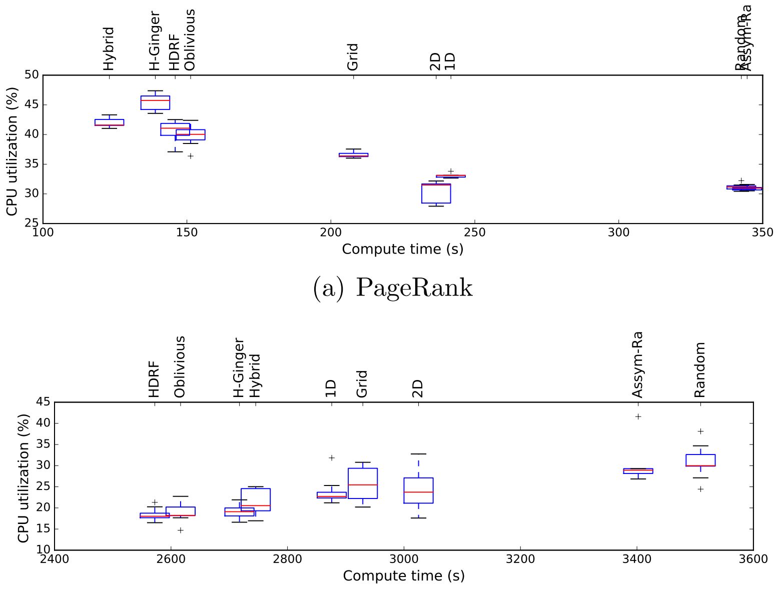
the same partition) yields even higher replication factors (Figure 8.1). Thus, we recommend avoiding both of these strategies while running PowerLyra.

8.2.3 Hybrid Engine Enhances 1D/2D Partitioning

PowerLyra's Hybrid Engine has low network traffic for natural applications when using a partitioning strategy that tends to co-locate "Gather-edges". PageRank, for example, gathers along only the in-edges, thus they are the gather-edges. Hybrid partitioning accounts for the gather direction of the application and accordingly co-locates gather-edges. On the other hand, 1D hashes edges by their source vertex and thus co-locates all the out-edges. Thus, the standard 1D implementation uses *more* network I/O. This is visible in Figure 8.3, where we interpolate (linear curve-fit) a line using the points; any performance point above the interpolation line is worse than expected according to its replication factor (this is true for 1D PageRank(C)).

To confirm this hypothesis, we implemented a variant of 1D, called **1D-Target**, that hashes edges by their target vertex, thus co-locating in-edges. As demonstrated in Figure 8.3, this strategy performs better and is *below* the interpolation line for PageRank.

Next, we observe that 2D is also able to benefit from the Hybrid engine (2D for PageRank is below the line in Figure 8.3). This is since 2D, in addition to imposing a $(2\sqrt{N}-1)$ upper bound on the overall replication factor, also imposes a tighter $\sqrt{N}$ upper bound on the number of machines a vertex's in-edges (or out-edges) can be assigned to. Having a smaller set of machines



(a) PageRank

(b) K-core

Figure 8.4: CPU utilization vs Compute phase duration (Local-9, UK-Web, PowerLyra-All). The box plots show min, 25th percentile, median, 75th percentile, and max but excluding outliers which can be seen as flier points.

on which the in-edges have to be placed increases the probablity that all in-edges will be co-located (especially for very low-degree vertices). Thus, we see 2D performing slightly better than the trend.

8.2.4 CPU Utilization Patterns

In Figure 8.4 we test the hypothesis of whether average CPU utilization is correlated with computation time. Here we see that the correlation to replication factor (and thus compute time) varies by application: increasing (Figure 8.4(b)) or decreasing (Figure 8.4(a)). We also note there are no clear correlations between load imbalance (spread of boxes' ranges in Figure 8.4) and compute time.

CHAPTER 9

GRAPHX: ALL STRATEGIES

Since GraphX's engine is significantly different in design from PowerGraph and PowerLyra, we implemented all (non-GraphX) strategies in GraphX and we compare them here.

9.1 Partitioning Strategies

We used Chen et. al's [9] implementation of Hybrid and Oblivious in GraphX as our baseline. On top of it, we implemented HDRF, Hybrid-Ginger and Grid to get all strategies on GraphX. In the process we also reported and fixed a bug in the implementation of Oblivious[1]. The bug caused incorrect clearing of the bit-sets used to keep track of the partitions a vertex has been replicated to – which essentially just made it random.

We also made Grid resilient to non-square number of machines by using a strategy similar to 2D's, whereby we construct a grid the size of the next largest square and then map it back down to the correct number of partitions.

9.2 Experimental Results

We ran our experiments on a local cluster of 9 machines. We ran all experiments to 25 iterations and additionally measured per-iteration times.

9.2.1 For Low-degree/Road-networks

For low-degree graphs (Fig 9.1), as seen in Section 7.4, we find that (Canonical) Random, for smaller number of iterations, delivers the fastest speed.

[1] Source code for our additional strategies can be found at `https://gitlab-beta.engr.illinois.edu/sverma11/graphx-extra-partitioners`

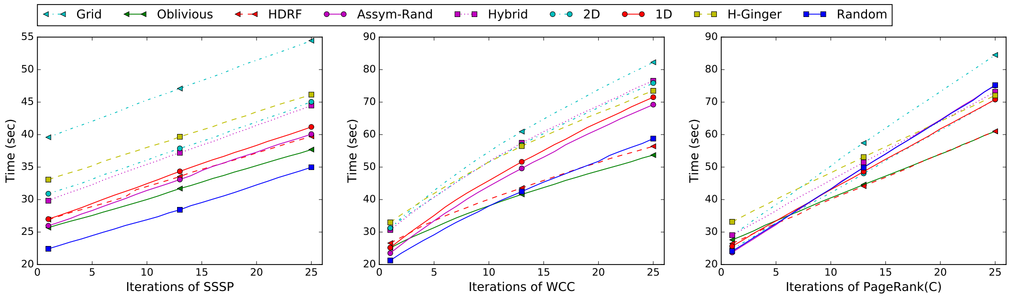


Figure 9.1: Total time takes at the end of each iteration. (GraphX-All, Road-net-CA, Local-9).

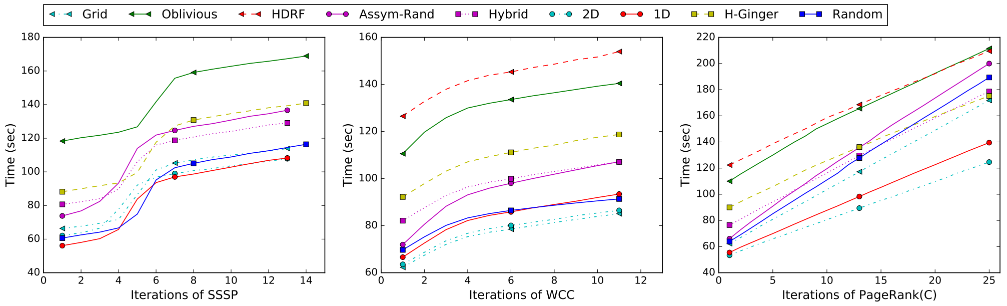


Figure 9.2: Total time takes at the end of each iteration. (GraphX-All, LiveJournal, Local-9).

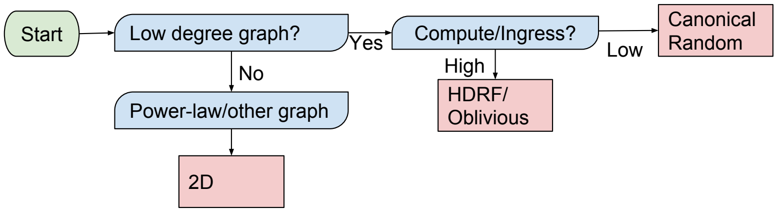


Figure 9.3: Decision Tree for Graphx-All.

As the number of iterations increases, HDRF and Oblivious catch up. The greedy strategies catch up faster when there are more active vertices. For example, the cross-over point between HDRF and Random appears earliest in PageRank (all vertices active), then in WCC (fewer active vertices) and finally in SSSP (fewest active vertices) it doesn't appear at all. That the greedy strategies generally have a lower average per-iteration time (lower slope in the charts), suggests that they yield higher quality partitions for low-degree graphs. This is similar to our conclusions for PowerGraph/Lyra. However, for short jobs on GraphX, it is preferable to pick Canonical Random and HDRF/Oblivious for long jobs.

9.2.2 For Power-law/Heavy-tailed Graphs

For Power-law-like graphs we see in Fig 9.2 that 2D is always the best or among the best strategies. This reconfirms our hypothesis from Section 7.4 that due to large number of partitions and presence of high-degree vertices, 2D's upper bound of $(2\sqrt{N}-1)$ on replication factor delivers higher quality partitions than other strategies. Grid, which has similar upper bounds on replication factor, usually follows 2D pretty closely in Fig 9.2. Because of 2D's fast partitioning speed, it is ideal for both long running and short-running jobs on such graphs.

9.2.3 Picking a Strategy

Putting together our results, we recommend that for low-degree graphs we use Canonical Random with short running jobs and HDRF/Oblivious with long running ones. For power-law graphs we prefer 2D regardless of job length. This can be summarized by the decision tree in Figure 9.3. Note that this

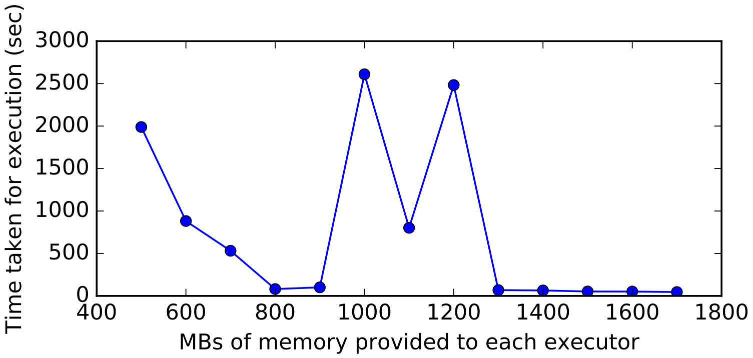


Figure 9.4: Affect of provided executor memory on the execution time (GraphX-All, Road-net-CA, Local-9).

is very similar to the original decision strategy for GraphX mentioned in Section 7.4

That the decision tree changes very little when the entire partitioning strategy set is implemented on both GraphX and PowerLyra implies that the partitioning strategies from other systems do not usually fare as well as the already-present strategies. As shown by our 1D-target experiment in Section 8.2.3 this is likely because the native strategies, unlike the migrated ones, have been optimized for their systems.

9.2.4 Memory Utilization Patterns in GraphX

Due to the uniqueness of the GraphX engine, memory is a critical resource. We analyze the effect of memory pressure on how GraphX assigns its partitions. To do this, we varied the "executor-memory" parameter and measured the resultant execution time. Fig 9.4 shows the results.

GraphX partition loading as follows. It incrementally adds edges to each partition. It first tries to co-locate partitions on a smaller number of machines (to reduce the cost of inter-partition communication). If this results in executors running out of memory, GraphX increases the number of machines to see if it fits (and so on). Because of this, the execution patterns fall into 3 cases:

- **Case 1**: The graph cannot fit on the entire cluster. (500MB in Fig 9.4).

- **Case 2**: The graph can fit on the cluster but not in a select few executors. (600MB – 1200MB).
- **Case 3**: The graph can fit in a select few (such as just 2) executors. (1300MB onwards).

In the first case, Spark tries to initially load the entire graph on just two executors. When they run out of memory, Spark tries to load the graph using the whole cluster. Spark may try several times to fit the job on the cluster by redistributing the partitions. After encountering too many out-of-memory errors, Spark finally fails the job. In Figure 9.4, this happens at the 500 MB point.

In the second case, Sparks' attempts to fit the graph by evenly distributing the partitions eventually succeed. But this is only after its initial attempt to load the graph in just two executors fails. Given that it can take an unknown number of attempts for Spark to fit the graph, it is hard to predict the execution time in this case. In Figure 9.4, this occurs in the 600 MB to 1200 MB range.

In the final case, the very first attempt to load the graph succeeds, this leads to a really fast execution time. In Figure 9.4, this happens after 1300 MB of memory. We also see that execution time decreases as more memory is provided. This is because when memory pressure is decreased, garbage collection overhead also reduces.

Therefore, even if the whole graph fits in memory (case 3), we recommend having some spare memory in order to 1) reduce the garbage collection overhead and 2) avoid the insidious "GC overhead limit exceeded" error.

CHAPTER 10

CONCLUSIONS

In this thesis we performed a thorough experimental evaluation and comparison of the partitioning strategies found in three leading distributed graph processing systems, namely PowerGraph, PowerLyra and GraphX.

For PowerGraph, we found that replication factor is a good indicator of partitioning quality as it is linearly correlated with network usage, computation time and memory utilization. We showed that HDRF/Oblivious strategies are ideal for low-degree graphs while Grid is ideal for heavy-tailed graphs. For power law-like graphs, job duration should be taken into account: Grid is better for short jobs due to its fast ingress, and HDRF/Oblivious are better for long jobs due to lower replication factors (this includes when partitions are reused across jobs). For PowerLyra, we need to additionally consider if the application is natural or not, as Hybrid strategies synergize well with natural applications. We also show that Random and Hybrid-Ginger should generally be avoided due to high replication factors and memory overheads, respectively. We present two decision trees to help users of these systems pick a partitioning strategy. For GraphX, Canonical Random should be used with low-degree graphs and 2D partitioning with power-law-like graphs.

When all techniques were implemented in PowerLyra, we found some strategies perform better if they have tighter integration with the underlying engine (also confirmed via our 1D-target variant). We also found that CPU utilization and load imbalance are not clearly correlated with performance.

Finally, when all the techniques were implemented in GraphX, we see that HDRF/Oblivious continue to deliver high quality partitions for low-degree graphs. Although Canonical Random is generally better for shorter running jobs, for heavy-tailed graphs 2D remains the partitioning strategy of choice.

REFERENCES

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: graph processing at Facebook-scale," *Proc. of VLDB Endowment*, 2015.

[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. of Symposium on Operating Systems Design and Implementation (OSDI).* USENIX, 2012.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. of Int'l. Conf. on Management of Data (SIGMOD).* ACM, 2010.

[4] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. of Symposium on Operating Systems Design and Implementation (OSDI).* USENIX, 2014.

[5] "Apache Giraph," http://giraph.apache.org/, last accessed 2016-04-18.

[6] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC." in *Proc. of Symposium on Operating Systems Design and Implementation (OSDI).* USENIX, 2012.

[7] I. Hoque and I. Gupta, "LFGraph: Simple and fast distributed graph analytics," in *Proc. of Conf. on Timely Results In Operating Systems (TRiOS).* ACM, 2013.

[8] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *Proc. of Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM).* ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/316188.316229 pp. 251–262.

[9] R. Chen, J. Shi, Y. Chen, and H. Chen, “Powerlyra: Differentiated graph computation and partitioning on skewed graphs,” in *Proc. of European Conf. on Computer Systems*, ser. (EuroSys). ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741970 pp. 1:1–1:15.

[10] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl, “All roads lead to Rome: Optimistic recovery for distributed iterative data processing.” in *Proc. of Int'l. Conf. on Information and Knowledge Management (CIKM).* ACM, 2013.

[11] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, “An experimental comparison of pregel-like graph processing systems,” in *Proc. of VLDB Endowment*, vol. 7. VLDB Endowment, Aug. 2014, pp. 1047–1058.

[12] S. Salihoglu and J. Widom, “GPS: A graph processing system.” in *Proc. of Int'l. Conf. on Scientific and Statistical Database Management (SSDBM).* ACM, 2013.

[13] R. Power and J. Li, “Piccolo: Building fast, distributed programs with partitioned tables.” in *Proc. of Symposium on Operating Systems Design and Implementation (OSDI).* USENIX, 2010.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “Pegasus: A peta-scale graph mining system implementation and observations,” in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2009.14 pp. 229–238.

[15] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, “Mizan: A system for dynamic load balancing in large-scale graph processing,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465369 pp. 169–182.

[16] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad: A timely dataflow system,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522738 pp. 439–455.

[17] F. Yang, J. Li, and J. Cheng, “Husky: Towards a more efficient and expressive distributed computing framework,” *Proc. VLDB Endow.*, vol. 9, no. 5, pp. 420–431, Jan. 2016. [Online]. Available: http://dx.doi.org/10.14778/2876473.2876477

[18] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442530 pp. 135–146.

[19] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *Proceedings of the 2015 Data Compression Conference*, ser. DCC '15. Washington, DC, USA: IEEE Computer Society, 2015. [Online]. Available: http://dx.doi.org/10.1109/DCC.2015.8 pp. 403–412.

[20] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu pp. 301–316.

[21] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[22] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions." in *Proc. of Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.

[23] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 410–424.

[24] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.

[25] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168846 pp. 85–98.

[26] “Neo4j Graph Database,” https://neo4j.com/product/, last accessed 2017-04-5.

[27] B. Shao, H. Wang, and Y. Li, “Trinity: A distributed graph engine on a memory cloud,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2463676.2467799 pp. 505–516.

[28] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer, “Weaver: A high-performance, transactional graph database based on refinable timestamps,” *Proceedings of the VLDB Endowment*, vol. 9, no. 11, 2016.

[29] R. Chen, J. Shi, B. Zang, and H. Guan, “Bipartite-oriented distributed graph partitioning for big learning,” in *Proc. of 5th Asia-Pacific Workshop on Systems*, ser. APSys ’14. ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2637166.2637236 pp. 14:1–14:7.

[30] N. Jain, G. Liao, and T. L. Willke, “Graphbuilder: scalable graph ETL framework,” in *First Int’l. Workshop on Graph Data Management Experiences and Systems, (GRADES)*, 2013. [Online]. Available: http://event.cwi.nl/grades2013/04-jain.pdf

[31] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997

[32] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, “Fennel: Streaming graph partitioning for massive scale graphs,” in *Proc. of 7th ACM Int’l. Conf. on Web Search and Data Mining (WSDM).* ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2556195.2556213 pp. 333–342.

[33] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, “A distributed algorithm for large-scale graph partitioning,” *ACM Trans. Auton. Adapt. Syst.*, vol. 10, no. 2, pp. 12:1–12:24, June 2015. [Online]. Available: http://doi.acm.org/10.1145/2714568

[34] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel, “Graph: Heterogeneity-aware graph computation with adaptive partitioning,” in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, June 2016, pp. 118–128.

[35] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John, “Data partitioning strategies for graph workloads on heterogeneous clusters,” in *Proc. of Int’l. Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. (SC ’15). ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807632 pp. 56:1–56:12.

[36] J. Huang and D. J. Abadi, “Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs,” *Proceedings of the VLDB Endowment*, vol. 9, no. 7, pp. 540–551, 2016.

[37] M. Pundir, M. Kumar, L. M. Leslie, I. Gupta, and R. H. Campbell, “Supporting on-demand elasticity in distributed graph processing,” in *Proc. of Int’l. Conf. on Cloud Engineering (IC2E).* IEEE, 2016, best Paper Award.

[38] T. Anwar, C. Liu, H. L. Vu, and C. Leckie, “Spatial partitioning of large urban road networks.” in *openproceedings.org*, 2014.

[39] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell, “Zorro: Zero-cost reactive failure recovery in distributed graph processing,” *Technical Report, IDEALS*, 2015. [Online]. Available: https://ideals.illinois.edu/handle/2142/75959

[40] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[41] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed GraphLab: A framework for machine learning and data mining in the cloud,” in *Proc. of VLDB Endowment.* VLDB Endowment, 2012.

[42] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, “GraphLab: A new parallel framework for machine learning,” in *Conf. on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[43] R. M. Karp, “Reducibility among combinatorial problems,” in *Proc. of a Symposium on the Complexity of Computer Computations*, 1972. [Online]. Available: http://www.cs.berkeley.edu/~luca/cs172/karp.pdf pp. 85–103.

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

[45] “Laboratory For Web Algorithms,” http://law.di.unimi.it/datasets.php, last Accessed 2016-04-18.

[46] "DIMACS Challenge 9 - Shortest Paths," http://www.dis.uniroma1.it/challenge9/, last Accessed 2016-05-30.

[47] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks," in *Proc. of Int'l. Conf. on World Wide Web (WWW).* ACM, 2011.

[48] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of Int'l. Conf. World Wide Web (WWW).* ACM, 2004.

[49] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. of Int'l. Conf. on World Wide Web (WWW).* ACM, 2010.

[50] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proc. of 24th ACM Int'l. on Conf. on Information and Knowledge Management (CIKM).* ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2806416.2806424 pp. 243–252.

[51] H. Halberstam and R. R. Laxton, "Perfect difference sets," *Proc. of Glasgow Mathematical Association*, vol. 6, pp. 177–184, July 1964. [Online]. Available: http://journals.cambridge.org/article_S2040618500034985

[52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of Conf. on Networked Systems Design and Implementation (NSDI).* USENIX, 2012.

APPENDIX A

OBLIVIOUS

Consider the task of placing the $i+1^{th}$ edge after having placed i edges. The objective function for Oblivious [2] reduces to:

$$\arg\min_k \Big[|A(v)| + |A(u)| \Big| A_i, A(e_{i+1}) = k \Big],$$

where $A(v)$ is the set of machines v is replicated on, A_i is the set of edge placements we have done so far, $A(e_{i+1})$ is where we will assign the $i+1^{th}$ edge (u,v). This devolves into a few simple cases.

Case 1: $A(v) \cap A(u) \neq \phi$. I.e. on at least one machine, replicas of u and v both are already present. The edge is placed at the least loaded machine in $A(v) \cap A(u)$.

Case 2: Only one of the vertices have been placed so far. So, without loss of generality: $A(v) = \phi$ and $A(u) \neq \phi$. The edge will be placed on the least loaded machine in $A(u)$.

Case 3: $A(v) = A(u) = \phi$. The edge will be placed on the least loaded machine.

Case 4: $A(u) \neq \phi$ and $A(v) \neq \phi$ but $A(u) \cap A(v) = \phi$. The edge will be placed on the least loaded machine in $A(u) \cup A(v)$.

Ties are broken randomly. In this context, *least loaded* refers to the machine which has been assigned the fewest edges.

APPENDIX B

HDRF

When processing edge (u, v) the partial degree counters (δ) of u and v are incremented. Then they are assigned a normalized value θ :

$$\theta(v) = \frac{\delta(v)}{\delta(u) + \delta(v)}$$

Each machine M is assigned a score C as follows:

$$C(u, v, M) = C_{REP}(u, v, M) + \lambda \times C_{BAL}(M)$$

$$C_{REP}(u, v, M) = g(u, M) + g(v, M)$$

$$g(v, M) = \begin{cases} 1 + (1 - \theta(v)) & if M \in A(v) \\ 0 & else \end{cases}$$

C_{BAL} is a score in $[0, 1)$ assigned to a machine on the basis of the number of edges assigned to it so far. A more loaded machine will have a lower C_{BAL}. The machine with the higher C score is selected.

Thus the λ parameter is used to tune the systems prioritization towards load-balance. When $\lambda \leq 1$ the balance parameter is used as a tie breaker. After that point balance importance rises in proportion to λ. In the PowerGraph implementation, λ is hardcoded to 1.