Data Partitioning Strategies for Graph Workloads on Heterogeneous Clusters

Michael LeBeane
mlebeane@utexas.edu

Shuang Song
songshuang1990@utexas.edu

Reena Panda
reena.panda@utexas.edu

Jee Ho Ryoo
jr45842@utexas.edu

Lizy K. John
ljohn@ece.utexas.edu

Department of Electrical and Computer Engineering
The University of Texas at Austin

ABSTRACT

Large scale graph analytics are an important class of problem in the modern data center. However, while data centers are trending towards a large number of heterogeneous processing nodes, graph analytics frameworks still operate under the assumption of uniform compute resources. In this paper, we develop heterogeneity-aware data ingress strategies for graph analytics workloads using the popular PowerGraph framework. We illustrate how simple estimates of relative node computational throughput can guide heterogeneity-aware data partitioning algorithms to provide balanced graph cutting decisions. Our work enhances five online data ingress strategies from a variety of sources to optimize application execution for throughput differences in heterogeneous data centers. The proposed partitioning algorithms improve the runtime of several popular machine learning and data mining applications by as much as a 65% and on average by 32% as compared to the default, balanced partitioning approaches.

CCS Concepts

•Computer systems organization → Cloud computing; Heterogeneous (hybrid) systems; •Information systems → *Data layout;* **•Computing methodologies** → *Distributed programming languages;* **•Mathematics of computing** → Graph algorithms;

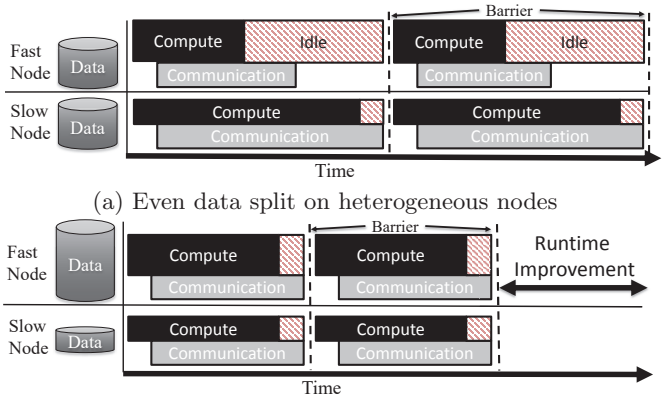
Keywords

Data Partitioning, Graph Algorithms, Heterogeneous, Data Center, Cloud Computing

1. INTRODUCTION

Large multi-node 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 [24], PowerGraph [13], Pregel [25], and Giraph [7]. These domain specific programming frameworks handle a large amount of functionality common to all graph algorithms behind the scenes, expressing an easy and tractable abstraction to the application programmer.

Programming frameworks, however, are not the only change induced by big data and cloud computing. To cheaply provide scale-out performance to users, data 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 *heterogenous 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



(a) Even data split on heterogeneous nodes

(b) Heterogeneity-aware data partitioning

Figure 1: Figure 1a illustrates a computational imbalance due to homogeneous data placement in a heterogeneous data center. The fast node is waiting at a barrier for a slower straggler node to finish computing. Figure 1b places data in accordance with a node's processing ability, optimizing CPU utilization and improving runtime.

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807632

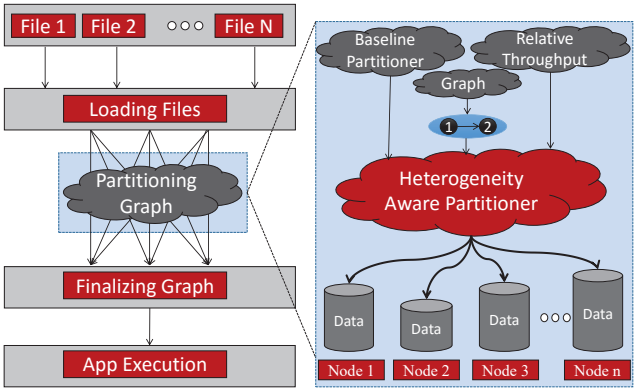


Figure 2: Overview of the heterogeneous static partitioning approach. Our approach modifies several popular data ingress partitioning strategies to accommodate for relative compute throughput differences in heterogeneous data centers.

needs of a variety of users and clients, and will provide different system configurations to target various performance and power points [9][12]. 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 [4] 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 (and most big data frameworks in general) operate under the assumption that all compute nodes are balanced in performance. This assumption leads to an imbalance of work distribution, causing some fast nodes to finish processing their chunk of the data sooner than slow nodes. The slow "stragglers" [31] decrease the overall cluster throughput whenever synchronization is required, as shown in Figure 1a. For graph analytics applications which require synchronization between iterations of an algorithm, straggler nodes result in low average processor occupancy and huge processing inefficiencies. Our own 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. While dynamic load balancing techniques may help alleviate some of these issues, the graph analytics frameworks that we surveyed [24][13][25] do not posses such mechanisms. Even in frameworks which support dynamic load balancing, big data workloads have been shown to exhibit massive load balancing overheads that are rarely amortized over the execution time of application programs [6]. Ideally, data partitioning should correctly skew the data in accordance with the processing capability of the nodes so that all machines reach the barrier at approximately the same time, as illustrated in Figure 1b.

This paper enhances several state of the art static partitioning algorithms to account for the affects of heterogeneous data centers on graph analytics. Figure 2 shows an overview of our approach, which is implemented in the popular PowerGraph [13] distributed graph processing system. We implement our partitioning strategies as pluggable edge/vertex cut methods in PowerGraph's streaming graph partitioner. These strategies are powered by a metric that describes the desired data load on each node, which we call the *skew factor*. Our work describes a number of ways that the skew factor of a cluster can be estimated and used to guide partitioning decisions. Specifically, our contributions to the state of the art include the following:

1. We offer a number of graph partitioning strategies to improve data-ingress on heterogeneous clusters. Our algorithms are extensions of several state of the art online graph partitioning techniques from the literature [13][8][16][29]. While other research attempt to utilize dynamic load balancing techniques to account for data center heterogeneity [18] [27], we show that a simple skewed partitioning performs very well without the overheads associated with dynamic load balancing techniques.

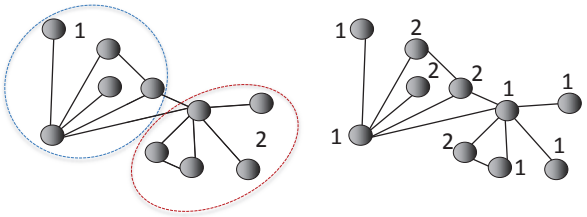
2. We illustrate a number of simple heterogeneity estimates based on relative intranode throughput. These estimates are used to develop the skew factor and to guide the graph splitting decisions offered by our partitioning algorithms.

3. We provide an in-depth performance evaluation on heterogeneous clusters built from the Amazon EC2 virtualized cloud environment. We compare standard online data partitioning algorithms to the heterogeneity-aware versions. The heterogeneity-aware partitioners improve application execution time by 32% averaged over a variety of real-world data sets and algorithms. In addition, we illustrate how our partitioning algorithms scale in a cluster of up to 48 heterogeneous nodes of multiple configurations.

This paper is organized into several sections that present and evaluate the heterogeneous partitioning strategies. Section 2 overviews important background information for our work. Section 3 explains how our algorithms modify and enhance current homogeneous online streaming partitioners. Section 4 evaluates how the different partitioning strategies perform on a small heterogeneous cluster and at scale. Section 5 surveys the related work and section 6 concludes the paper.

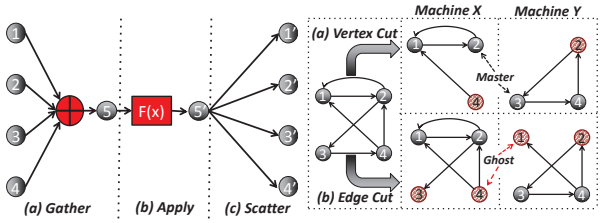
2. BACKGROUND

Our work is implemented on the PowerGraph graph processing framework. Since our paper introduces a number of partitioning strategies for heterogenous nodes, it is important to understand graph partitioning concepts used in most graph analytics frameworks. This section will describe the difference between offline vs online graph partitioning, the gather/apply/scatter (GAS) computational model, and the difference between edge and vertex cuts. While these explanations will be framed in the context of PowerGraph, these concepts should extend to most graph processing frameworks.



(a) Offline partitioning (b) Online partitioning

Figure 3: Offline vs streaming graph partitioning.



(a) GAS model (b) Edge/vertex cuts

Figure 4: Gather/Apply/Scatter execution model and edge vs vertex cuts.

2.1 Offline vs Online Partitioning

Graphs can be partitioned using what we will refer to as offline or online techniques. Figure 3 illustrates the differences between offline and online graph 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 [17] 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 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.

Skewing graph partitions based on cluster heterogeneity can be performed for both online and offline graph cuts. Since online cuts are more popular in data analytics frameworks and are what have been implemented in the PowerGraph engine, the work in this paper will focus on online partitioning algorithms.

2.2 PowerGraph Computation Model

PowerGraph employs the common "think like a vertex" idiom by forcing users to express their algorithms in terms of *vertex programs* that adhere to the gather/apply/scatter (GAS) computational 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 4a illustrates the GAS computational paradigm on a set of vertices. During the gather phase of a vertex, the PowerGraph engine 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. The scatter stage makes the new value of v visible to neighboring vertices and edges during the next iteration of a GAS operation. This sequence of events is repeated until a user defined stopping condition (usually an iteration cap or convergence criteria) has been reached. The GAS programming model has undergone a number of optimizations and has an extremely efficient and streamlined implementation in the latest version of PowerGraph.

To run GAS vertex programs, PowerGraph employs a computational engine that is either synchronous or asynchronous. The synchronous engine enforces strict barriers between the steps of a vertex program. Alternatively, the asynchronous engine allows for nodes to run out of sync, but enforces some data consistency through fine grained-locking. This paper exclusively uses the synchronous engine, since a number of our test applications require the strict guarantees of full barrier synchronization to ensure correct operation.

2.3 Edge vs Vertex Cuts

Whenever a graph is cut and split between two 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. In general, a graph partitioning algorithm has the choice of cutting a graph according to its edges or its vertices, as illustrated in Figure 4b. 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 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:

$$P(d) \propto d^{-\alpha} \tag{1}$$

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

Table 1: Amazon c4-type virtual node configurations and skew factors.

Amazon EC2 Configuration				Skew Factors	
Name	**HW Threads**	**Memory**	**Network [3]**	**Thread Based**	**Memory Based**
c4.xlarge	4	7.5GB	100 Mbps to 1.86 Gbps	1	1
c4.2xlarge	8	15GB	100 Mbps to 1.86 Gbps	3	2
c4.4xlarge	16	30GB	100 Mbps to 1.86 Gbps	7	4
c4.8xlarge	36	60GB	up to 8.86Gbps	17	8

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 GAS sub-step.

In this paper, we apply most of our heterogeneous data skewing techniques to the baseline partitioning strategies in PowerGraph, which perform vertex cuts exclusively. However, we also present some strategies from PowerLyra [8], which performs both edge and vertex cuts depending on the degree of each vertex.

3. HETEROGENEOUS PARTITIONING ALGORITHMS

In this section, we describe how we modified five distinct online graph partitioning algorithms to more ideally map to heterogenous clusters. The modifications to each partitioning strategy keeps the flavor and intuition that was present in the original algorithm, while still skewing the data in accordance with the relative throughput of each node. While the concept of skewing data partitioning for heterogeneity can apply to any method, we choose to modify existing approaches to allow for comparison to a solid baseline. Before we present heterogeneity aware algorithms, however, we must first have some formal method of expressing the differences in performance between each node in the cluster.

3.1 Estimating Heterogeneity in a Cluster

The main idea in data partitioning for heterogeneous nodes is simple: we want to divide the input data set into shards such that each node receives an amount of data in accordance with a metric. We define the desired data partitioning ratio between nodes to be the *skew factor* of the cluster. For the purposes of this paper, the *skew factor* is always written relative to other nodes, with the least powerful node receiving the value of 1.

We have provided a few simple and tractable methods of representing the skew factor of a heterogeneous cluster. To ground our discussion, we will describe how the skew factor is calculated for a number of Amazon EC2 configurations, which we will later use for our experimental section. Table 1 shows the node configurations and the skew factor that would be used for each approach.

Thread Based Skew Factor: For smaller data sets where the graph can easily fit into the memory of any single node, a reasonable proxy of performance between two machines can be derived by looking at the relative performance of the CPUs. For our example Amazon cluster, the CPU type is the same, and only the number of hardware threads assigned to each virtual node vary across configurations. Therefore, we define the thread based skew factor to estimate relative throughput differences between nodes using the number of worker threads. Table 1 shows the thread based skew factor for our sample machines. We use the number of logical cores reserved for computation in PowerGraph ($num\_logical\_cpus - 2$) to calculate relative throughput. Two logical cores are reserved for communication and are not used to compute the skew factor.

Memory Based Skew Factor: Many graphs are too large to fit in the memory of a single node in the cluster. For graphs of these types, it is critical to assign a skew factor based on the size of main memory on the nodes to avoid page faults during processing. The PowerGraph framework in particular is especially vulnerable to memory imbalance, since it will not run if the entire graph and all related data structures do not fit in main memory. We use a memory based skew factor to allow cluster with some small memory nodes to participate in graph calculations without facing memory constraints. This memory based skew factor is computed as the ratio of node DRAM capacities between all of the nodes and is shown for the example EC2 cluster in Table 1.

Profiling Based Skew Factor: In addition to these two simple metrics, we have investigated using lightweight profiling to develop our skew factors. A true estimate of a node's processing throughput is inherently tied to the characteristics of the workload. Different workloads stress different parts of a compute node, so a machine that can process workload A 3x faster than another machine may not process workload B 3x faster. Therefore, we investigated a simple profiling based skew factor that implicitly factors in the intranode bottlenecks when processing these algorithms. However, we found that profiling based skew factors resulted in very similar results as the thread based skew factor and could not justify the added complexity. We leave the development of an efficient profiling based skew factor as future work.

While there are certainly more complex methods of calculating the skew factor, we show in Section 4 that these rough estimates can provide large performance benefits for the heterogeneity-aware partitioning algorithms.

3.2 Problem Formulation

Now that we have a method of calculating the skew factor for a given cluster, we can formalize the requirements of an online graph partitioning algorithm. We will formulate it as a vertex cut problem, as this applies to the majority of our algorithms, but a similar formulation and constraints can be derived for an edge cut objective. 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. We also formally

define Sk such that $Sk(p) = \frac{skewfactor(p)}{sum(skewfactor)}$ where the skew factor is calculated as discussed in Section 3.1. 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[13]:

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)| \quad (2)$$

$$\text{s.t. } \forall p \in P, \mathbf{abs}(|\{e \in E : A(e) = p\}| - Sk(p) * |E|) < \lambda \quad (3)$$

where we define λ 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 2 attempts to place edges for a given vertex v on a small number of machines, minimizing the communication and memory overhead. Equation 3, 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.

3.3 Partitioning Algorithms

In this section, we illustrate how five popular streaming graph cut algorithms from the literature [13][8][16][29] can be modified to support optimized heterogeneous data placement using the skew factor. The five streaming algorithms 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 or memory based approach as illustrated in Section 3.1, or from a user defined constraint. For our explanation of the algorithms, we will use Sk as defined in Section 3.2. Additionally, we will use and define iSk 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 function is trivially constructed from the skew factor or Sk.

3.3.1 Skewed Random Hash

The random streaming vertex cut (Random) was originally proposed in the PowerGraph [13] 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 1 describes the formulation of Random for a heterogeneous environment.

Algorithm 1 Skewed Random Hash Cut

```
1: procedure SRandom
2:     for e ∈ E do
3:         s ← Src(e)                          ▷ s is source vertex of edge e
4:         d ← Dest(e)                         ▷ d is destination vertex of edge e
5:         p ← iSk[HashE(s,d) / maxHash]       ▷ compute skewed hash
6:         e.owner ← p                         ▷ assign owning node for e
7:     end for
8: end procedure
```

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.

3.3.2 Skewed Greedy

The greedy vertex cut algorithm (Greedy) was originally proposed in PowerGraph [13] 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. Greedy comes in two primary forms: oblivious and coordinated, depending on whether the data structures are shared between machines or private. However, the algorithm otherwise functions exactly the same in both variations. Algorithm 2 describes the formulation of Greedy for a heterogeneous environment.

Algorithm 2 Skewed Greedy Edge Cut

```
1: procedure SGreedy
2:     for e ∈ E do
3:         s ← Src(e)
4:         d ← Dest(e)
5:         for p ∈ P do
6:             srcPresence ← (Edges(s, p) > 0)
7:             destPresence ← (Edges(d, p) > 0)
8:             bal ← Balance(Sk, p)
9:             score[p] ← bal + srcPresence + destPresence
10:        end for
11:        e.owner ← 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 pseudocode, 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 3.2. Skewed Greedy cut uses the following modified heuristics from [13], 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 Skewed Random does. Therefore, we have implemented an optional override (not shown in the pseudocode) which switches to a random hash-based approach if the partitions exceed a user defined balance threshold.

3.3.3 Skewed Grid Constrained

The grid constrained algorithm (Grid) [16] attempts to place an upper bound on memory and network communication by constraining the amount of mirrors that can be

generated. This is accomplished by forming a matrix from the requested number of nodes, and hashing each vertex to an entry in the matrix. The valid list of candidates that an edge can be assigned to is the intersection of the processors in the rows and columns of the source and destination vertex. If the number of nodes in the cluster is N, then Grid places a $2\sqrt{N}-1$ upper bound on the replication factor. The heterogeneous version of Grid is defined in Algorithm 3.

Algorithm 3 Skewed Grid Hash Cut

```
1: procedure SGrid
2:     Grid ← makeGrid(P)
3:     SGrid ← makeSkewedGrid(Sk, Grid)
4:     for e ∈ E do
5:         s ← Src(e)
6:         d ← Dest(e)
7:         srcSet ← SGrid[HashV(s)/maxHash]
8:         destSet ← SGrid[HashV(d)/maxHash]
9:         cand ← srcSet ∩ destSet
10:        for p ∈ cand do
11:            score[p] ← Balance(Sk, p)
12:        end for
13:        e.owner ← argmax_p(score)
14:    end for
15: end procedure
```

Essentially, the Skewed Grid keeps the $2\sqrt{N}-1$ upper bound of the original Grid algorithm while accounting for heterogeneity. The matrix creation step, *makeGrid*, is kept from the original Grid algorithm. However, we create a new grid based on the old one by skewing the probability of selecting shard i based on the relative weights of its rows and columns. The candidate processor list is still formed from the intersection of the selected shards' rows and columns. Finally, the owner of an edge is selected from the candidate processor list using the *Balance*() function defined for greedy cut. Like the Skewed Greedy above, this heuristic is vulnerable to load balancing issues depending on the graph. We have chosen not to enforce strict load balancing constraints in Skewed Grid so that we can still enforce the upper bound of the replication factor found in the standard grid partitioning algorithm.

3.3.4 Skewed Hybrid Cut

The Hybrid cut algorithm [8] 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 methodology. Algorithm 4 describes the heterogeneous variant of Hybrid.

Skewed Hybrid modifies both phases of Hybrid cut assignment in a similar manner as Skewed 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.

Algorithm 4 Skewed Hybrid Cut

```
1: procedure SHybrid
2:     for e ∈ E do
3:         d ← Dest(e)
4:         p ← iSk[HashV(d)/maxHash]
5:         e.owner ← p
6:     end for
7:     for v ∈ V do
8:         if inDegree(v) > Threshold then
9:             for e ∈ Edges(v) do
10:                s ← Src(e)
11:                p ← iSk[HashV(s)/maxHash]
12:                e.owner ← p
13:            end for
14:        end if
15:    end for
16: end procedure
```

3.3.5 Skewed Ginger

The Ginger cut partitioning algorithm [8] is an extension of Hybrid cut enhanced with a locality heuristic called Fennel[29]. 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 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 $c(v,p) = |N(v) \cap V_p| - b(p)$, where $N(v)$ denotes the set of neighboring vertices along the in-edges of v. The first term, $|N(v) \cap V_p|$ 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: $\frac{1}{2}(|V_p| + \frac{|V|}{|E|} * |E_p|)$. Algorithm 5 describes the formulation of Ginger cut for a heterogeneous environment.

Algorithm 5 Skewed Ginger Cut

```
1: procedure SGinger
2:     for e ∈ E do
3:         d ← Dest(e)
4:         p ← iSk[HashV(d)/maxHash]
5:         e.owner ← p
6:     end for
7:     for v ∈ V do
8:         if inDegree(v) > Threshold then
9:             for e ∈ Edges(v) do
10:                s ← Src(e)
11:                p ← iSk[HashV(s)/maxHash]
12:                e.owner ← p
13:            end for
14:        else
15:            for p ∈ P do
16:                V_p ← Verts(p)
17:                cost[p] ← |N(v) ∩ V_p| − (1 − Sk[p]) * b(p)
18:            end for
19:            e.owner ← argmax_p(cost)
20:        end if
21:    end for
22: end procedure
```

Table 2: Graph and Matrix Data Sets used in evalution

Name	Vertices	Edges	Size (Uncompressed)	Type	Algorithms
amazon[22]	403,394	3,387,388	46MB	Directed Graph	PR,CC,TC
citation[22]	3,774,768	16,518,948	268MB	Directed Graph	PR,CC,TC
netflix[5]	NA	NA	100MB	Sparse Matrix	ALS,SGD
road-map[22]	1,379,917	1,921,660	84MB	Undirected Graph	PR,CC,TC
social-network[22]	4,847,571	68,993,773	1.1GB	Directed Graph	PR,CC,TC
twitter[20]	41,000,000	1,400,000,000	25GB	Directed Graph	PR,CC,TC
wiki[22]	2,394,385	5,021,410	64MB	Directed Graph	PR,CC,TC

Our Skewed 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 Skewed Greedy cut algorithm, this algorithm employs a heuristic which can generate partitions that slightly deviate from the requested skew factor. Therefore we allow for an optional override of the Fennel heuristic step (not shown in the algorithm) if the nodes become too imbalanced, changing Skewed Ginger back into the Skewed Hybrid cut, which guarantees near perfect load balancing.

4. EVALUATION

This section evaluates the effectiveness of our heterogeneous partitioning algorithms on a wide variety of data sets, applications, and cluster configurations. For our studies we have selected a number of different data sets of various types as shown in Table 2. The data sets range in size from a few million edges (amazon,road-map,wiki) to over a billion (twitter). While most of our data sets are traditional directed and undirected graphs, we also include a user ratings matrix data set (netflix) to evaluate recommender engine algorithms.

To accompany these data sets, we have also selected five common applications frequently leveraged in a machine learning and data mining (MLDM) environment. These algorithms are briefly described as follows:

PageRank (PR): The PageRank algorithm [26] attempts to increase the relative rating of a node based on the weights of all the nodes it is connected to. It's main use is in ranking the importance of web pages on the internet and is defined as follows:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v)}{L(v)} \quad (4)$$

This equation states that the PageRank for a page u is dependent on the PageRank 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.

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. In PowerGraph, 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, PowerGraph's triangle count implementation [28] 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.

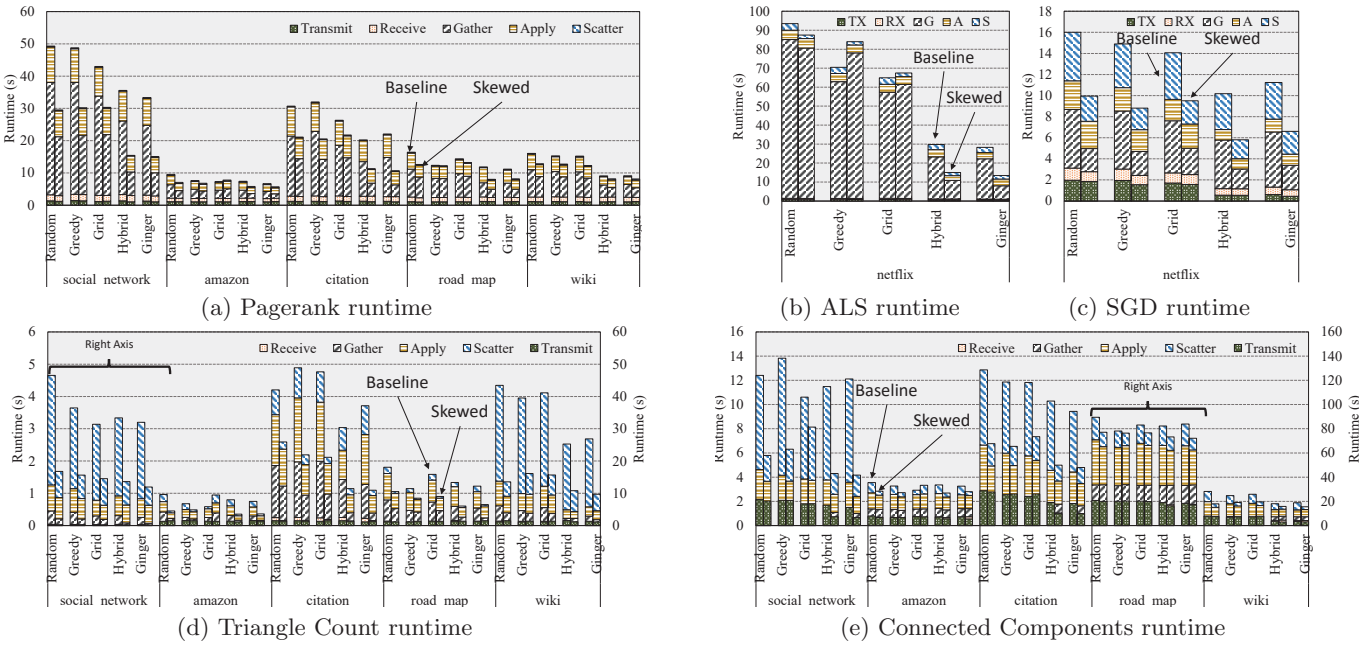
Stochastic Gradient Descent (SGD): The stochastic gradient descent algorithm [19] is a popular technique used to optimize an objective function. In PowerGraph, it is used to construct recommender engines by optimizing the squared error function derived from factoring the ratings matrix r_{ui} into p_u and q_i such that $r_{ui} = p_u * q_i$. SGD optimizes the squared error function by iteratively computing the prediction error from an initial estimate of p_u and q_i, and then modifies the parameters in the opposite direction of the gradient.

Alternating Least Squares (ALS): The alternating least squares algorithm [32] is also implemented by PowerGraph to construct recommender engines using matrix factorization using the same problem formulation as *SGD*. To minimize the squared error objective function, ALS rotates between fixing p_u and q_i and solving for the other matrix. Since either p_u or q_i is always fixed, the problem is quadratic and can be solved efficiently.

4.1 Fixed Cluster

In this section, we focus on evaluating the performance of our heterogeneous partitioning algorithms on a small fixed cluster of four heterogeneous nodes. For this study we use one node each from c4.xlarge, c4.2xlarge, c4.4xlarge, and c4.8xlarge. The performance characteristics of these nodes were described previously in in Table 1. All of the nodes are based on an Intel 2.9Ghz E5-2666 v3 processor. For these studies, we used the thread based skew factor since the graphs are small enough that running out of memory on any single node is not a concern.

Figure 5 shows the runtime of each heterogeneity-aware partitioning strategy and the unmodified version when applied to several algorithms 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 com-



(a) Pagerank runtime

(b) ALS runtime

(c) SGD runtime

(d) Triangle Count runtime

(e) Connected Components runtime

Figure 5: Runtime for algorithms and data sets under skewed and non-skewed partitioning constraints. The heterogenity-aware algorithms decrease application runtime by as much as 65%, and on average 32% when compared to the baseline partitioners.

pute threads are putting/getting data into its RX/TX buffers for the network threads.

The PageRank algorithm illustrated in Figure 5a shows an average improvement in runtime of 25% among all workloads and data sets, with the Hybrid variants demonstrating the best performance. Note that the PageRank algorithm as implemented in PowerGraph does not have a scatter phase on the out-edges. ALS in Figure 5b, which performs most of its computation in the gather stage, shows an improvement of 17% over the baseline methods. The Skewed Grid and Skewed Greedy algorithms actually performs slightly worse for this application and data set. SGD in Figure 5c illustrates an improvement of 39% on the netflix data set. Triangle count, presented in Figure 5d, has an average improvement in runtime of 48%. Triangle count displays a strong amount of data dependent performance, with the social network graph taking much longer than the others. Social network is a power-law graph with a small number of extremely high degree vertices, which stresses the triangle counting algorithm. Finally, connected components (Figure 5e) shows an average improvement in runtime of 25% over all benchmarks. The road map data set takes significantly longer to process due to the unique nature of road topologies, which differs significantly from the other graphs. Overall, the skewed partitioning algorithms net an improvement of 32% over their respective baseline across all algorithms and data sets.

To obtain the optimal benefit from heterogeneity, it is important that the skewed partitioning strategies achieve the desired load balance as measured or calculated in Section 3. However, many of these strategies are heuristics and do not guarantee a perfect data skew. Therefore it is important to study how the choice of partitioning algorithm and data set can affect the load balance of our cluster. Figure 6 shows the

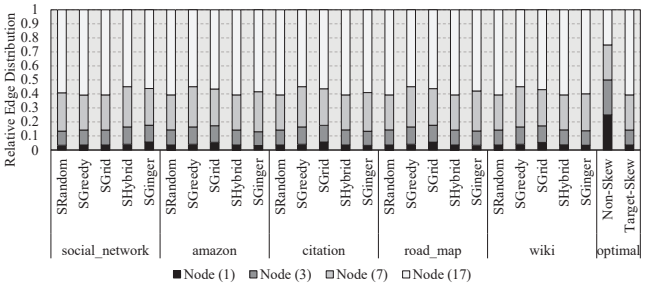


Figure 6: Relative distribution of edges to nodes for graphs and matrix data sets. Edge distribution defines the workload balance for the application programs.

balance of edge distributions among different data sets and partitioning strategies when compared to the targeted skew factor and the original homogeneous split approaches. The two bars on the right side of the graph illustrate the original homogeneous load balance objective and the skewed 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 the targeted skew. 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. We also see from the graph that 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 target data skew factors.

In addition to proper load balancing, reducing the network communication between the partitions is a key factor

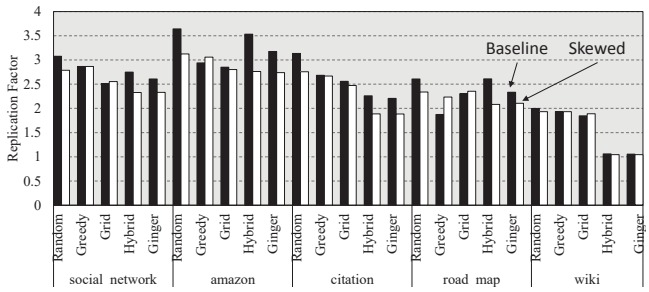


Figure 7: Replication factor for graphs and matrix data sets. A lower replication factor decreases the amount of network traffic and memory utilization in the cluster.

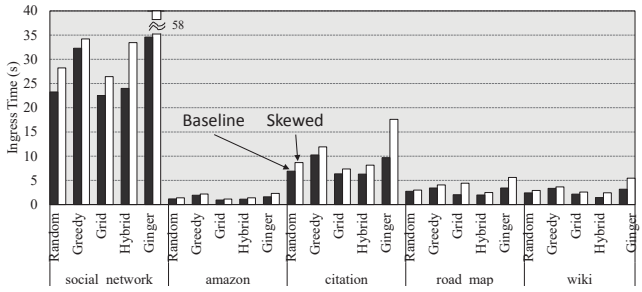


Figure 8: Overhead of the ingress partitioning techniques. The overhead for most heterogeneity-aware algorithms is approximately 20% compared to the baseline, which will amortize over the execution of most non-trivial applications.

in improving performance, especially when the algorithm has a small computation to communication ratio. For graph workloads, network communication overheads are directly correlated to the number of mirrors that exist on machines other than the master. The average number of mirrors can be expressed as the *replication factor*, where a replication factor of 1 corresponds to no communication, and a replication factor of p corresponds to an all-to-all broadcast for every time step for each edge or vertex operation. Figure 7 shows the replication factor for each baseline algorithm and the modified heterogeneous version over several different data sets. Generally, Random has the highest replication factor due to its complete disregard for locality and topology when placing edges. Hybrid or Ginger cut is generally the best performing on most graphs, depending on the proportion of high degree vertices to low degree vertices. In regards to the heterogeneity-aware partitioners, most strategies illustrate an improvement in replication factor as compared to the baseline. This is a fortunate side effect of skewing the data partitioning across nodes. A large amount of skew in a cluster pushes more data on a smaller number of nodes than a homogeneous partitioning strategy, leading to a smaller number of mirrors and a generally smaller replication factor. Skewed Grid does not perform appreciably better when skewing the data set, since the primary concern of the baseline version of Grid is ensuring an upper bound on the number of mirrors.

One final factor to consider is the amount of time it takes to perform the partitioning step of the algorithm. Higher quality partitions usually take more time than naive algorithms. Therefore it is important to investigate the impact

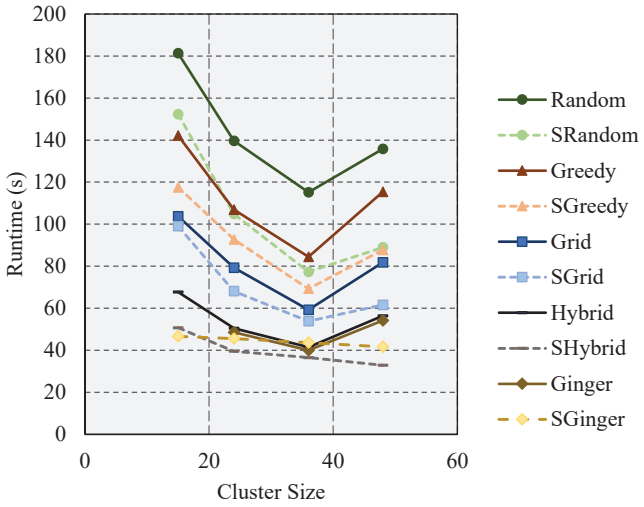


Figure 9: Scale-out runtime of PageRank on Twitter graph. Heterogeneity aware algorithms decrease runtime by 18% on average.

of the heterogeneous scheduling decisions on the data ingress phase. Figure 8 shows how data ingress is affected by heterogeneity across data sets. In general, Random and Grid based 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 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 data skew approach will exceed the extra computation needed to load the data.

4.2 Scale Out

While the previous section shows large benefits in runtime for relatively small to medium sized graphs, the data sets and accompanying data structures generated by PowerGraph are small enough to fit within the memory of one node. We would like to show that the benefits heterogeneous partitioning offers for our small test cluster continues for large data sets and scale-out clusters. For these studies, we will be using the Twitter [20] graph from Table 2 which contains over 1.4 billion edges. We will vary our node configuration from 15 to 48 nodes using nodes of the type c4.2xlarge, c4.4xlarge, and c4.8xlarge. All the configurations are divisible by three, so there are an even number of each type of compute node in every cluster configuration. Figure 9 shows the results from the scale-out experiments on the twitter data set using the PageRank algorithm. Generally,

Table 3: Scale out configurations (c4 type)

Config Name	c4.2xlarge	c4.4xlarge	c4.8xlarge
Config 1	12	8	4
Config 2	8	8	8
Config 3	4	8	12
Config 4	3	5	16

the hybrid and ginger algorithms perform the best, followed by grid, greedy, and random. At a size of 48 nodes, there is an inflection point and performance begins to worsen for all nodes. Between 36 and 48 nodes, the added communication imposed by scaling out overcomes the benefit of additional compute units. For all partitioning approaches, the skewed version performs better than the unskewed approaches. Overall, the skewed approaches achieve a 18% decrease in runtime across all cluster sizes when compared to the baseline versions. We were not able to include the baseline Ginger partitioning scheme in our scale-out experiment for a 15 node cluster, since it has a very large memory overhead for auxiliary data structures that exceeds the memory capacity of our nodes for the smallest split. However, Skewed Ginger works on correctly on all node configurations due to the use of a memory based skew factor, which optimizes the placement of the graph according to the memory requirements of each node.

The previous experiments all possessed an even number of each heterogeneous node type in the cluster. Heterogeneity-aware partitioning, however, is equally applicable to clusters built from any ratio of nodes. Figure 10 shows the percentage performance improvement of the skewed vs. unskewed partitioning algorithms on a 24 node cluster using the configurations listed in Table 3. The skewing algorithms are able to provide a performance improvement over the baseline versions all of the algorithms and data sets, with the relative benefit of the skewing algorithms increasing as the cluster is scaled in proportion to larger nodes. Config 4, which contains the most amount of powerful c4.8xlarge nodes, sees the most performance improvement across most of the workloads. This phenomena makes good intuitive sense, as the added benefit of correctly partitioning data in the presence of a large number of powerful nodes is more than correctly skewing across the same number of weaker nodes. While all the algorithms perform well, Random illustrates the greatest percentage improvement, due to the original version's poor performance.

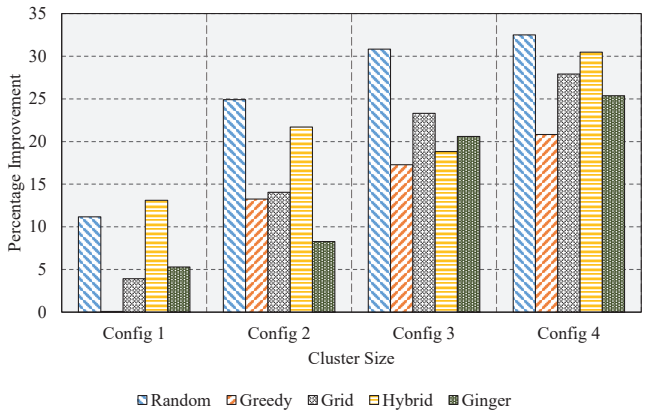


Figure 10: Skew vs. unskew percentage improvement. The improvement from the heterogeneity-aware algorithms increases as more powerful nodes are introduced into the cluster.

5. RELATED WORK

Our work is implemented on the PowerGraph[13] platform with PowerLyra [8] partitioning algorithm extensions. However, there are many other graph analytics frameworks available. Variants and extensions of GraphLab include the single node graph processing frameworks[24][21] and distributed frameworks[23]. Other popular non-GraphLab systems that are commonly used for graph applications include Pregel[25], Spark [30], and Giraph [7]. The various strengths and weakness of these and other similar graph processing platforms have been thoroughly explored in the prior work [15][14].

A few papers attempt to extend the insights obtained in these graph processing frameworks to specifically tackle data center heterogeneity for graph workloads. Zuhair[18] et al. target heterogeneity in their Pregel-like system called Mizan. Mizan performs runtime monitoring and attempts to perform vertex migration to balance computation and communication overheads. A similar technique to leverage the potential of node-level heterogeneity is performed by Semih [27] et. al in their GPS (graph processing system) framework, which also uses dynamic load balancing in a Pregel-like graph abstraction to support heterogeneity. To the best of our knowledge, no currently released graph processing framework specifically supports heterogeneous placement of data during ingress.

A number of non-graph specific processing frameworks have attempted to tackle the issue of data center heterogeneity, either explicitly or indirectly through the use of general purpose load balancing techniques. A significant number of these approaches have focused on the MapReduce [10] computational model. One of the earliest works in this area was LATE [31], a MapReduce optimization strategy that focuses on mitigating the affects of stragglers on data center performance. Zacharia[11] et al. propose the MARLA MapReduce framework for improving the performance of Hadoop[2] systems in a heterogeneous environment. MARLA's dynamic load balancing attempts to hide heterogeneity and load imbalance from the user while maintaining the reliability and useability features of regular Hadoop. Faraz[6] et al. propose Tarazu, another framework that targets heterogeneous clusters with MapReduce. Their work proposes a number of MapReduce specific optimizations, including communication aware scheduling of tasks and load balancing techniques. Despite operating on large data sets on heterogeneous clusters, these works do not overlap with our own due to our graph centric approach.

6. CONCLUSION

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, work imbalance, 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, we have developed a number of graph partitioning strategies that attempt to account for

heterogeneity in the data ingress phase of the popular PowerGraph framework. We show that simple estimations of intranode throughput, which we call the *skew factor*, can drive huge performance wins using heterogeneity-aware partitioning strategies. We have modified 5 existing data ingress strategies to take advantage of the skew factor to optimize application execution.

We illustrate that our graph ingress strategies reduce application runtime by as much as 64% with an average of 32% on a small 4-node heterogeneous cluster across a variety of applications and data sets. These partitioning strategies also reduce the replication factor while adhering to the load balancing constraints imposed by the skew factor estimation. Additionally, we show that our strategies operate at scale on a 48 node cluster, achieving an average 18% improvement in runtime. Finally, we investigate the impact of different heterogeneity mixes within a cluster, and achieve a runtime reduction of as much as 32% on PageRank when using clusters skewed towards high performance nodes.

7. ACKNOWLEDGMENTS

This work was supported in part by Semiconductor Research Corporation Task ID 2504, National Science Foundation grants 1337393 and 1117895, Oracle, Huawei and AMD. The authors would also like to thank Amazon for their donation of the EC2 compute resources used in this work. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, or any other sponsors.

8. REFERENCES

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

[2] Apache hadoop. https://hadoop.apache.org/. Accessed: 08-11-2015.

[3] Ec2 network estimations. http://www.aerospike.com/blog/boosting-amazon-ec2-network-for-high-throughput. Accessed: 04-16-2015.

[4] Intel xeon phi coprocessor. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html. Accessed: 04-16-2015.

[5] Netflix dataset. http://www.select.cs.cmu.edu/code/graphlab/datasets/. Accessed: 04-16-2015.

[6] F. Ahmad, S. T. Chakradhar, A. 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.

[7] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[8] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, Apr. 2015.

[9] B.-G. Chun, G. Iannaccone, G. Iannaccone, R. Katz, G. Lee, and L. Niccolini. An energy case for hybrid datacenters. *SIGOPS Oper. Syst. Rev.*, 44(1):76–80, Mar. 2010.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Z. Fadika, E. Dede, J. Hartog, and M. Govindaraju. Marla: Mapreduce for heterogeneous clusters. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12, pages 49–56, Washington, DC, USA, 2012. IEEE Computer Society.

[12] S. Garg, S. Sundaram, and H. D. Patel. Robust heterogeneous data center design: A principled approach. *SIGMETRICS Perform. Eval. Rev.*, 39(3):28–30, Dec. 2011.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30. USENIX Association, 2012.

[14] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 395–404, Washington, DC, USA, 2014. IEEE Computer Society.

[15] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, Aug. 2014.

[16] N. Jain, G. Liao, and T. L. Willke. Graphbuilder: Scalable graph etl framework. GRADES, pages 4:1–4:6, 2013.

[17] G. Karypis and V. Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

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

[19] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[20] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[21] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[22] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data. Accessed: 04-16-2015.

[23] 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. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[24] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *UAI*, pages 340–349.

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

[26] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

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

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

[29] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. 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.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. 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.

[32] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.