

© 2018 Mayank Bhatt

TOPOLOGY-AWARE DISTRIBUTED GRAPH PROCESSING FOR
TIGHTLY-COUPLED CLUSTERS

BY

MAYANK BHATT

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, 2018

Urbana, Illinois

Adviser:

Professor Indranil Gupta

ABSTRACT

Cloud applications have burgeoned over the last few years, but they are typically written for loosely-coupled clusters such as datacenters. In this thesis we investigate how one can run cloud applications in tightly-coupled clusters and network topologies, namely supercomputers. Specifically, we look at a class of distributed machine learning systems called distributed graph processing systems, and run them on NCSA Blue Waters. Partitioning the graph is key to achieving performance in distributed graph processing systems. We present new topology-aware partitioning techniques that better exploit the structure of the network topologies in supercomputers. Compared to existing work, our new Restricted Oblivious and Grid Centroid partitioning approaches produce 25-33% improvement in makespan, along with a sizable reduction in network traffic. We also discuss optimizations such as smart network buffers that further amplify the improvement. To help operators select the best graph partitioning technique, we culminate our experimental results into a decision tree.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Indranil Gupta, for his advice and guidance during my Master's degree. I would like to thank Jayasi Mehar, a frequent collaborator in my work. I would also like to thank the members of the Distributed Protocols Research Group for their invaluable feedback. Finally, I would like to thank my friends and family, the constant pillars of support in my life.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
1.1	Contributions of this Thesis	2
CHAPTER 2	BACKGROUND	4
2.1	Blue Waters Interconnect	4
2.2	Distributed Graph Processing Systems	4
2.3	Types of Graph Partitioning	7
2.4	Vertex-Cut Partitioning Strategies	8
CHAPTER 3	RELATED WORK	11
CHAPTER 4	PARTITIONING STRATEGIES	13
4.1	Topology-Aware Partitioning	13
CHAPTER 5	SYSTEM OPTIMIZATIONS	18
5.1	Optimizations that worked: Smart Network Buffer Sizes	18
5.2	Optimizations that did not work	18
CHAPTER 6	EVALUATION	20
6.1	Performance metrics for partitioning strategies	20
6.2	Experimental Results	20
6.3	Decision Tree	27
CHAPTER 7	CONCLUSION	30
CHAPTER 8	FUTURE WORK	31
REFERENCES	32

CHAPTER 1: INTRODUCTION

Recent years have seen a massive proliferation in the amount of graph data available, from social networks (like Twitter, Facebook), peer-to-peer networks like Gnutella [1], Web graphs, and autonomous systems [2]. These data troves provide an opportunity to glean useful insights about the nature of such interconnected systems. For instance, what are the PageRank [3] values in a graph? How long are paths (e.g., to provide friend recommendations)? What is the graph’s diameter?

These graphs are enormous—the Facebook graph has over a billion vertices and a trillion edges [4], and the human brain network has many billions of vertices. To answer the above types of questions for large graphs, various *distributed graph processing systems* such as Pregel [5], PowerGraph [6] and GraphX [7] have emerged. Similar to how the MapReduce programming model [8] allowed easily parallelizable programs, such frameworks embody simplicity of development, by allowing users to write programs in a vertex-centric and iterative manner. In each iteration vertices update their own state by exchanging information with their neighbours. The computation finishes after either a pre-specified number of iterations or on convergence (when vertex values stop changing).

Cloud software such as these distributed graph processing engines (alongside other cloud software such as OLAP systems, key-value/NoSQL stores, Hadoop, etc.) were originally built for loosely-coupled clusters, such as datacenters and private clusters. At the same time, there is growing interest in the idea of High Performance Data Analytics, where organizations run existing data processing frameworks on HPC clusters—for instance, PayPal used Hadoop backed by the Lustre filesystem [9] [10]. In addition, there is some interest in Supercomputing as a Service [11]. Finally, developments in cloud-based datacenters such as RDMA [12], Infiniband [13], disaggregated networks [14], etc., all point to the possibility that the datacenter architecture is becoming more tightly-coupled, and is on a path to convergence with traditional supercomputer architectures.

These trends motivate the need to explore how to run such distributed software systems on tightly-coupled clusters. In this thesis we look at a specific class of distributed machine learning systems called distributed graph processing systems, and run them on NCSA Blue Waters [15]. In distributed graph processing, the key to performance is to intelligently but cheaply partition the graph. The goal of partitioning is to reduce the amount of communication during the iterations, so that these iterations complete faster, and thus overall runtime is small. However, complex partitioning can incur high up front cost (before iterations start)

but give only marginal benefit in the total computation time. On the other hand, simplistic partitioning can be fast up front but prolong iterations. We explore how we can combine the ideas in these two approaches to optimize both ingress and runtime.

We target a well-known and open-source distributed graph processing system called PowerGraph [6]. We present two new partitioning techniques called Restricted Oblivious and Grid Centroid, specifically intended for supercomputer interconnects. We call these *topology-aware* variants of partitioning techniques. Our approach takes advantage of the fact that in tightly-coupled topologies, the compute nodes¹ are embedded among the routing networks. We also discuss other system optimizations such as smart network buffers that amplify performance improvements.

1.1 CONTRIBUTIONS OF THIS THESIS

The contributions of this thesis are:

1. We present two new graph partitioning techniques called Restricted Oblivious and Grid Centroid, which allow a distributed graph processing engine to better exploit the network topology of a tightly-coupled cluster, reducing both makespan and network communication.
2. We discuss system optimizations, showing that network buffer size can be exploited to improve performance.
3. We integrate our techniques into PowerGraph, a widely-used engine for distributed graph processing.
4. We experimentally study the impact of our partitioning strategies on a wide variety of graph classes and graph algorithms on the Blue Waters supercomputer.
5. We demonstrate performance improvements in the range 25-33% for makespan, and sizable improvements in network traffic.
6. We present a decision tree to help operators select the best technique based on the nature of the workload.

The rest of this thesis is organized as follows: in Chapter 2, we provide an overview on tightly-coupled cluster interconnects and distributed graph processing. In Chapter 3,

¹To avoid confusion, this thesis uses the term *node* to refer to a computation node/device/machine, while the term *vertex* is used in the context of the graph being computed upon.

we discuss related work in existing literature. In Chapter 4, we present the design for our topology-aware graph partitioning strategies. In Chapter 5, we discuss other systems optimizations that we made to PowerGraph to improve its performance on tightly-coupled clusters. In Chapter 6, we evaluate the performance of our algorithms and optimizations with various graphs and graph algorithms, and present a summary of our results in a decision tree. In Chapter 7, we present the conclusions of this work. Finally, in Chapter 8, we discuss directions for future work.

CHAPTER 2: BACKGROUND

This chapter provides an overview of the interconnect in the Blue Waters supercomputer, along with an explanation of distribution graph processing systems and the various graph partitioning schemes present in our system of choice, PowerGraph.

2.1 BLUE WATERS INTERCONNECT

Blue Waters is a supercomputer run by the National Center for Supercomputing Applications (NCSA) at Urbana, Illinois [15]. It houses over 22000 Cray XE compute nodes and over 4000 Cray XK compute nodes. Running a job on Blue Waters involves requesting an allocation of nodes. This allocation is provided in a custom geometry based on how the available nodes are positioned.

As Figure 2.1 depicts, Blue Waters nodes are connected in the shape of a 3D torus where each node has neighbors along the X, Y and Z axes. The darker nodes represent a sample allocation. The communication cost between two nodes in the torus can be approximated using the Manhattan Distance, which is the total number of hops between the two nodes. Fewer hops result in lower communication time, lower average load on each link, and reduced interference.

In such a tightly-coupled architecture, compute nodes are embedded inside the network topology. In other words, when a compute node A sends messages to compute node B, the message will transit through other compute nodes on its way to B [16]. This is unlike a loosely-coupled datacenter where compute nodes run over a decoupled network (using routers and switches). One of our goals is to exploit this embedding (in our new partitioning schemes), in order to intelligently place data and computation, and to minimize communication to avoid long routes during communication.

Blue Waters also provides users the X, Y and Z coordinates of compute nodes in the system map. Using this, we obtain estimates of communication times between pairs of nodes, which are used in our optimized partitioning strategy discussed in Section 4.1.

2.2 DISTRIBUTED GRAPH PROCESSING SYSTEMS

Graph processing runs in iterations. In an iteration, each vertex gathers its neighboring vertices' values, uses these and its own value to compute its new value, and then scatters its

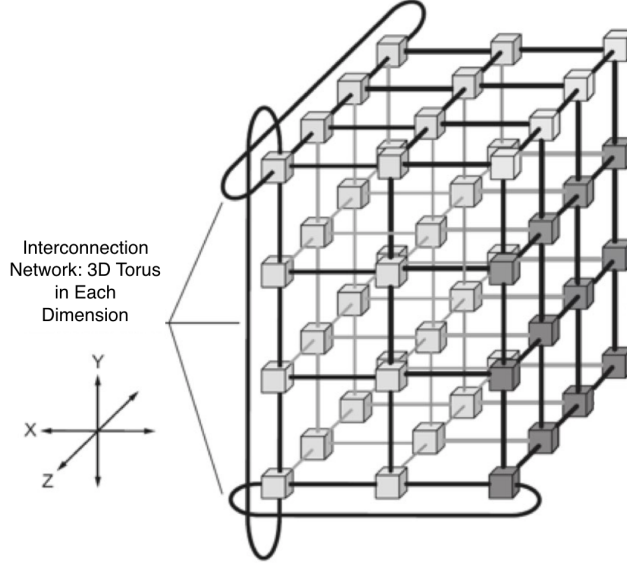


Figure 2.1: **Blue Waters Topology:** *The 3D torus interconnect in the NCSA Blue Waters supercomputer.*

new value to its neighbours. This common approach is called Gather-Apply-Scatter. There is a barrier at the end of each iteration. Computation terminates after either a fixed number of iterations or when a user-specified convergence criterion is met.

Most distributed graph processing systems employ the paradigm called “Think Like a Vertex”. Programmers write programs from the viewpoint of a vertex. For instance, in PageRank [3], the equation to update a vertex’s rank is as follows:

$$PR(v) = \sum_{q, (q,v) \in E} \frac{PR(q)}{|out(q)|} \times 0.85 + 0.15 \quad (2.1)$$

where $out(q)$ represents outgoing edges from vertex q , $PR(q)$ is the PageRank of vertex q , and E is the graph’s edge set.

One of the first distributed graph processing systems is Pregel, developed at Google [5]. Pregel follows a Bulk Synchronous Processing model [17] of graph computation. That is, each vertex is assigned to a node, and each process computes new values for each vertex it owns, and this entire process is carried out in a series of synchronized ‘super steps’, where there is a barrier between the different super steps in the processing. Thus, every process must finish its computations for an iteration before the algorithm can proceed with the next iteration. Pregel uses an edge cut approach to graph partitioning, which means there is a 1:1 mapping from graph vertices to nodes in the framework.

2.2.1 Power Law Graphs

There are certain types of graphs, known as power law graphs, which follow what is called a scale free distribution. This means there are certain hubs or highly connected vertices. As discussed in [6], for a power law graph, the probability that a vertex has degree k is given by the equation

$$P_{deg}(k) \propto k^{-\alpha} \quad (2.2)$$

where the exponent (α) represents the 'skew' of the distribution. A skew of around 2.5 is common in natural graphs, for instance, the Internet is estimated to have a skew of around 2.2 [18].

For Pregel, with such graphs, where a fraction of vertices have a large number of edges, the barrier means that such vertices become bottlenecks for computation, as they must be processed before the next iteration can begin.

PowerGraph has been designed to work with graphs following the power law distribution. With a vertex-cut strategy dividing the graph on multiple nodes, it ensures that the few heavily connected vertices are no longer bottlenecks. Computation is modeled using the Gather-Apply-Scatter (GAS) paradigm, and many graph processing algorithms can be expressed in this way.

2.2.2 Graph Algorithms

Several algorithms can be expressed using the Gather-Apply-Scatter paradigm, and each one of them has different computation and network characteristics.

- **PageRank** - A link analysis algorithm that is used to rank websites in a web link graph.
- **Approximate Diameter** - Used to find the diameter of the given graph.
- **Undirected Triangle Counting** - Counts the number of three edge cycles in a graph.
- **Single Source Shortest Path (SSSP)** - This algorithm computes the shortest path from a single vertex to all other vertices in the graph.
- **K-core Decomposition** - Finding the maximal subset of a graph where the degree of all the vertices is at least K [19].

Frameworks for distributed graph processing have also found their use in expressing machine learning algorithms. For instance, PowerGraph ships with an implementation of Latent

Dirichlet Allocation [20], used in analyzing text corpora, and collaborative filtering [21], used in providing recommendations to users based on their interests and the interests of other users. Other examples include matrix factorization [22], click through rate prediction [23], and Collapsed Gibbs Sampling [24]. Given the growing interest in machine learning and massive amounts of data available, optimizing the performance such frameworks has become increasingly important.

2.3 TYPES OF GRAPH PARTITIONING

To effectively use a cluster and parallelize computation (within an iteration), the graph needs to be partitioned across its computation nodes. This partitioning is done automatically by the graph computation engine (rather than by the user or application). There are two broad approaches to partitioning graphs.

2.3.1 Vertex-cut partitioning

Each edge in the graph gets assigned to a node in the cluster. This results in vertices having replicas, called *mirrors* (Figure 2.2-a). PowerGraph uses vertex-cut partitioning.

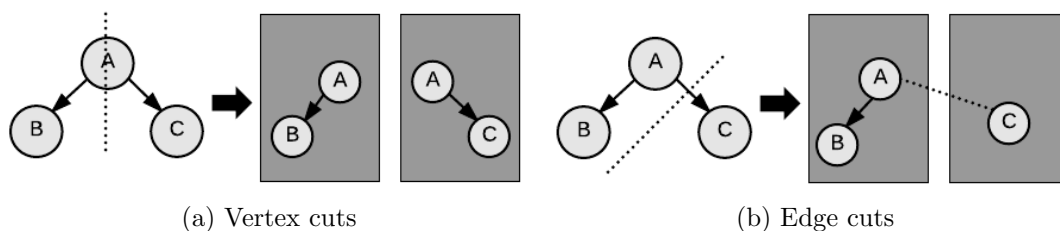


Figure 2.2: **Two Styles of Graph Partitioning:** *Boxes represent compute nodes. Our work uses Vertex-cut partitioning.*

2.3.2 Edge-cut partitioning

Each graph vertex gets assigned to a node, thus ‘cutting’ edges (Figure 2.2-b). Pregel [5] uses edge-cut partitioning.

Past work has shown that vertex-cuts are superior to edge-cuts, performance-wise [6]. Essentially, vertex replication works well with power-law and power-law-like graphs, the most commonly found kind of graphs. These graphs have some vertices with a very high

degree and many vertices with low degrees, e.g., social network graphs, Web graphs, internet graph. Replication helps spread load and reduce computation time (in iterations) for heavy vertices’ values. Henceforth, this thesis only focuses on vertex-cut partitioning.

2.3.3 Hybrid-cut partitioning

PowerLyra introduced a hybrid-cut approach to partitioning, where vertex cuts are used for vertices with higher degrees, and edge cuts are used for vertices with lower degrees. This approach prevents mirroring for smaller degree vertices.

Since the hybrid approach still retains vertex cuts for higher degree vertices, our topology awareness strategies would still be applicable there. Future work might include extending our algorithms to work with hybrid cuts, and analyzing performance benefits in such systems.

2.4 VERTEX-CUT PARTITIONING STRATEGIES

A distributed graph processing system chooses one compute node to be the *master* of a vertex. The master communicates with the mirrors (replicas) of that vertex, thus acting as a central collection and distribution point in each iteration. Mirrors of a vertex perform local computation and aggregation. They send their partial vertex values to the master, which calculates the new aggregated vertex value, and then sends it back out to the replicas so they can use it in the next iteration. Minimizing the time and network overhead of this master-mirror communication is critical as it reduces iteration time (time to barrier) and thus decreases runtime.

In general, partitioning techniques aim to minimize vertex replication factor. High replication factors have been shown to be correlated with increased execution time [25] as the synchronization cost across nodes rises.

In this thesis we focus on the PowerGraph distributed graph processing system [6]. PowerGraph’s native graph partitioning strategies include:

2.4.1 Random

This approach randomly assigns edges of the graph to nodes. While it balances load, random partitioning creates a high replication factor and thus longer runtime.

2.4.2 Oblivious

This uses a greedy heuristic to place edges at nodes. As discussed in the appendix of [26], based on the source and target vertices of the edge, there are 4 cases:

1. Both vertices have at least one mirror node in common: place the edge at the least loaded common node.
2. Only one vertex has been placed so far: place the edge at its least loaded mirror node.
3. Neither vertex has been placed so far: place the edge at the least loaded node in the cluster.
4. Both vertices have been placed, but have no mirror nodes in common: place the edge at the least loaded mirror node in the union of the two mirror sets.

Note that the ingress time (loading time) incurred by Oblivious can be high because these steps require repeatedly computing cluster-wide costs.

2.4.3 Grid

This strategy maps all compute nodes into a 2D grid to constrain the candidate nodes for each vertex. It requires the number of computation nodes to be a perfect square. Nodes in the cluster are arranged into a square grid (node position within this grid is independent of network topology). Each vertex gets hashed to one (row, column) pair, and is permitted to be replicated only across that set of nodes. When placing an edge, the intersection of the two sets of possible nodes is considered and the edge is hashed onto a node in that intersection

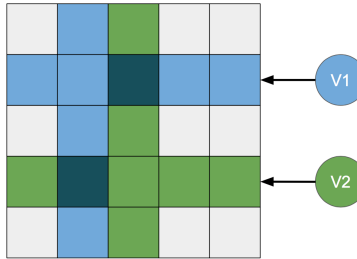


Figure 2.3: **Grid Partitioning:** *An illustration of the Grid partitioning strategy. An edge between the two vertices $V1$ and $V2$ would be placed at one of the intersection cells.*

Consider the example in Figure 2.3. 25 nodes are arranged in a 5x5 grid. Assume vertex $V1$ hashes to the blue nodes, and vertex $V2$ to green. The $V1$ - $V2$ edge can only be replicated at the two intersecting nodes in the grid—this is chosen by another hash.

2.4.4 HDRF (Highest Degree Replicated First)

This strategy is similar to the Oblivious strategy, but prefers to replicate higher degree vertices first allowing better parallelism.

CHAPTER 3: RELATED WORK

This chapter presents an overview of related work in the fields of distributed graph processing, high performance computing, and cloud computing.

Network heterogeneity is common, not only in Blue Waters like supercomputers, but also in commodity data centers. Nodes connected to the same switch can communicate over a high speed link, while distant nodes get poorer performance due to the multi-hop forwarding of packets [27]. Using this underlying heterogeneity to optimize distributed computations has caught the interest of many researchers over multiple domains.

Over the years, domains like high performance computing (HPC) [28], stream processing [29], batch processing [30], scheduling [31] etc. have explored network topology-based scheduling. Within graph processing, there has been some work on vertex-cut approaches for graph partitioning that account for compute heterogeneity or monetary costs of communication, but they are not intended for tightly-coupled clusters.

Choreo [32] builds a network aware placement system for cloud applications. Their positive results compared to other placement methods that do not consider network rates or inter-task communication patterns, highlight the need for topology-aware application placement.

In the field of parallel computing, topology mapping is the problem of mapping an application communication topology to physical nodes such that the application communication efficiently utilizes the underlying physical links. Hoefler and Snir [33] proved that the topology mapping problem is NP-Complete, and presented heuristics for performing such an embedding. However, their work, as well as Choreo’s, needs an existing application communication graph, i.e., a traffic matrix (which is actually created by the graph partitioning strategies in systems like PowerGraph [6]) and is hence orthogonal to our direction of work.

Parallel computing has seen other work with respect to network topology. Sack and Gropp [34] presented improvements to MPI primitives like AllGather and Reduce-Scatter that are more efficient for certain network topologies like Clos and Torus networks. By using some redundant communication, they are able to reduce network congestion and improve the overall performance of communication. PARAGON [35] is another approach to task allocation and placement in HPC systems that considers both inter-node (network topology) and intra-node (non uniform memory access) communication costs while partitioning a task graph. They also account for shared resource contention within a single node while making such partitioning decisions. Ajwani et al. [36] also present a similar approach to high performance computing where they leverage a reconfigurable topology to further improve the performance

of applications. However, all of the above work is targeted at high performance computing applications, where the application topology is already available to be partitioned among the compute nodes.

In the area of distributed graph processing, there has been some previous work that takes topology into account. GrapH [37] is an example of such a framework, however, it focuses on a geodistributed setting, where the aim is to minimize monetary communication costs (for instance, in an EC2 environment where inter-region data transfer costs money) rather than latency and performance. Surfer [38] performs bandwidth aware graph partitioning by measuring network bandwidth in inter-node communication. However, it uses a Pregel-like edge-cut approach [5], which is inferior in performance to vertex-cuts in systems like PowerGraph [6].

LeBeane et al [39] use a different approach to partitioning, where they consider factors like node memory and computation speed while partitioning graphs. Similarly, Xu et al [40] also consider heterogeneity but they use edge cuts. While we do not account for heterogeneity, our work engenders an interesting new avenue engendered for exploration: considering both heterogeneity and tightly-coupled clusters in partitioning. We leave this as an open problem.

MOSAIC [41] is a graph processing system that leverages coprocessors for graph processing and makes NUMA-aware partitioning decisions when distributing a graph, using Hilbert curves and edge cuts. However, it runs on a single machine instead of on a cluster. As the compute nodes in Blue Waters also have NUMA properties, it might be worth exploring potential optimizations to the graph data structure within a node to make the best use of memory locality within a node.

Verma et al. [26] present a comparison of various graph partitioning strategies in graph processing frameworks. They also provide recommendations on the choice of partitioning strategy depending on the properties of the graph and the being processed and the running algorithm. We extend this idea by evaluating how these various strategies end up performing in environments with a heterogeneous, tightly-coupled network and how that impacts the overall system performance.

The results from our smart network buffers in Section 6.2.8 parallels existing work on how injection bandwidth impacts application scalability [42].

Finally, research has looked at running HPC applications on AWS clouds [43]. This work explores the opposite direction.

CHAPTER 4: PARTITIONING STRATEGIES

This chapter discusses the topology-aware algorithms we implemented in PowerGraph to better leverage the interconnect of tightly-coupled clusters.

4.1 TOPOLOGY-AWARE PARTITIONING

We aim to incorporate the underlying network topology into the graph partitioning stage. Partitioning a graph in a distributed cluster needs to achieve two goals. The first is *edge placement*: this involves picking which machine to place edges at. This results in the creation of vertex mirrors (replicas). The second goal is *master placement*: we need to choose, per vertex, which compute node will have the master (to coordinate among the mirrors of that vertex). Partitioning algorithms can perform master placement before or after mirror placement. We consider both these cases, and propose new techniques for each case.

4.1.1 Weighted Centroid: Topology-Aware Master Placement

We first target the scenario where the mirrors of a vertex have already been placed and now a master needs to be placed (for each vertex). State of the art graph partitioning algorithms have only focused on strategies to improve edge placement. All of these strategies employ the same way of picking master nodes for the vertices—hashing (among all nodes in the cluster). While hashing ensures that the number of masters on each node is equally distributed, it can lead to the masters being placed far away from their mirrors in the underlying network topology, thus injecting more traffic into the network and prolonging the communication phase of each iteration.

To minimize the master-mirror communication, we propose placing the master at a node that minimizes the average distance (number of hops) between the master and its mirrors. We call this the *centroid node*. Note that the centroid need not be a mirror of the vertex.

Fig. 4.1(a) shows an example where we have a vertex mirrored on two compute nodes, A and B. Using hashing, the node C could be picked as the master for the two mirrors on A and B. However, that is not an ideal placement for a master since both mirrors are multiple hops away. Fig. 4.1(b) shows how picking the centroid (node D) as the master lowers communication costs. Concretely, for this example, hashing the master in an $3 \times M$ grid results in a mirror-master distance that is $O(M)$, while placing it at a centroid results

in mirror-master distance that is $O(1)$. In general, this approach can save up to 25% hops (as Table 6.1 will show later).

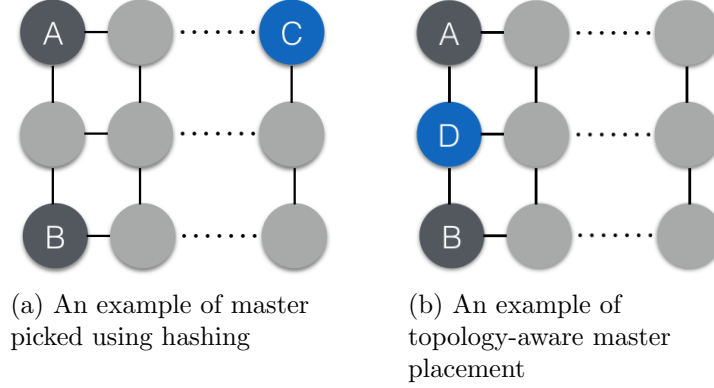


Figure 4.1: **Weighted Centroid:** *Differences between hashing and topology-aware master placement.*

Weighted Centroid If all mirrors of a vertex had identical load (number of local edges placed at that mirror), then a geographical centroid would suffice. However, in reality mirrors have heterogeneous load. This is because different mirrors have different degree, i.e., number of neighboring vertices that are local at the computation node.

The data exchanged between the master and a mirror is a function of this edge degree of the mirror. For instance, Triangle Counting sends a vector of edges each iteration, and thus a mirror with more edges assigned to it sends large amounts of data to the master.

To target such cases we present a technique to choose a weighted centroid that is skewed towards “heavier” mirrors. Given a candidate node c for placing the master and a mirror node m , we calculate the cost to capture the overhead of network communication between c and m . This cost, which we name *weighted hop count* (whc), is:

$$whc(c, m) = n_{edges}(m) \times n_{hops}(c, m) \times l(c) \quad (4.1)$$

Here, n_{hops} is the Manhattan distance between the candidate and mirror node. The degree of the mirror (n_{edges}) gives more weight to heavier mirrors. $l(c)$ is a function of the number of masters already placed at candidate c . This is incorporated to evenly distribute vertex masters in the cluster.

The total score for candidate node c is the sum of all whc costs across mirrors. The node with the lowest total (weighted hop count sum) is chosen as the master for the vertex.

$$master = \underset{c}{\operatorname{argmin}} \left(\sum_{m \in \text{mirrors}} whc(c, m) \right) \quad (4.2)$$

Our topology-aware master placement can be combined with any arbitrary edge placement strategy. In our implementation and experiments (Section 6.2) we combine our centroid approach with the Grid partitioning strategy. We chose Grid because it keeps the replication factor low [25] and refer to this approach as *Grid Centroid*.

4.1.2 Restricted Oblivious: Topology-Aware Mirror Placement

We propose a topology-aware algorithm for edge placement in the second scenario, when the master for each vertex has already been placed. We augment the greedy heuristic used in Oblivious graph partitioning (Section 2.4) to account for network topology when placing a graph edge. Oblivious uses hashing over all nodes to select masters for each vertex—we retain this feature. Our intuition is to use the (x, y, z) coordinates of these pre-computed masters and place mirrors around it, minimizing communication costs while maintaining load balance.

Our new Restricted Oblivious technique follows these rules (vs. the Oblivious rules of Section 2.4.2):

1. Both vertices have the same master node: place the edge at the common master node.
2. Both vertices have at least one mirror node in common: place the edge at the least loaded common mirror node.
3. Neither vertex has any mirror in common: place the edge based on a cost function accounting for topology score and load balance score.

Compared to the Oblivious heuristics of PowerGraph (Section 2.4.2), our technique is different because it accounts for masters, and uses topology-aware and load-aware scores to place edges.

Algorithm 4.1 shows the pseudocode for placing an edge using our heuristics. We look at the masters and mirrors of the two vertices associated with the edge. These are represented as *src_master*, *dst_master*, *src_mirrors* and *dst_mirrors* respectively in the pseudocode. We detail our steps.

1) Our initial check returns the common master node as the candidate (lines 1 - 3). Reusing the master nodes as the mirrors is beneficial because it reduces the overall replication factor.

2) If the source and destination master nodes are different, we find the intersection of the nodes containing the mirrors of the two vertices (lines 4 - 6). If the intersection is not empty, we find the node within that set which has the least number of edges, and place the edge at it.

Algorithm 4.1 Restricted Oblivious: Algorithm for picking node to place a graph edge.

```
1: if (src_master == dst_master) then
2:   return src_master
3: end if
4: src_mirrors = replicas of source vertex
5: dst_mirrors = replicas of destination vertex
6: candidates = Intersect(src_mirrors, dst_mirrors)
7: if (candidates.size > 0) then
8:   return least loaded candidate
9: end if
10: TS[][] = Precomputed Topology Score using Equation 4.5
11: n_edges[] = Number of edges on each node
12: candidates = All nodes in the cluster
13: max_cost = Integer.MIN_VALUE
14: best_candidate = null
15: for i = 0 to candidates.length do
16:   b_score = LS(i)
17:   t_score = TS[i][src_master][dst_master]
18:   if (b_score + t_score > max_cost) then
19:     max_cost = b_score + t_score
20:     best_candidate = i
21:   end if
22: end for
23: n_edges[best_candidate] ++
24: return best_candidate
```

3) If there is no intersection between the mirrors of the source and destination vertices (lines 18 - 20), all the nodes in the cluster become candidates. To reduce inter-master distance (between $m1, m2$) while balancing load, we select:

$$mirror = \underset{c \in \mathcal{N}}{\operatorname{argmax}}(LS(c) + TS(c, m1, m2)) \quad (4.3)$$

Here, LS is the *load score* that captures the load of edges on the candidate node and is calculated by Equation 4.4. TS , the *topology score*, captures the communication cost of placing the edge on this candidate. It is calculated in Equation 4.5.

Given the number of edges on a candidate node (n_edges) and the maximum and minimum number of edges on a node in the cluster (max_edges and min_edges) so far, we compute the relative load on each candidate:

$$LS(c) = \frac{max_edges - n_edges(c)}{\epsilon + (max_edges - min_edges)} \quad (4.4)$$

Used in line 16 of the pseudocode, $LS(c)$ returns a value between 0 and 1, where 0 corresponds to the most loaded and 1 corresponds to the least loaded node in the cluster. It is calculated by dividing the difference of maximum edges and the total number of edges on a node by the difference of the maximum and minimum number of edges on a node in the cluster. Using the difference with max_edges helps constrain the score in the range $[0, 1)$. ϵ is a small positive constant needed to avoid dividing by zero (we set it as 1.0 in all our experiments, regardless of graph and algorithm).

Topology score TS finds the communication cost for candidate c given the source and destination masters, $m1, m2$.

$$TS(c, m1, m2) = \frac{2 \times n_hops(m1, m2) - (n_hops(c, m1) + n_hops(c, m2))}{\delta + n_hops(m1, m2)} \quad (4.5)$$

A lower TS means that the candidate is farther away from the two masters while a higher TS means that the placement is more favourable topologically. n_hops is the number of hops between candidate c and masters ($m1, m2$), and δ ($= 0.001$) is a small positive constant to avoid dividing by zero. In an ideal candidate for the edge, the sum of hops between the candidate and the two masters ($n_hops(c, m1) + n_hops(c, m2)$) should be smaller than the distance between the two masters $n_hops(m1, m2)$. We use the difference in the numerator so that the topology score maximizes when the candidate c lies on the shortest paths between master nodes $m1, m2$.

With Blue Waters' static routing (Section 2.1), we pre-compute a 3D matrix with topology scores for all master-candidate node combinations and use it while picking a new mirror (line 17). This speeds up ingress time.

CHAPTER 5: SYSTEM OPTIMIZATIONS

In this chapter, we present other approaches to systems optimizations to improve the performance of distributed graph processing systems on tightly-coupled clusters.

Running a real system on a real cluster requires us to pay attention to important system optimizations. We explored a wide variety of optimizations, and found some of them more effective than others. We describe the optimizations that worked (smart network buffer sizes) as well as ones that did not—we believe discussing the latter is important for the research community, so that researchers and developers can avoid exploring them in the future.

5.1 OPTIMIZATIONS THAT WORKED: SMART NETWORK BUFFER SIZES

By intelligently selecting the network buffer size as a function of graph and computation we can optimize network communication and performance. For efficient use of the network, a mirror/master does not send data right when it is available. Instead, a network buffer is used, and when it is full, it is flushed out on to the network (this happens in the background).

We can control the size of this network buffer, to affect the batch size of messages sent out on the communication network. For instance, if the messages are small and the computations are simple (e.g., floating point in PageRank), we recommend smaller network buffer sizes than if the messages are bigger (e.g., vectors in Triangle Count) or computations are complex (e.g., bitwise or’s in Approximate Diameter). Section 6.2 shows that this optimization can give additional improvements of upto 48% in runtime.

5.2 OPTIMIZATIONS THAT DID NOT WORK

Batched Oblivious We attempted to improve ingress times by batching the process of Oblivious’ ingress, by only recomputing the heuristic every K edges, where K is a tunable batch size. While this improved ingress times, we found that it increased runtime. This is because load balance information is often stale, and this results in poor quality of partitioning. This optimization is worth pursuing only in cases where the ingress time greatly exceeds computation time, e.g., loading a large graph, followed by running the Triangle Count algorithm.

Per worker parallelism We only observed performance improvements as we increased the number of worker threads upto the number of cores. Beyond that, the cost of thread context switches became excessive, and runtime suffered. Instead, PowerGraph’s lock-free approach to thread parallelism makes better use of the cores of the system.

CHAPTER 6: EVALUATION

This chapter presents our experimental results with various graphs, algorithms, partitioning strategies, and network buffer sizes, and concludes with a summary of results captured in a decision tree that recommends the most appropriate partitioning strategy for a given scenario.

6.1 PERFORMANCE METRICS FOR PARTITIONING STRATEGIES

There are several metrics that we consider for the performance of a graph partitioning algorithm. The following is a list of all metrics that we are using for evaluating our techniques.

- **Ingress time** - The amount of time it takes to convert the input graph into an in-memory format suitable for graph processing algorithms to work with. Simple algorithms can result in fast ingress, while more complicated ones increase this overhead.
- **Computation time** - The amount of time spent by the system running the graph algorithm. Simple algorithms such as random may result in larger computation time, while algorithms like oblivious greedy can result in lower computation time.
- **Replication factor** - The average number of mirrors of a vertex.
- **Load** - The distribution of masters across nodes. A skewed load distribution means some nodes do a lot more computational work than others. Ideally, we should have an even distribution, which means each vertex does approximately equal amounts of computational work.
- **Average master-mirror hop count** - The further apart a master and its mirrors are, the higher the communication cost between them.

6.2 EXPERIMENTAL RESULTS

6.2.1 Setup

We modified PowerGraph to incorporate Grid Centroid, Restricted Oblivious, and smart network buffers. We performed experiments on the NCSA Blue Waters supercomputer, using

Partitioning Strategy	Replication Factor	Average Master-mirror hops
Grid	10.9	13.1
Grid Centroid	10.7	9.8
Oblivious	26.1	12.5
Restricted Oblivious	22.6	11.6

Table 6.1: **Replication Factor and Hop Count:** *Topology-aware variants have reduced number of hops.*

Cray XE6 nodes. Each XE6 node has 2 AMD Interlagos model 6276 CPU processors with a nominal clock speed of at least 2.3 GHz and 64 GB of physical memory [15].

We use multiple datasets for our experiments. These include publicly available graphs like Twitter-2010 [44] and UK-2007 [45] [46], and those generated from a synthetic graph generators like Darwini [47] and [48].

6.2.2 Master-Mirror Distance

In order to determine that our topology-aware variants are placing masters and mirrors close by, we partition a graph with 5 Billion edges on a cluster of 36 nodes using different partitioning strategies and measure the average number of hops between the master and mirrors. To find this average, we take the total sum of the master-mirror distances for all vertices (\mathcal{V}) in the graph and divide it by the number of vertices.

$$Average = \frac{\sum_{v \in \mathcal{V}} (\sum_{m \in mirrors(v)} (n_{hops}(master(v), m)))}{|\mathcal{V}|} \quad (6.1)$$

As Table 6.1 shows, Grid Centroid reduces the hops to 9.8 (25% lesser than Grid), and Restricted Oblivious gives a 10% reduction compared to Oblivious.

6.2.3 Runtime and Ingress

We evaluate the effect of our partitioning techniques on the user-facing performance metrics. In all our experiments smart network buffers were disabled, unless otherwise stated. We run the Approximate Diameter algorithm on a graph of 20 million vertices [48] on a cluster size of 36 compute nodes.

Figure 6.1 shows the trends in ingress time and runtime (excluding ingress) using different partitioning strategies. We observe that Grid Centroid brings upto 20% runtime improvement over baseline Grid, and Restricted Oblivious brings 32% runtime improvements compared to baseline Oblivious. The runtime for Grid Centroid is overall the lowest for this

graph. This benefit is in spite of a slightly longer time for graph ingress (because we are intelligently selecting masters and mirrors). For the majority of applications where ingress time is much smaller than the runtime, our approaches provide tangible benefits.

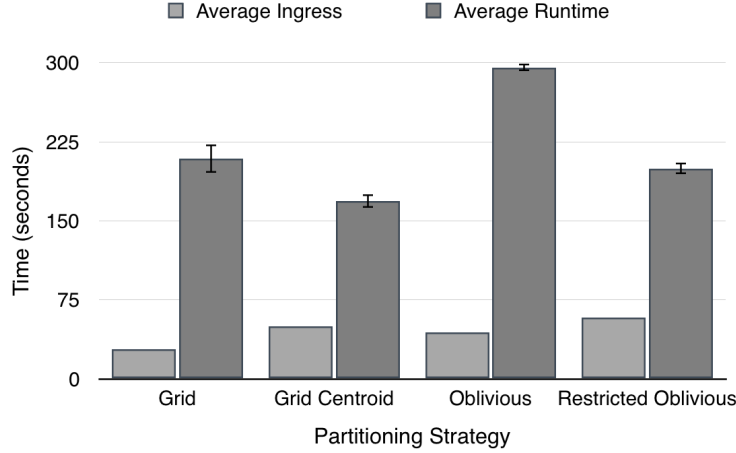


Figure 6.1: **Ingress and Runtime:** *Impact of different partitioning strategies on ingress and runtime.*

6.2.4 Graph Algorithms

We experiment with three graph algorithms that cover a wide swathe of computation and communication patterns: PageRank, K-Core and Approximate Diameter. For PageRank, the amount of data sent decreases across iterations as the number of active vertices decreases. For K-Core decomposition, each mirror sends integer values to its masters conveying the decrement counter for that iteration. In every iteration for Approximate Diameter, each mirror sends bitmaps of size $O(|V|)$ to its master. This is more voluminous than K-core and PageRank.

Figure 6.2 shows that our partitioning strategies give different levels of improvements depending on how message-heavy the algorithm is. We see that Grid Centroid performs 20% better than Grid for Approximate Diameter, whereas for K-core and PageRank the improvements are smaller (2% - 5%). Thus topology-aware algorithms have the biggest impact when the algorithm is message-heavy. When the amount of data sent is low, decreasing the communication cost does not decrease the overall runtime significantly.

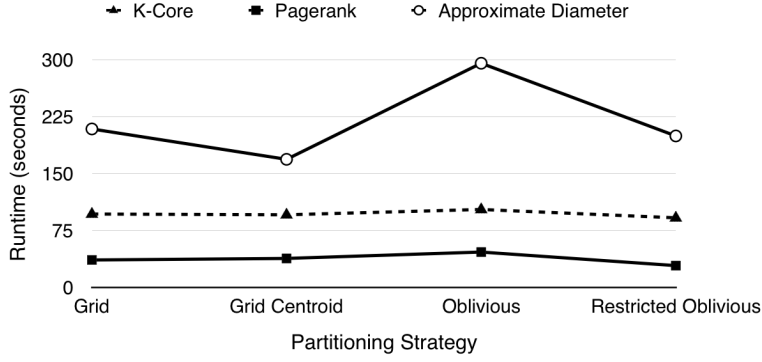


Figure 6.2: **Graph Algorithms:** *Impact of different partitioning strategies on different graph algorithms.*

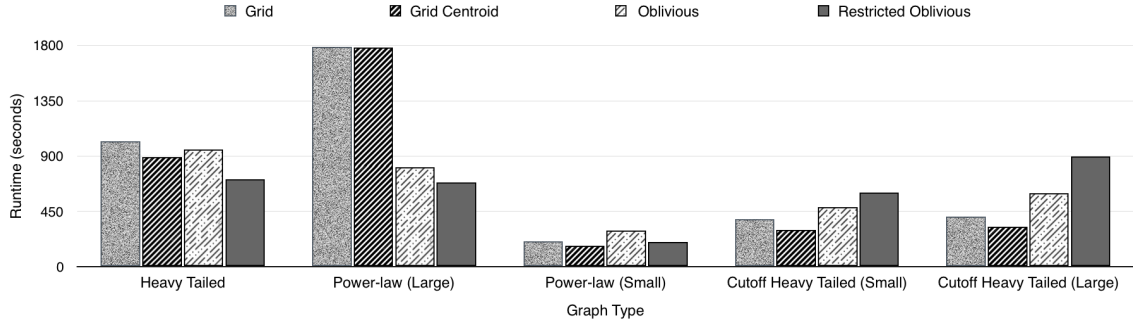


Figure 6.3: **Variation in Runtime for Different Graphs:** *Restricted Oblivious outperforms Oblivious on Heavy-Tailed and Power-law (large), while Grid Centroid outperforms Grid on Power-law (small) and 2 Cutoff Heavy-Tailed graphs.*

6.2.5 Different Graphs

In this section we evaluate how different graph types affect the performance gains from our proposed partitioning strategies. Many well-known graphs are *heavy-tailed* in nature, i.e., they have some vertices with very high edge degree—a prominent example is the Twitter graph. In social networks like Facebook, the number of friends that a user can have is upper-bounded by the system [49]. Thus we refer to Facebook-like graphs as *cutoff heavy-tailed*. A third category is graphs that have a *power-law* distribution—this is a special subclass of heavy-tailed; examples include Web graphs.

We study how these properties affect the overall runtime with our partitioning strategies. As Table 6.2 shows we use the Twitter graph, the UK-2007 graph, and synthetically generated graphs to cover the three classes of graphs. We run Approximate Diameter on a 36 node cluster for all graphs. (For the UK-Web graph we had to run our experiments on a 64

Graph	Kind	#vertices	#edges
Twitter [44]	Heavy Tail	40M	1.5B
UK-Web [45] [46]	Power-law (Large)	109M	3.7B
Synthetic 1 [48]	Power-law (Small)	20M	100M
Synthetic 2 [47]	Cutoff Heavy-Tailed	6.1M	2B
Synthetic 3 [47]	Cutoff Heavy-Tailed	15.3M	5B

Table 6.2: **Graphs used in our experiments.**

node cluster because of memory limitations.)

Figure 6.3 shows our results. In the heavy-tailed graph, Restricted Oblivious gives 25% improvement over Oblivious, while Grid Centroid gives 15% improvement over Grid. For the power-law graph (large), we did not observe any improvement for Grid Centroid, but Restricted Oblivious gives 15% improvement over Oblivious. We also note that for the cutoff heavy-tailed graphs, Restricted Oblivious is slightly worse than Oblivious, possibly due to the load imbalance of edges, yet Grid Centroid still gives 25% improvement over Grid. In each of the graph classes, at least one of our topology-aware variants outperforms the baseline. Later we will use these results in creating our decision tree (Section 6.3).

6.2.6 Minimizing Network Traffic

Since our topology-aware algorithms attempt to keep mirrors relatively close to their masters, we are able to reduce the amount of data that is sent over long hops. We plot the CDF of bytes versus hop count in Figure 6.4. This uses the Approximate Diameter computation over the Twitter graph.

We observe that our topology-aware algorithms shift the distribution towards the left (lower number of hops), implying that more data is sent over fewer hops. For instance, Grid Centroid restricts 75% of network traffic to be within 2 hops, compared to Grid and Oblivious (whose curves overlap), which send only 57% of traffic within 2 hops.

[50] demonstrated the importance of a metric called *hop-bytes* to measure performance of high performance computing applications. Hop-bytes can be computed as:

$$HB = \sum_{i \in \text{messages}} \text{bytes}(i) \times \text{hops}(i) \quad (6.2)$$

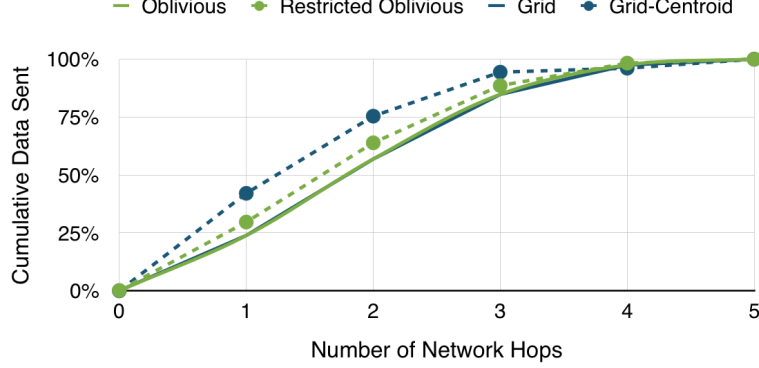


Figure 6.4: **Bytes-Network Hops CDF:** *Our topology-aware variants send less data over larger hops.*

Partitioning Strategy	Hop-Bytes	Reduction from Baseline
Grid	1.02×10^{12}	-
Grid Centroid	7.84×10^{11}	23.4%
Oblivious	1.39×10^{12}	-
Restricted Oblivious	9.24×10^{11}	33.3%

Table 6.3: **Hop-Bytes:** *Across partitioning strategies.*

As PowerGraph instruments the number of bytes exchanged between each pair of nodes, we can compute the hop-bytes metric for different partitioning strategies when running Approximate Diameter on Twitter.

Table 6.3 shows that our topology-aware partitioning schemes reduce the hop-bytes metric compared to the baseline strategies. For instance, Grid Centroid reduces the hop-bytes metric by around 23% compared to Grid, and Restricted Oblivious reduces it by around 33% compared to Oblivious.

Thus, even in cases where our topology-aware algorithms do not significantly impact runtime, they yield the benefit of reducing network transfer volume, and network congestion.

6.2.7 Scalability with Cluster Size

We evaluate the performance of our partitioning strategies for different cluster sizes in Figure 6.5. On the smaller cluster of 16 nodes, we observe Restricted Oblivious shows 25% reduced runtime compared to Oblivious. Grid Centroid takes 5% more time than Grid in the 16 node cluster because of uneven master load. However, as the size of the cluster increases, network topology starts mattering more and we see more significant improvements. Restricted Oblivious gives 34% reduced runtime and Grid Centroid gives 20% reduced runtime

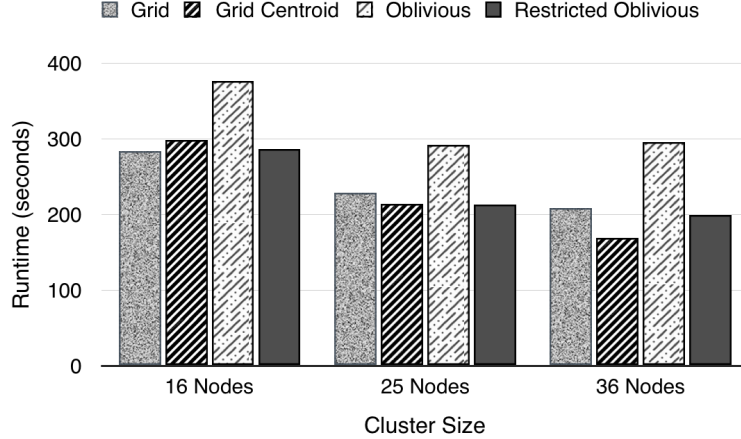


Figure 6.5: **Impact of Cluster Size on Runtime:** *The benefits of topology-aware partitioning increase as the cluster size grows.*

compared to their respective baselines on a cluster of 36 nodes. Since more nodes implies bigger clusters with higher hop counts, our topology-aware variants are able to perform better.

6.2.8 Smart Network Buffers

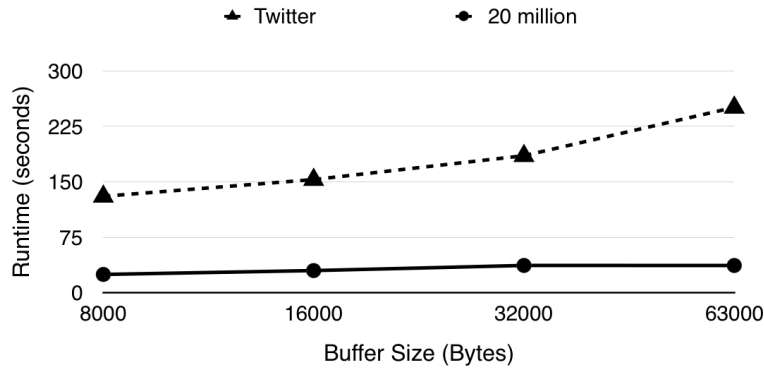


Figure 6.6: **Impact of Network Buffer Size for PageRank on different graphs.** *Lower buffer sizes result in better runtimes.*

In this section, we analyze the performance impact of configuring smart network buffers (described in Section 5.1). By carefully selecting their size based on the algorithm, we can achieve significant performance improvements. We plot in Figures 6.6 and 6.7 the runtimes observed for different algorithms and datasets as we tune the network buffer sizes. We observe that PageRank (Figure 6.6) benefits the most from lower buffer sizes (corresponding

to higher rates of data injection), the runtime reducing by 48% for the Twitter graph and 37% for a synthetic 20 million vertex graph, while Approximate Diameter is not affected significantly (or consistently).

We hypothesize that this difference is due to the nature of the graph computation taking place. PageRank sends small floating point values over the network and performs simpler computations, so the frequent data injection into the network reduces the time spent waiting for data, while Approximate Diameter sends large bitmaps and performs much more complicated computations in its vertex program, and thus frequent flushing does not impact its performance much (runtime trends with varying buffer sizes were not consistent). Hence we conclude that smaller network buffers should be used only when the algorithm is not message-heavy.

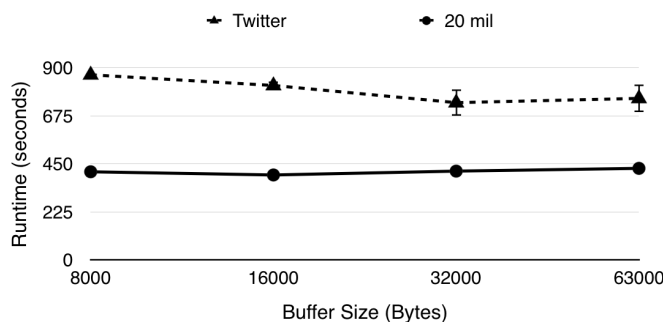


Figure 6.7: **Impact of Network Buffer Size** for *Approximate Diameter* on different graphs. The impact on runtime is not significant, unlike PageRank in Fig. 6.6.

6.3 DECISION TREE

Developers have a myriad choice of partitioning techniques for their distributed graph processing systems. Literature provides suggestions for selecting the right techniques for loosely-coupled clusters [25].

We now present a decision tree for selecting partitioning strategies for tightly-coupled clusters. Our decision tree (shown in Figure 6.8) takes as input characteristics of the graph as well as of the application. We explain it below.

Our experiments for different types of graph algorithms in Section 6.2.4 showed that algorithms which are more message-heavy have higher performance gains with topology-aware partitioning. We use this parameter as our first branch condition in the decision tree. For algorithms that are not message-heavy, we recommend setting lower sizes for network buffers, as they tend to benefit the most from this optimization (Section 5.1).

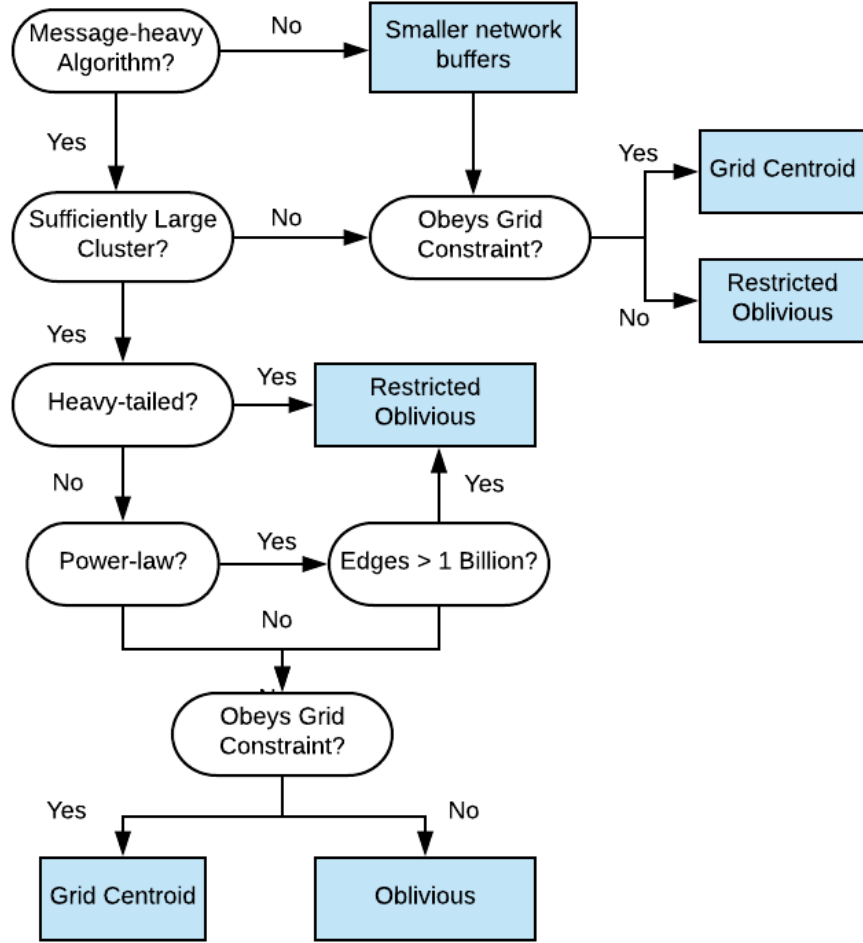


Figure 6.8: **Decision Tree:** *For tightly-coupled clusters, to help operators choose the best partitioning strategy.*

For non message-heavy algorithms or a small cluster (≤ 25 nodes in our experiments), we recommend using Grid Centroid if the Grid constraint can be satisfied (cluster size is a perfect square)—otherwise one should use Restricted Oblivious. We recommend our topology-aware variants because they have similar performance to the baseline algorithms and reduce overall interference.

For message-heavy algorithms running on large clusters (greater than 25 nodes in our experiments), our next level in the tree captures our observation from Section 6.2.5 that heavy-tailed graphs, and large power-law graphs (> 1 Billion edges), benefit most from using Restricted Oblivious.

Smaller graphs (< 1 Billion edges) that are power-law show lowest runtimes with the Grid

Centroid. Grid Centroid is also preferred for cutoff heavy-tailed graphs. Thus, if the perfect square constraint can be met, we recommend using Grid Centroid partitioning, otherwise one should use Oblivious partitioning.

CHAPTER 7: CONCLUSION

In this thesis, we proposed two new graph partitioning algorithms for distributed graph processing engines running on tightly-coupled clusters (like Blue Waters). The key idea was to leverage the fact that routing passes via computation nodes. We proposed two new partitioning techniques: Grid Centroid (which places masters at the weighted centroid of vertex replicas), and Restricted Oblivious (which places mirrors around existing masters, to minimize master-mirror distance). Experiments showed that our techniques reduce runtime (makespan) by 25-33%, and even when runtime benefits are small they reduce network traffic volume by 23-33%. Network optimizations reduce runtime further by 48%. We culminated our experimental findings into a decision tree that can help operators choose the right partitioning strategy in a tightly-coupled cluster topology.

CHAPTER 8: FUTURE WORK

As discussed in Chapter 2, there are three ways of partitioning a graph on a cluster. While our work focuses on vertex cuts, a direction of future work could explore how our algorithms perform work with hybrid cuts in systems like PowerLyra [51]. Since the hybrid approach still retains vertex cuts for higher degree vertices, our topology-aware strategies could be applicable there as well.

In addition, modifying the system to better leverage the NUMA properties of Blue Waters compute nodes could be another avenue of work that can be explored.

REFERENCES

- [1] M. Ripeanu, A. Iamnitchi, and I. Foster, “Mapping the Gnutella network,” *IEEE Internet Computing*, vol. 6, no. 1, pp. 50–57, 2002.
- [2] D. Magoni and J. J. Pansiot, “Analysis of the autonomous system network topology,” *SIGCOMM Computer Communication Review*, vol. 31, no. 3, pp. 26–37, 2001.
- [3] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web,” 1999.
- [4] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at Facebook-scale,” in *Proceedings of the VLDB Endowment*, vol. 8, no. 12. VLDB Endowment, 2015, pp. 1804–1815.
- [5] 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 International Conference on Management of data*. ACM, 2010, pp. 135–146.
- [6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs,” in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2012, pp. 17–30.
- [7] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, “GraphX: A resilient distributed graph system on spark,” in *Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, pp. 2:1–2:6.
- [8] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [9] “Big data meets high performance computing,” *White Paper*: <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/big-data-meets-high-performance-computing-white-paper.pdf>.
- [10] P. Schwan et al., “Lustre: Building a file system for 1000-node clusters,” in *Proceedings of the Linux symposium*, 2003, pp. 380–386.
- [11] “HPC as a Service,” *Article*: <https://www.cray.com/solutions/supercomputing-as-a-service>.
- [12] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler, “An RDMA protocol specification,” 01 2005.
- [13] G. F. Pfister, “An introduction to the infiniband architecture,” *High Performance Mass Storage and Parallel I/O*, vol. 42, pp. 617–632, 2001.

- [14] A. Zamer, “Network disaggregation: What it is, what it can do for your data center,” *Article*: <https://community.hpe.com/t5/Networking/Network-disaggregation-What-it-is-what-it-can-do-for-your-data/ba-p/6796883>.
- [15] “Blue Waters hardware summary,” *User Guide*: <https://bluewaters.ncsa.illinois.edu/hardware-summary>.
- [16] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, “Overview of the Blue Gene/L system architecture,” *IBM Journal of Research and Development*, vol. 49, no. 2, pp. 195–212, 2005.
- [17] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.
- [18] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On power-law relationships of the internet topology,” in *ACM SIGCOMM computer communication review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.
- [19] V. Batagelj and M. Zaversnik, “An $O(m)$ Algorithm for Cores Decomposition of Networks,” *eprint arXiv:cs/0310049*, Oct. 2003.
- [20] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent Dirichlet Allocation,” *Journal of Machine Learning Research*, vol. 3, no. Jan, pp. 993–1022, 2003.
- [21] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, “Large-scale parallel collaborative filtering for the netflix prize,” in *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–348.
- [22] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, “Scalable coordinate descent approaches to parallel matrix factorization for recommender systems,” in *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 765–774.
- [23] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich, “Web-scale bayesian click-through rate prediction for sponsored search advertising in microsoft’s bing search engine,” in *Proceedings of the 27th International Conference on International Conference on Machine Learning*. USA: Omnipress, 2010, pp. 13–20.
- [24] T. L. Griffiths and M. Steyvers, “Finding scientific topics,” *Proceedings of the National academy of Sciences*, vol. 101, no. suppl 1, pp. 5228–5235, 2004.
- [25] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta, “An experimental comparison of partitioning strategies in distributed graph processing,” in *Proceedings of the VLDB Endowment*, vol. 10, no. 5. VLDB Endowment, 2017, pp. 493–504.

- [26] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta, “An experimental comparison of partitioning strategies in distributed graph processing,” *Technical Report, IDEALS* <http://hdl.handle.net/2142/91657>, 2016.
- [27] C. Peng, M. Kim, Z. Zhang, and H. Lei, “Vdn: Virtual machine image distribution network for cloud data centers,” in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 181–189.
- [28] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp, “Multi-core and network aware MPI topology functions,” in *Recent Advances in the Message Passing Interface*. Springer-Verlag, 2011, pp. 50–60.
- [29] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, “Network-aware operator placement for stream-processing systems,” in *Proceedings of the 22nd International Conference on Data Engineering*. IEEE, 2006, pp. 49–49.
- [30] G. Wang, T. E. Ng, and A. Shaikh, “Programming your network at run-time for big data applications,” in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks*. ACM, 2012, pp. 103–108.
- [31] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, “Network-aware scheduling for data-parallel jobs: Plan when you can,” in *Proceedings of the ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 407–420.
- [32] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, “Choreo: Network-aware task placement for cloud applications,” in *Proceedings of the Conference on Internet measurement conference*. ACM, 2013, pp. 191–204.
- [33] T. Hoefer and M. Snir, “Generic topology mapping strategies for large-scale parallel architectures,” in *Proceedings of the International Conference on Supercomputing*. ACM, 2011, pp. 75–84.
- [34] P. Sack and W. Gropp, “Faster topology-aware collective algorithms through non-minimal communication,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2012, pp. 45–54.
- [35] A. Zheng, A. Labrinidis, P. H. Pisciuneri, P. K. Chrysanthis, and P. Givi, “PARAGON: Parallel architecture-aware graph partition refinement algorithm,” in *Proceedings of the 19th International Conference on Extending Database Technology*, 2016, pp. 365–376.
- [36] D. Ajwani, S. Ali, and J. P. Morrison, “Graph partitioning for reconfigurable topology,” in *Proceedings of the 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 836–847.
- [37] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel, “Graph: Heterogeneity-aware graph computation with adaptive partitioning,” in *Proceedings of the 36th International Conference on Distributed Computing Systems*. IEEE, 2016, pp. 118–128.

- [38] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, “Improving large graph processing on partitioned graphs in the cloud,” in *Proceedings of the 3rd ACM Symposium on Cloud Computing*. ACM, 2012, pp. 3:1–3:13.
- [39] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John, “Data partitioning strategies for graph workloads on heterogeneous clusters,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 56.
- [40] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao, “Heterogeneous environment aware streaming graph partitioning,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 6, pp. 1560–1572, 2015.
- [41] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, “Mosaic: Processing a trillion-edge graph on a single machine,” in *Proceedings of the 12th European Conference on Computer Systems*. ACM, 2017, pp. 527–543.
- [42] K. T. Pedretti, R. Brightwell, D. Doerfler, K. S. Hemmert, and J. H. Laros, “The impact of injection bandwidth performance on application scalability,” in *Proceedings of the 18th European MPI Users’ Group Conference on Recent Advances in the Message Passing Interface*. Springer-Verlag, 2011, pp. 237–246.
- [43] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, “Performance analysis of high performance computing applications on the amazon web services cloud,” in *Proceedings of the IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, 2010, pp. 159–168.
- [44] H. Kwak, C. Lee, H. Park, and S. B. Moon, “What is Twitter, a social network or a news media?” in *Proceedings of the 19th International Conference on World Wide Web*, 2010, pp. 591–600.
- [45] P. Boldi and S. Vigna, “The WebGraph framework I: Compression techniques,” in *Proceedings of the 13th International World Wide Web Conference*. ACM, 2004, pp. 595–601.
- [46] P. Boldi, M. Rosa, M. Santini, and S. Vigna, “Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks,” in *Proceedings of the 20th International Conference on World Wide Web*. ACM, 2011, pp. 587–596.
- [47] S. Edunov, D. Logothetis, C. Wang, A. Ching, and M. Kabiljo, “Darwini: Generating realistic large-scale social graphs,” *CoRR*, vol. abs/1610.00664, 2016.
- [48] F. Viger and M. Latapy, “Efficient and simple generation of random simple connected graphs with prescribed degree sequence,” in *Proceedings of the International Computing and Combinatorics Conference*. Springer, 2005, pp. 440–449.

- [49] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, “The anatomy of the Facebook social graph,” *CoRR*, vol. abs/1111.4503, 2011.
- [50] A. Bhatele, G. R. Gupta, L. V. Kalé, and I. Chung, “Automated mapping of regular communication graphs on mesh interconnects,” in *Proceedings of the International Conference on High Performance Computing*. IEEE, 2010, pp. 1–10.
- [51] R. Chen, J. Shi, Y. Chen, and H. Chen, “Powerlyra: Differentiated graph computation and partitioning on skewed graphs,” in *Proceedings of the 10th European Conference on Computer Systems*. ACM, 2015, pp. 1:1–1:15.