UNIVERSITY OF CALIFORNIA

SANTA CRUZ

LOAD BALANCING LARGE DISTRIBUTED GRAPHS FOR THE ACTOR GRAPH LIBRARY

A project submitted in partial satisfaction

of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Vincent Titterton

December 2024

The Master's Project of
Vincent Titterton is approved:

Professor Scott Beamer

Professor Lindsey Kuper

Acknowledgments

I would like to express my deepest gratitude to my faculty advisor, Dr. Scott Beamer, for the countless hours he has dedicated to guiding and supporting this work. His extensive expertise in graph computation, performance, and benchmarking has been invaluable to this project. Beyond his technical insight, Scott's encouragement and thoughtful feedback have fostered many productive discussions that greatly enriched my research and learning.

I am also grateful to my faculty reader, Dr. Lindsey Kuper, whose seminar on distributed systems has provided a critical foundation for this work. The knowledge gained from her class has been instrumental in shaping my understanding of distributed communication and data routing, allowing me to approach the problem from a much deeper perspective.

Special thanks go to my fellow graduate student researchers on the Hardware Systems Collective graphs team: Amogh Lankar, Omkar Lankar, and Nishant Khanorkar. Their camaraderie, collaboration, and shared dedication have been crucial to the success of this project. It has been a privilege to work alongside them.

Finally, I would like to thank my parents for their unwavering belief in me over the past six years of my academic journey. Their constant support and encouragement have been my greatest source of motivation.

Titterton, Vincent

Load Balancing Large Distributed
Graphs for the Actor Graph Library

Advisor: Dr. Scott Beamer
Master of Science degree conferred December 13, 2024
Masters Project Report completed December 13, 2024

The Actor Graph Library (AGL), is a distributed graph processing framework, designed to assist developers with the loading, generating, and distributing of large graphs. To distribute graphs among multiple processing elements (PEs), each vertex is assigned to a unique PE (edge-cut partitioning) via a hash function. Unfortunately, this rigid method of vertex placement can lead to an imbalance of edges (and therefore work) among PEs.

We present the Partition Refiner, a distributed method for exchanging vertices between PEs in order to evenly distribute edges. This utilizes our prior research on Stateful Mappers, which unlike simple hash functions, allow flexibility in vertex placement. We explore different implementation parameters for the Partition Refiner, and quantify its performance impact on AGL. Specifically, we see a slight increase in the time required to build and distribute graphs. However, for graphs with imbalanced edges, we see notable performance improvements in kernel applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The Actor Graph Library (AGL), is a distributed graph framework designed to process exceptionally large graphs [1]. As graphs scale, they become unable to fit into the memory of a single computer, and thus, must be distributed between multiple (potentially millions of) processing elements (PEs). This comes with many challenges, but also brings the performance benefits of massive parallelism. To distribute a graph between multiple PEs, it must be partitioned (split up into multiple smaller sub-graphs). AGL uses edge-cut partitioning to to achieve this, meaning each PE has an exclusive set of vertices assigned to it. If two vertices on different PEs have an edge connecting them, this edge is said to be cut, since it crosses partitions.

AGL's current implementation represents undirected graphs with a 64 bit integer ID for each vertex. These IDs are assumed to be densely assigned, ranging from 0 up to the total number of vertices in the graph. To assign vertices to PEs, AGL uses Stateless Mappers (a form of hash function) which take an ID as input and return the PE on which the vertex resides. These are beneficial as they have low latency, are easily scalable to a massive number of vertices/PEs, and use negligible amounts of memory. Due to vertex IDs being densely assigned, it also becomes trivial for these mappers to evenly distribute vertices among PEs. However, not all vertices are the same size. In AGL, each vertex is stored with its set of neighboring vertex IDs, representing each edge emanating from that vertex. Since the

number of edges (degree) is arbitrary for any given vertex, there is no guarantee that a Stateless Mapper will evenly distribute edges between PEs.

For optimal performance of a distributed system, the workload must be as evenly distributed as possible. If one PE is assigned more work than others, it will become a bottleneck for the entire system. There can be orders of magnitude more edges than vertices in graphs, and their number is highly correlated with graph processing time, and cross-PE communication. As such, having an imbalance of edges across PEs, will cause an imbalance in memory and processing power utilization.

It is important to note, that vertex placement can also affect performance in ways unrelated to load balance. Specifically, if vertices which are connected to each other, reside on the same PE, the inter-PE communication can be reduced. This can reduce communication bottlenecks, and in turn, improve graph kernel performance. This is not directly explored in this paper, and we explain why in Section 5.1.

An analysis of current Stateless Mappers approaches shows that the imbalance of edges between PEs increases when degree skew increases and/or when the number of PEs increase. While the degree to which Stateless Mappers suffer from this differs, this is an unavoidable issue for any static mapping approach (assuming no prior knowledge of the graph being loaded).

This means, for AGL to scale to millions of PEs, it cannot fully rely on the mappings of vertices from Stateless Mappers. As a potential solution to this fundamental issue, we present the notion of a Partition Refiner used in conjunction with a Stateful Mapper [2]. A

Partition Refiner is a scalable distributed algorithm that, given an initial partitioning from a Stateless Mapper, determines where vertices should be moved to effectively balance edges. A Stateful Mapper is an extension of a Stateless Mapper, which allows vertices to be moved to reflect the output of the Partition Refiner.

The key insights and contributions of this paper are:

- The introduction of the Partition Refiner, a novel distributed approach to determining where vertices should be moved to balance edges on a distributed graph.
- Performance results and analysis of the Partition Refiner on different graphs.
- Contributions to the source code of Actor Graph Library.

With the introduction of the Partition Refiner, Actor Graph Library is able to achieve nearly perfectly balanced edges across a truly massive number of PEs. This can greatly improve load balance, and is done in a way that can preserve many of the advantages of using a Stateless Mapper. Overall, this eliminates a huge road block for scaling to larger graphs, and improves performance while doing so.

Chapter 2

Background

2.1. Actor Graph Library

The Actor Graph Library (AGL) is a large scale distributed graph framework written in C++, and inspired by the non-distributed graph framework: GAP Benchmark Suite [1] [3]. It is part of Flow-Optimized Reconfigurable Zones of Acceleration (FORZA), a collaboration with Georgia Tech. Most of AGL's basic functionality was developed by Tanuj Gupta and Dr. Scott Beamer. Since then, Amogh Lankar, Omkar Lankar, Nishant Khanorkar, and I (Vincent Titterton), have aided in the maintenance and development of new features for AGL. As of now, AGL v1.0.0 is the most up to date release. The code written for this Masters Project was developed in a fork, and is expected to be included in the next release of AGL.

2.1.1. HClib Actor Framework

AGL is built on the HClib Actor Framework, which provides many benefits for parallel, high performance computing [4]. HClib uses an actor-based model, where each actor encapsulates state and logic. Actors communicate through asynchronous, event-driven messages. This allows for non-blocking program execution, maximizing resource utilization. Abstractions for efficient distributed reductions (a type of global synchronization) are also provided. These blocking functions reduce (add, maximize, minimize, etc) a value which is present on each PE, and return this result to each PE. Distributed reductions efficiently scale to a large number of PEs, since they take $O(\log n)$ time, where

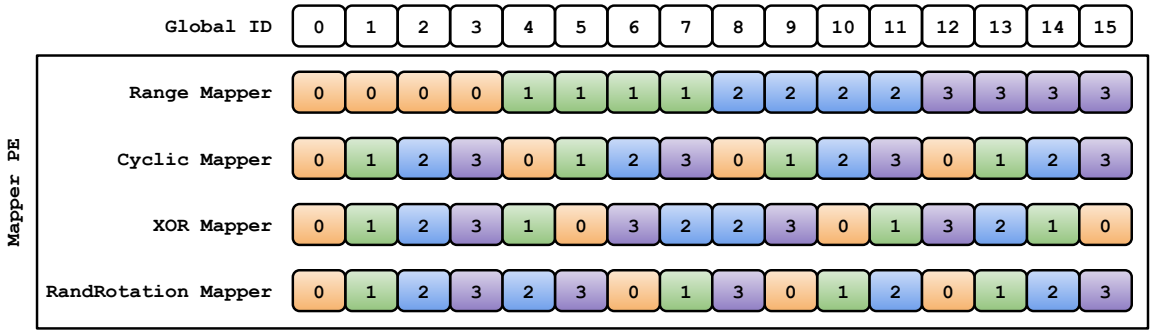
n is the number of PEs. Bulk synchronous parallelism (BSP) is a structured programming model where phases of asynchronous communication (via actors) and computation, are separated by global synchronization steps (via distributed reductions) [5]. BSP simplifies the complexity of managing concurrency, while offering excellent scalability. It is also particularly efficient for many graph algorithms, such as single source shortest path and minimum spanning tree [6]. However, BSP can see large bottlenecks from synchronization overhead if a good load balance is not maintained.

2.1.2. AGL Basic Functionality Actor Graph Library assists in the development of distributed graph processing software as it abstracts many of the general tasks away from the developer. These tasks include: synthetic graph generation, loading graphs from files, and building/distributing graphs. This allows developers to spend their time on the unique aspects of their graph algorithms, without focusing on distributed implementation details. For generated graphs, AGL provides both uniform random (URand) and recursive matrix (RMAT). These graphs can be arbitrarily sized, which allow quick and easy testing of graph kernel applications at any scale. To distribute a graph between multiple PEs, AGL uses edge cut partitioning. This means each PE has an exclusive set of vertices assigned to it. These assignments are determined by a Stateless Mapper (a form of hash function). Figure 2.1 shows how different Stateless Mappers distribute vertices among 4 PEs. When running kernel applications, a command line interface can select different Stateless Mappers, generate/load different graphs, and modulate a number of other features. It is important

to note that the number of PEs is statically determined, and cannot be changed during runtime.

2.1.3. Distributed Graph Representation Each vertex in a graph has a unique 64 bit integer ID. These IDs are assumed to be densely assigned, ranging from 0 up to the total number of vertices in the graph. To store edges, AGL uses a condensed sparse row (CSR) representation for its adjacency matrix. In this representation, each vertex has its own row, and each row contains a sorted list of neighboring vertex IDs. This allows for constant time access to any given vertex ID. Finding a particular edge of a vertex, takes $O(\log d)$ time, where d is the number of edges (degree) of the vertex. To utilize CSR for distributed edge cut partitions, each PE must have its own distinct set of vertices stored in a local CSR. We determine which PE each vertex is stored on, via a Stateless Mapper. Since vertex IDs are densely packed, an even distribution of vertices can be easily guaranteed. However, we want to maintain dense packing of rows, so vertex IDs can no longer be used to index rows. To remedy this, we introduce the notion of a localID. LocalIDs are used to index rows of local CSRs, and are only meaningful for vertices local to a given PE. The task of translating between (global) vertex IDs and localIDs, is also fulfilled by the Stateless Mapper.

Figure 2.1: Visualization of different Stateless Mappers



Overall, Stateless Mappers enable super low latency access to vertex placements, while evenly distributing the vertices among any number of PEs. However, there is no guarantee that stateless mappers will evenly distribute edges among PEs. The Stateless Mapper interface utilized by developers, is shown in Table 2.1.

2.2. Stateless Mapper Limitations One major reason for load imbalance on AGL's kernel applications, is an imbalance of edges across PEs. A PE with significantly more edges than others, will spend more time processing, becoming a bottleneck for the whole system. To optimally balance load, we want to minimize the processing time on the slowest PE. To relate this to edges, we introduce the metric: edge load factor. Edge load factor is defined as the ratio of the maximum number of edges on a PE, to the average number of edges on a PE. The edge load factor will always be at least 1, and the closer to 1 it is, the more evenly distributed the edges are.

The choice of Stateless Mapper can impact the edge load factor of different graphs. If the graph has a correlation of vertex ID to degree, then a poorly chosen Stateless Mapper will reflect this pattern with an increased edge load factor. This is similar to how a hash map can get excessive collisions when a hash function is poorly chosen for the keys being added.

Table 2.1: Stateless Mapper interface

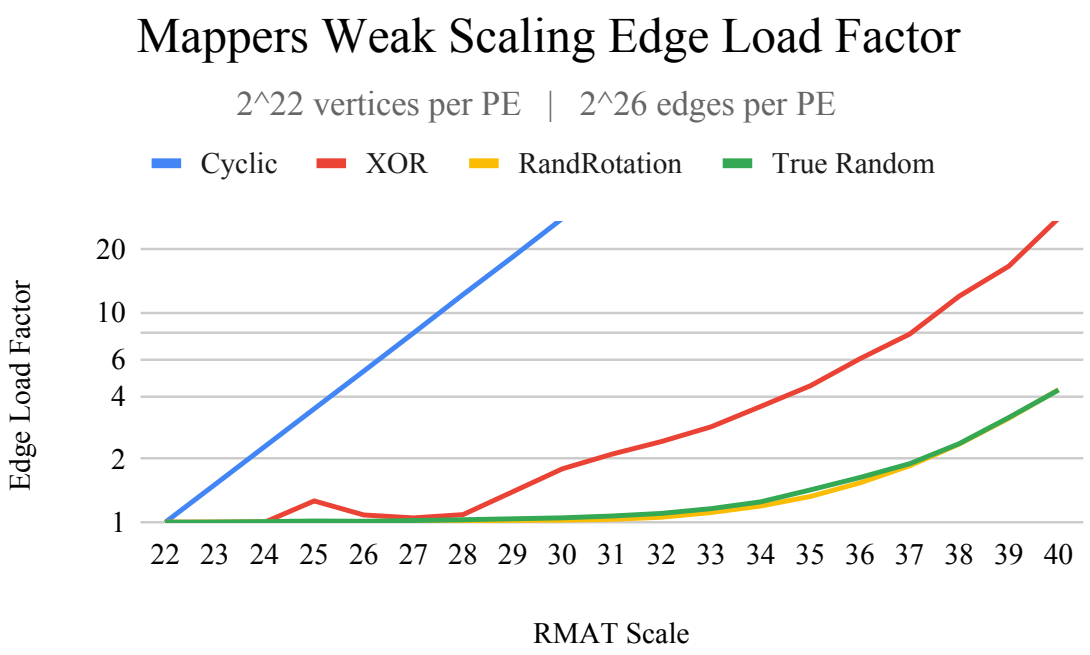
toGlobal	toLocal	toHost
given a local ID, returns the global ID (only applicable for vertices local to this PE)	given a global ID, returns the local ID (only applicable for vertices local to this PE)	given a global ID, returns the vertex's host PE.

One common pattern in input graphs, is where vertex IDs closer to 0, have higher degrees. This inflates the edge load factor (from PE 0) when using the Range Mapper. For many graphs, using the Cyclic Mapper is enough to fully eliminate any patterns of degree in the mappings. However, some graphs contain more intricate patterns such as the Twitter Follower Network (with IDs in multiples of 10), or RMAT (with IDs popcount).

To counteract such ID patterns, Stateless Mappers with different types of scrambling have been tested with good success. In Figure 2.2, we simulate weak scaling of various different Stateless Mappers on an RMAT graph. Some do better than others, however, as the RMAT graph grows larger, no Stateless Mapper is able to maintain an edge factor close to 1. Even using the mt19937 random number generator (graphed as True Random) to randomly assign vertices to PEs, we are not able to avoid this. This is a fundamental issue with using Stateless Mappers for graphs with a high skew in vertex degrees, such as social networks [7]. If we do not have any prior knowledge of the graphs specific vertex degrees, there is no way to maintain a good edge load factor (and therefore good load balance), when processing large social networks.

2.3. Stateful Mappers As we've shown, Stateless Mappers are insufficient for maintaining good edge load factor as graph degree skew and number of PEs scale. However, Stateless Mappers provide exceptionally low mapping latency and memory overhead, things we would like to (mostly) maintain. Thanks to Nishant Khanorkar's Masters Project, different Stateful Mappers, which allow vertices to be moved to different PEs from their initial stateless mappings, have been theorized, developed, and tested [2]. Stateful Mapper initialization is a form of dynamic preprocessing. It happens after static placement of vertices via

Figure 2.2: Stateless Mapper weak scaling edge load factor on RMAT graphs



a Stateless Mapper, but before the algorithm in a kernel application is run. These Stateful Mappers use the same interface (Table 2.1) as Stateless Mappers, and thus, can be substituted in without the need to modify kernel applications. These methods make it possible to modify the edge load factor of graphs, and are a necessary foundation to this work. The Stateful Mappers include:

- 2-Hop Indirect Mapping - Each PE uses a hash table to map its previously local vertices to their new host PE. When messaging the host PE of a moved vertex, two hops must be made: first to the original host (via Stateless Mapper), then to the new host (via hash table lookup). This Stateful Mapper comes with remarkably fast initialization, but comes with a significant slowdown to kernel applications as it requires extra messages to be sent.

- Remote Neighbor Hash Table - Each PE uses a hash table to map all moved vertices, that neighbor one of its local vertices, to their new host PE. This requires more time to initialize than 2-Hop Indirect Mapping, since the hash tables are much larger. Hash table lookups also take more time, but the lack of extra messages still allows Remote Neighbor Hash Table to outperform 2-Hop Indirect Mapping in kernel applications.

- 3-Member Compressed Sparse Row Graph - Each neighbor vertex in the CSR is stored with its host PE's ID. This requires a significant amount of time to initialize, since the CSR grows in size. It has remarkably fast mapping speed, but still can come with a slight slowdown for kernel applications due to each edge taking up more cache.

- Partial Relabeler - The vertex IDs of moved vertices are modified so that the original Stateless Mapper reflects the desired partitions. This requires a significant amount of time to initialize, but notably, does not have any mapping overhead in kernel applications compared to the Stateless Mapper. It does, however, require translating vertex IDs for I/O operations (slowing down some kernel applications).

Each Stateful Mapper has its own unique trade offs, but they all share the trait of being more efficient (in runtime and/or initialization time) when less vertices are moved from their initial stateless mappings. 2-Hop Indirect Mapping, Remote Neighbor Hash Table, and 3-Member Compressed Sparse Row Graph were developed as part of Nishant's Masters Project. The Partial Relabeler was recommended as future work, and later developed by me (Vincent Titterton). These Stateful Mappers (or a subset of them) are expected to be included in AGL's next release.

2.4. Multiway Number Partitioning The problem this paper aims to solve is: given an initial partitioning of vertices (from a Stateless Mapper) how can we decide where vertices need to be moved to effectively minimize the edge load factor. This task of partitioning vertices to balance edges, can be generalized to the multiway number partitioning problem (optimization version), an NP hard problem [8]. For a given number k, this is to partition a set of integers into k subsets of as equal size as possible. Despite being NP hard, this problem has many effective approximation algorithms which are able to quickly find nearly optimal solutions in practice. This is because, as the problem size grows, the number of possible partitions grows exponentially, giving most possible subset sums many different solutions [9]. For inputs similar to the degrees of large graphs' vertices, approximation algorithms should have more than enough flexibility to create highly balanced partitions.

One high quality approximation algorithm for the multiway number partitioning problem is the Largest Differencing Method (LDM) (also known as the Karmarkar-Karp Heuristic) [10]. To test the potential of partition approximation algorithms on large graphs, we implemented LDM on vertex degrees from a uniform random graph of scale=18 degree=11, with k=1000 (representing 1000 PEs). We found that the resulting subsets consistently differed from the mean by at most 1 or 2 edges. We then ran this experiment again, by first randomly partitioning the vertices into 1000 subsets. Then, in each subset, we randomly removed vertices until its total number of edges was below the mean. Then for each subset, we combined the remaining edges into a single number before partitioning. As such, each partition started out with a number slightly below the mean edges, and LDM only had the flexibility to move around a small subset of the total vertices. Surprisingly, LDMs

performance was barely worse than the initial results. This is a key insight going into the development of the Partitioner Refiner. It means that only a small percentage of the vertices need to be moved from an initial random partition to effectively minimize edge load factor. If we are able to replicate this in AGL, this means Stateful Mappers can be particularly efficient when refining partitions. Unfortunately, LDM is not a distributed algorithm, and therefore is not suitable for AGL.

Graphs with highly skewed degree distribution, like social networks, are the cause of major edge imbalances in initial (Stateless Mapper) partitions. However, for the same reason the partitions are imbalanced, they should become even easier to balance by moving a small percentage of vertices. This is because PEs off from the mean by a lot can send/receive the few high degree vertices in the graph, and PEs which are close to the mean, have many low degree vertices to work with.

Chapter 3

Partition Refiner

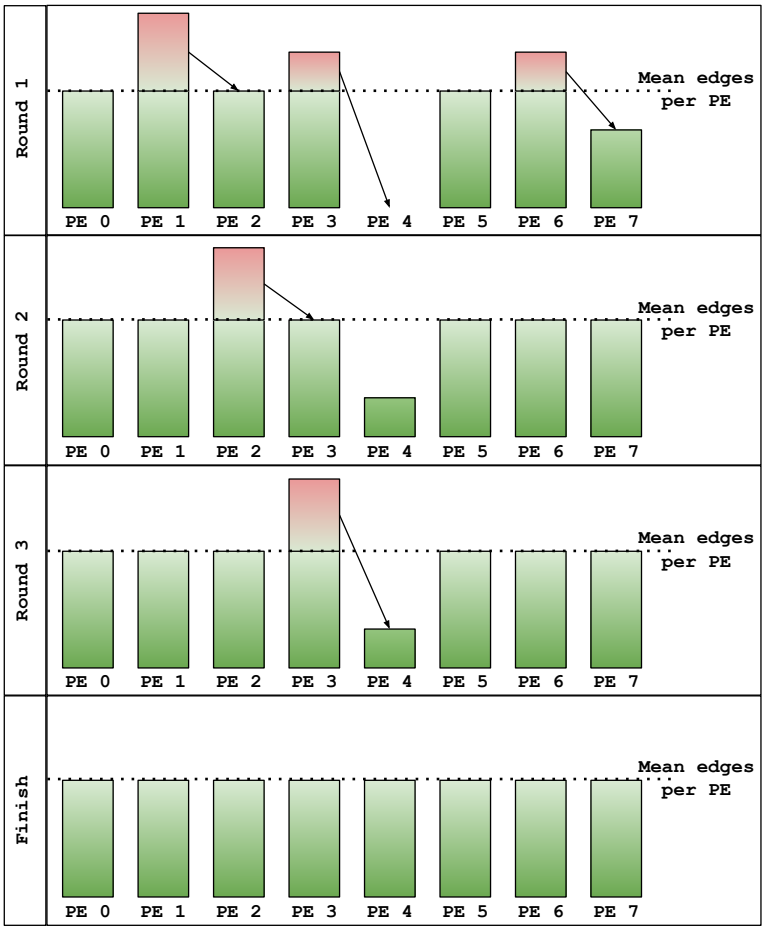
The partitions created by Stateless Mappers effectively and efficiently distribute vertices between PEs, however, these partitions can have potentially large imbalances in their number of edges (resulting in edge load factors significantly greater than 1). We introduce Partition Refiner as a distributed algorithm that, given an initial partition (produced by a Stateless Mapper), determines how vertices should be moved to minimize the edge load factor. This is intended to be paired with a Stateful Mapper, to actually move the vertices during a dynamic preprocessing phase. Regardless of how we move vertices, the PE with the most edges, must have at least the mean edges per PE, rounded up to the nearest integer. We can quickly compute this number via distributed reduction, and will refer to it as the target number of edges.

3.1. Simple Partition Refiner The general idea behind the Partition Refiner is: PEs with more edges than the target should send their extra edges to PEs with less edges than the target. To facilitate this exchange of edges between PEs, we will build an overlay network with a ring topology. This means that each PE has exactly 1 successor (PE that it will be sending to), and 1 predecessor (PE that it will be receiving from). In each synchronous round of the algorithm, all PEs which have more edges than the target will send their extra edges to their successor. Once all PEs are at or below the target number of edges, the algorithm terminates, and the resulting partitioning has the minimum possible edge

load factor (approximately equal to 1). Notably, this algorithm follows the BSP approach, with the caveat that many PEs will go partially or completely unutilized. The only global synchronization is determining whether or not all PEs are at or below the target number of edges between each round (via a singular distributed reduction). Thus, it is highly scalable to a large number of PEs. An example of this algorithm is shown in Figure 3.1.

We obviously cannot split up the edges of individual vertices (due to AGL's edge cut partitioning), so whole vertices must be exchanged instead of edges. Since vertices may need to be passed through multiple PEs, we've opted to not send edge data at this time

Figure 3.1: A Partition Refiner balancing edges between 8 PEs



(only determining where vertices should be moved, but not actually move them). This is possible since the algorithm only takes into account the total number of edges (degree) of each vertex, not individual edge information. Thus, communication and computation can be greatly reduced by representing vertices as pairs of (vertex ID, degree). We leave the exchange of edges up to a Stateful Mapper (initialized after the algorithm's completion). As a simple optimization, we exclude zero degree vertices from consideration as they have no edges to move.

However, exchanging whole vertices comes with the issue of determining which vertices to send. In our implementation, we use a simple greedy selection algorithm: Repeatedly send the vertex with the highest degree that won't make this PE go below the target number of edges. This algorithm is highly efficient when dealing with a large number of vertices, and usually does a good job at minimizing the number of vertices sent. We can modify this approach to also minimize the number of vertices moved from their original PE, making Stateful Mappers more efficient. We do this by keeping vertices received from other PEs separate. We prioritize sending these already moved vertices by selecting from them first. Only afterwards, we select from vertices which originated from this PE (if necessary).

Unfortunately, greedy selection cannot guarantee reaching the target number of edges exactly. In the case a PE remains above the target after greedy selection, we opt to send slightly too many edges by sending the remaining vertex with the lowest degree. As before, vertices will keep being passed around until all PEs are at or below the target number of edges. However, there is now a possibility that the Partition Refiner will never terminate. To assist such cases, we define a tolerance value which is initialized to zero, and slowly increases

as the algorithm runs without making progress. To give this extra tolerance to PEs, we simply add it to the target number of edges when selecting vertices to send. Tolerance gives the Partition Refiner extra flexibility for vertex placement, at the cost of a slightly higher edge load factor. In practice, social networks rarely, if ever, require a tolerance above zero to terminate. For graphs of even degree distribution (such as URand), we observe an increased need for tolerance, since PEs have a limited rage of vertex degrees to send. Even then, The Partition Refiner usually terminates with zero tolerance when PEs have a large number of vertices to select from.

3.2. Dropout Mechanism After selecting (and excluding) vertices to send, it would be nice if PEs at or above the target number of edges, could be prevented from being sent any more vertices in the future. Figure 3.5 (a) and (b) give a visual representation of this optimization. After sending this final group of vertices, these PEs would be excluded from any further communication. We refer to these PEs as dropped out. Notably, among the PEs and vertices which are not dropped out, the mean number of edges per PE never increases. Also, since this happens after selecting vertices to send, the dropped out PEs will have no more edges than the target plus tolerance. This more targeted routing approach will increase the efficiency of communication, especially as the algorithm approaches completion. It also eliminates the issue of a would-be dropout PE, receiving vertices, and then dipping below target number of edges after selecting vertices to send on the next round. To implement this, we split each round into 3 stages:

- Vertex Selection stage

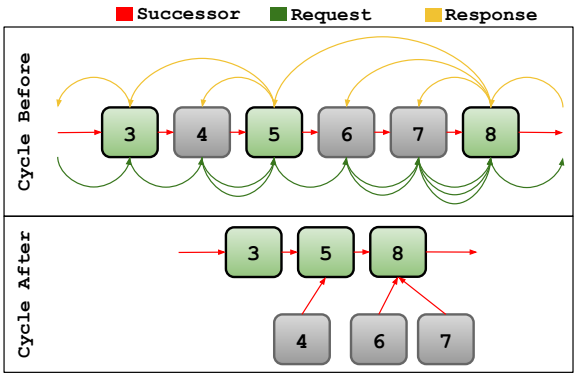
- Cycle Update stage
- Vertex Send stage

The Vertex Selection stage uses the previously described greedy selection technique, however, vertices are not actually sent in this stage. Instead, they are set aside (and excluded from this PE's edge count), only to be sent in the Vertex Send stage. In the Cycle Update stage (figure 3.2), PEs first determine if they will drop out (if they have at least the target number of edges). If a PE's successor is to drop out, it will update its successor to the next non-dropped out PE in the cycle. Notably, dropping out PEs maintain a successor (but not a predecessor) in the cycle, since they still need to send vertices this round. To do the cycle update, each PE sends a request to its successor. When receiving a request, one of two actions are taken:

- If the receiving PE is going to drop out, it will forward this request to its successor.
- If the receiving PE is not going to drop out, it will reply to the original sender (becoming the sender's new successor).

In the worst case, the Cycle Update stage can result in a request being forwarded θ(number of PEs) times. However, this is highly unlikely given a randomized initial partition (especially in the case of power law graphs, where we expect to see most PEs having fewer edges than the mean). This highlights the importance of using an appropriate Stateless Mapper for the initial partitioning.

Figure 3.2: Cycle Update stage communication



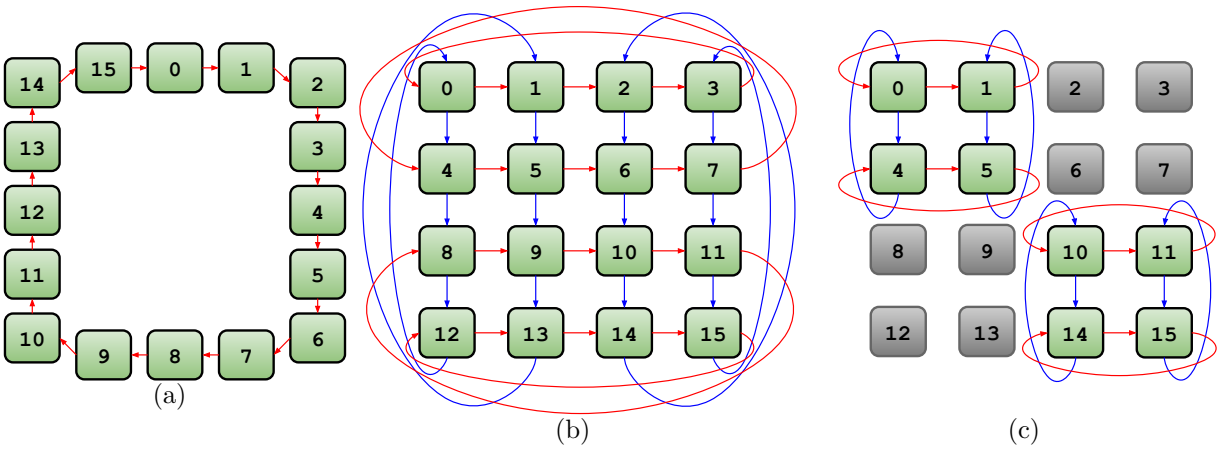
3.3. Multi-Dimensional Communication Imagine a situation where PE 0 has many more edges than the target, and all other PEs have slightly fewer edges than the target. With the current implementation, the vertices will need to be sent through each PE consecutively, meaning that PE n will not receive any vertices until round n. This example is exaggerated, but the same type of inefficiency exists even on randomized initial partitions (especially for power law graphs since the majority of PEs will have an edge count below the mean). This is a fundamental issue with only having one successor PE. Thus, the natural way to fix it is by giving each PE multiple successors.

One way to implement multiple communication paths in parallel computing systems, is called a torus network. A torus can be 1 or more dimensions, and it allows each PE to send data to d other PEs (where d is the torus' dimension). Figure 3.3 (a) and (b) show 1 and 2 dimensional torus networks between PEs. A higher dimensional torus is particularly useful for routing messages since it creates short routing paths between any two PEs with a minimal amount of wiring. Many supercomputers, such as IBM's Blue Gene/L, use torus networks for messaging [11].

Notably, the communication pattern of the previously described Partition Refiner (Section 3.1) is a 1 dimensional torus, and thus could potentially be generalized to work on higher dimensional tori. Unfortunately, the dropout optimization is not compatible with this idea. The highly connected nature or a torus will quickly degrade as PEs are removed. In fact, it can even become disconnected as shown in Figure 3.3 (c).

Alternative to having multiple small cycles for each dimension of a torus, we could have a full sized cycle, consisting of all PEs, for each dimension. The dropout optimization would work exactly the same as before, just happening on each cycle. In order to maximize connectivity and minimize the number of hops between any two PEs, these cycles must each have a unique order. We have settled for creating each cycle (besides the first) in a deterministic pseudo random fashion. This results in a greatly reduced distance between any two PEs in the communication graph. Since each communication cycle is randomly ordered in relation to the others, the probabilistic connectivity is also maintained after PEs drop out. Unlike the supercomputers which use torus networks, the Partition Refiner does not need to route vertices to any particular PE, it just needs to distribute them as evenly as possible. As such, random communication cycles become a viable option. We show an example of a two-dimensional communication cycle in Figure 3.4. Figure 3.5 demonstrates how the dropout mechanism and multidimensional communication work together to more quickly distribute vertices. When enabling the dropout mechanism on a 1D topology, we see a 3-10x speedup (with better speedups for more PEs). Increasing the topology dimension, gives us an additional 1.5-2x speedup (also with better speedups for more PEs). For the rest of this work, we use both of these optimizations because of their effectiveness.

Figure 3.3: (a) 1-D Torus (b) 2-D Torus (c) Disconnected 2-D Torus



However, this now introduces the question: if there are multiple successor PEs, which PE should each vertex be sent to? We've tested two possible heuristics for this:

- send cyclically - on each round send the first vertex to the successor on cycle 1, the second vertex to the successor on cycle 2, and so on.
- send randomly - pick a random successor for each vertex.

We currently default to cyclic, even though both methods work well and have similar performance. However, successor selection could likely be improved with future research.

Figure 3.4: Two Dimensional Partition Refiner Successor graph on 16 PEs

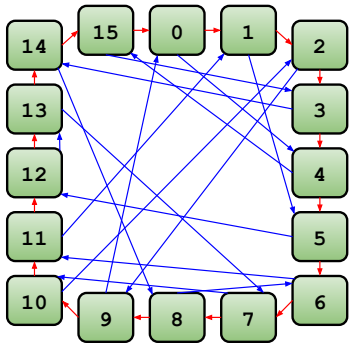


Figure 3.5: Console Outputs of Partition Refiners on RMAT Scale 25 with Cyclic Mapper

We compare Partition Refiners: (a) 1-dimensional, (b) 1-dimensional with dropout mechanism, and (c) 4-dimensional(cyclic routing) with dropout mechanism. In these charts, each PE is represented as a column. The column entries define the number of vertices sent (by this PE) on each round. Once a PE has dropped out, its future column entries are left blank. The tolerance for each round is shown on the far right.

```
Partitioning (1-D Communication):
    PE:  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | Tolerance
 Round:-------------------------------------------------------------------------------------------------------------------------------|----------
    0: 1e5 1e4 1e4   0 1e4   0   0   0 1e4   0   0   0   0   0   0   0 1e4   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    1:   0 1e5 1e4 1e4   0 1e4   0   0   0 1e4   0   0   0   0   0   0   0 1e4   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    2:   0   0 1e5 1e4 1e4   0 8e3   0   0   0 8e3   0   0   0   0   0   0   0 8e3   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    3:   0   0   0 9e4 1e4 1e4   0 868   0   0   0 869   0   0   0   0   0   0   0 870   0   0   0   0   0   0   0   0   0   0   0 | 0
    4:   0   0   0   0 9e4 1e4 1e4   0 868   0   0   0 325   0   0   0   0   0   0   0 328   0   0   0   0   0   0   0   0   0   0 | 0
    5:   0   0   0   0   0 8e4 1e4 7e3   0 868   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    6:   0   0   0   0   0   0 7e4 1e4 7e3   0 868   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    7:   0   0   0   0   0   0   0 7e4 1e4 7e3   0 868   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    8:   0   0   0   0   0   0   0   0 7e4 1e4 7e3   0 870   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    9:   0   0   0   0   0   0   0   0   0 7e4 1e4 6e3   0  10   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
   10:   0   0   0   0   0   0   0   0   0   0 7e4 1e4 5e3   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
   11:   0   0   0   0   0   0   0   0   0   0   0 6e4 1e4 2e3   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
   12:   0   0   0   0   0   0   0   0   0   0   0   0 5e4 7e3 546   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
   13:   0   0   0   0   0   0   0   0   0   0   0   0   0 4e4 4e3   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
   14:   0   0   0   0   0   0   0   0   0   0   0   0   0   0 3e4 1e3   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
   15:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 3e4 1e3   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
   16:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 3e4 1e3   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
   17:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 3e4 1e3   0   0   0   0   0   0   0   0   0   0   0   0 | 0
   18:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 3e4 1e3   0   0   0   0   0   0   0   0   0   0   0 | 0
   19:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 3e4 1e3   0   0   0   0   0   0   0   0   0   0 | 0
   20:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 2e4 671   0   0   0   0   0   0   0   0   0 | 0
   21:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 2e4  61   0   0   0   0   0   0   0   0 | 0
   22:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 1e4   0   0   0   0   0   0   0   0 | 0
   23:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 1e4   0   0   0   0   0   0   0 | 0
   24:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 1e4   0   0   0   0   0   0 | 0
   25:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 8e3   0   0   0   0   0 | 0
   26:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 6e3   0   0   0   0 | 0
   27:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 3e3   0   0   0 | 0
   28:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 2e3   0   0 | 0
   29:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 1e3   0 | 0
   30:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 305 | 0
   31:   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
  Total Rounds: 31
  Partition Time:    0.51388

Partitioning (1-D Communication with Dropout Mechanism):
    PE:  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | Tolerance
 Round:-------------------------------------------------------------------------------------------------------------------------------|----------
    0: 1e5 1e4 1e4   0 1e4   0   0   0 1e4   0   0   0   0   0   0   0 1e4   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    1:             1e5     1e4   0   0     1e4   0   0   0   0   0   0     1e4   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    2:                       1e5   0         8e3   0   0   0   0   0         8e3   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    3:                           9e4             869   0   0   0   0             870   0   0   0   0   0   0   0   0   0   0   0 | 0
    4:                                         8e4   0   0   0                 328   0   0   0   0   0   0   0   0   0   0   0 | 0
    5:                                             6e4   0   0                     0   0   0   0   0   0   0   0   0   0   0 | 0
    6:                                                 4e4   0                     0   0   0   0   0   0   0   0   0   0   0 | 0
    7:                                                   3e4                     0   0   0   0   0   0   0   0   0   0   0 | 0
    8:                                                                       2e4   0   0   0   0   0   0   0   0   0   0   0 | 0
    9:                                                                           2e4   0   0   0   0   0   0   0   0   0   0 | 0
   10:                                                                               1e4   0   0   0   0   0   0   0   0   0 | 0
   11:                                                                                   1e4   0   0   0   0   0   0   0   0 | 0
   12:                                                                                       8e3   0   0   0   0   0   0   0 | 0
   13:                                                                                           6e3   0   0   0   0   0   0 | 0
   14:                                                                                               3e3   0   0   0   0   0 | 0
   15:                                                                                                   2e3   0   0   0   0 | 0
   16:                                                                                                       1e3   0   0   0 | 0
   17:                                                                                                           310   0   0 | 0
   18:                                                                                                                 0   0 | 0
  Total Rounds: 18
  Partition Time:    0.14170

Partitioning (4-D Communication with Dropout Mechanism):
    PE:  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | Tolerance
 Round:-------------------------------------------------------------------------------------------------------------------------------|----------
    0: 1e5 1e4 1e4   0 1e4   0   0   0 1e4   0   0   0   0   0   0   0 1e4   0   0   0   0   0   0   0   0   0   0   0   0   0   0 | 0
    1:             2e4     4e3   0   0     843 887   0   0 6e3  14   0     891 4e3   0   0   0   0   0 901   0 3e3 4e3   0   0   0 | 0
    2:                       5e3   0             330 1e3         10             0   0   0   0   0     711         2e3   0   0   0 | 0
    3:                           250                                     164  60   0   0   0                     1e3  27   0 | 0
    4:                                                                       313   0   0                         173 | 0
    5:                                                                           65   0                             | 0
    6:                                                                                0                             | 0
  Total Rounds: 6
  Partition Time:    0.09150
```

Chapter 4

Evaluation

We evaluate the performance of the Partition Refiner when implemented in AGL. Specifically, we determine the following:

- The optimal choice of parameters for multi-dimensional communication.
- The expected changes to graph partitions.
- The expected increase to graph build time.
- The expected runtime speedup for kernel applications.

We run our experiments on a single-socket AMD EPYC 9554 (64 cores and 128 threads) with a base clock of 3.1 GHz. Our system has 256 MB of shared L3 cache and 768 GB of RAM. For graphs, we have chosen:

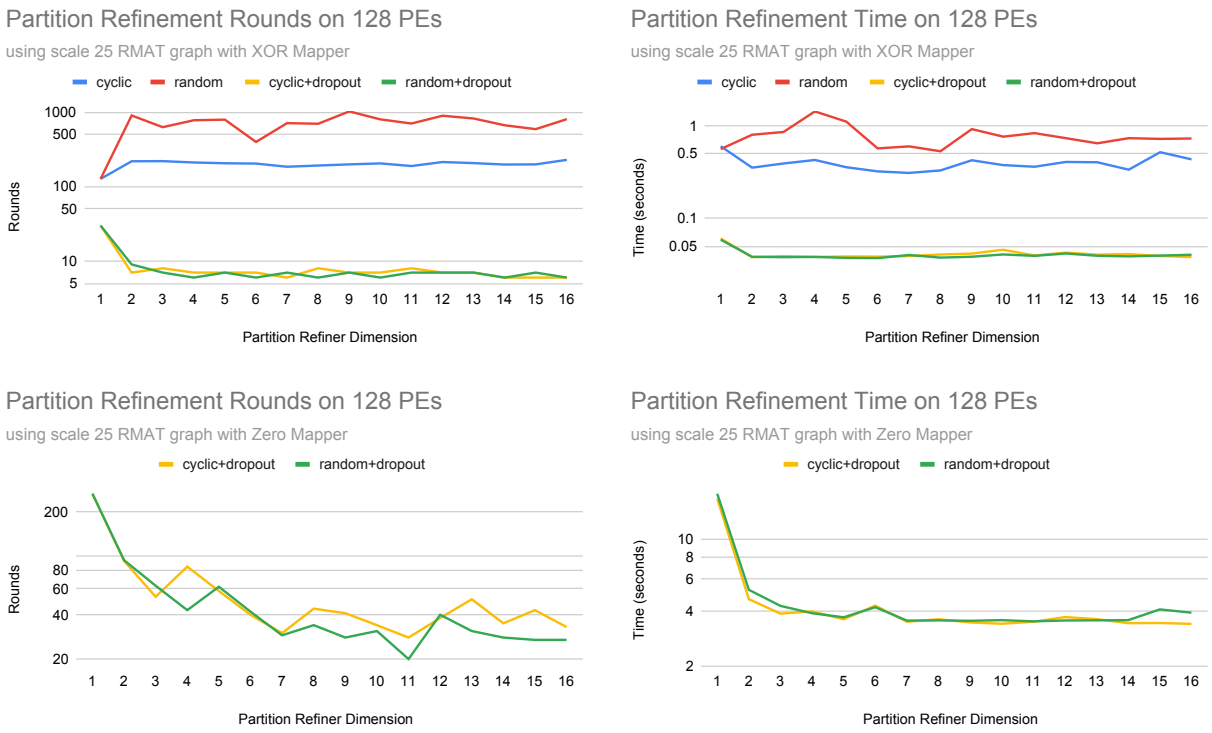
- Uniform Random (URand): An arbitrarily sized generated graph with low degree skew.
- Recursive Matrix (RMAT): An arbitrarily sized generated graph with power law properties similar to social networks [12]. We use $a = 0.57, b = 0.19, c = 0.19, d = 0.05$.
- Twitter Follower Network: a real-world social network with 61,578,415 vertices and 1,202,513,046 edges [13].

For kernel applications, we have chosen a push-direction Jacobi-style page rank using 8 iterations (PR), and a top-down breadth first search running 8 trials from randomly selected starting vertices (BFS).

4.1. Multi-Dimensional Parameters To determine the optimal way to implement multi-dimensional communication for the Partition Refiner, we measure how the algorithm's time and number of rounds change as the communication dimension increases. We compare both cyclic and random routing choice, and test with and without the dropout mechanism. For these tests, we have chosen to use 128 PEs (one for each thread on our chip) on a scale 25 RMAT graph (33,552,896 vertices and 523,599,618 edges). We first initialize the partitions with the XOR Stateless Mapper, resulting in a vertex load factor of 1.00047, and an edge load factor of 1.90093. This edge load factor could be reduced by using a better Stateless Mapper, but it is well suited for this evaluation as it simulates the extra edge load factor we would expect a well chosen Stateless Mapper to produce as AGL scales to extremely large social graphs. However, to truly test the limits of the Partition Refiner, we also initialize partitions with the impractical Zero Mapper. This Stateless Mapper maps all vertices to PE 0, and results in a vertex load factor and edge load factor of 128.0.

From the Partition Refinement results on 128 PEs (Figure 4.1), we see that the dropout mechanism is massively beneficial to performance. As such, it is implicitly enabled in all further tests of the Partition Refiner. When using dropout mechanism, we see a clear reduction in both rounds and time as the dimension increases. This effect is most pronounced when changing from 1 to 2 dimensions, and it seems to plateau after 4 dimensions for the XOR Mapper, and 6 dimensions for the Zero Mapper. Unfortunately, it is inconclusive whether

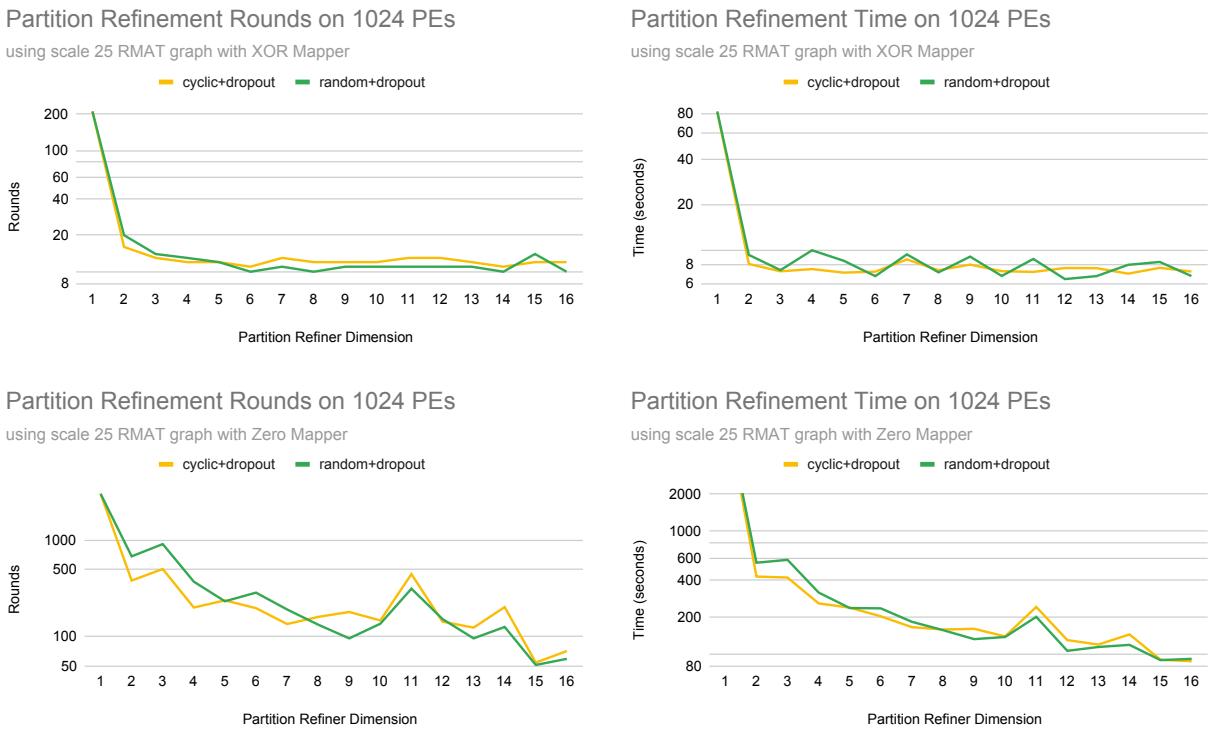
Figure 4.1: Sweep Partition Refiner dimension with 128 PEs



cyclic or random routing is better when using the dropout mechanism. However, without the dropout mechanism, cyclic routing clearly outperforms random routing.

To see how the effect of dimension changes as PEs increase, we rerun the tests with 1024 PEs (simulating 16 PEs on each physical core of our chip). Using the same scale 25 RMAT graph (with 33,552,896 vertices and 523,599,618 edges), XOR results in a vertex load factor of 1.00159, and an edge load factor of 3.54468. Zero results in a vertex load factor and edge load factor of 1024.0 (Figure 4.2). Even with an 8x increase to the number of PEs, the time to refine XOR partitions still seems to plateau after around 4 dimensions. However, with the Zero Mapper, we observe a benefit of increasing the partition dimension all the way to our maximum tested value of 16. We conclude that larger dimensions see a benefit when there are a small number of PEs (initialized with more edges than the target) which need to

Figure 4.2: Sweep Partition Refiner dimension with 1024 PEs



distribute their vertices among a large number of total PEs. While not as extreme as with the Zero Mapper, we expect such scenarios to occur when loading highly skewed power law graphs across a large number of PEs. Given the scale at which we run our other test, we chose to stick with 4 dimensions for the rest of the evaluation. We also choose to stick with cyclic routing, as it is unclear whether cyclic or random routing is superior.

4.2. Imbalanced Graph Performance As AGL scales to large graph sizes and numbers of PEs, we expect graphs with skewed degree distributions to have an elevated edge load factor when partitioned from a Stateless Mapper. Unfortunately, we have limited ability to test graphs at this scale with the hardware available to us. Also, relying on this type of imbalance limits our ability to isolate variables in experimentation (as edge load factor will change with the number of PEs and graph size). One way around this, is to increase the

vertex load factor of the stateless partitions. We do this by creating a cyclic type Stateless Mapper which gives PE 0 a controlled amount of extra vertices. For example, we can give PE 0, 3 vertices for every 2 vertices each other PE has (we refer to this as a 1.5x vertex imbalance). This Stateless Mapper enables us to achieve arbitrary edge load factors which remain mostly constant when changing the number of PEs or graph size. It also allows us to arbitrarily change the edge load factor while keeping number of PEs and graph size constant. The following experiments are useful, but not fully representative of what we would actually expect from AGL, since well designed Stateless Mappers will always result in a vertex load factor very close to 1 (unlike in these experiments).

4.2.1. Strong Scaling We would like the see the effect increasing the number of PEs has on the partition refiner. We isolate this variable by keeping the problem size constant (strong scaling) and by keeping the edge imbalance approximately constant. We do this by using a vertex imbalance of 1.5x which is able to maintain an edge load factor of approximately 1.5 (Figures 4.3, 4.4, and 4.5). We see that the Partition Refiner is able to perfectly balance edges on all data points to an edge load factor of 1.00000. Building the graph using the Partial Relabeler Stateful Mapper, we see between a 15%-30% slowdown which is mostly constant, if not slightly trending downwards, as the number of PEs increase. However, we do notice a clear trend that both the build slowdown and Partition Refinement time decrease as the graph gets more degree skew. Specifically, URand, which has very low degree skew, has an approximate 25% slowdown for Partial Relabeling, and a 3% slowdown when only adding on partition refinement. This is in contrast with the Twitter Follower Network which

only has 15% slowdown with relabeling and a 0.8% slowdown without. The kernel speedups for BFS and TC are slightly less than the edge imbalance that the Partition Refiner fixes, but still substantial.

4.2.2. Sweep Graph Scale We would like the see the effect increasing the size of a graph has on the partition refiner. We isolate this variable by keeping the number of PEs constant and by keeping the edge imbalance approximately constant. We do this by using a vertex imbalance of 1.5x which is able to maintain an edge load factor at approximately 1.5 (Figures 4.6 and 4.7). Since we must modify the size of graphs here, we omit the real-world Twitter Follower Network. We see that the Partition Refiner is able to perfectly balance edges (1.00000 edge load factor) on almost all data points, with the exception of small

Figure 4.3: Strong scaling with 1.5x vertex imbalance on uniform random graph

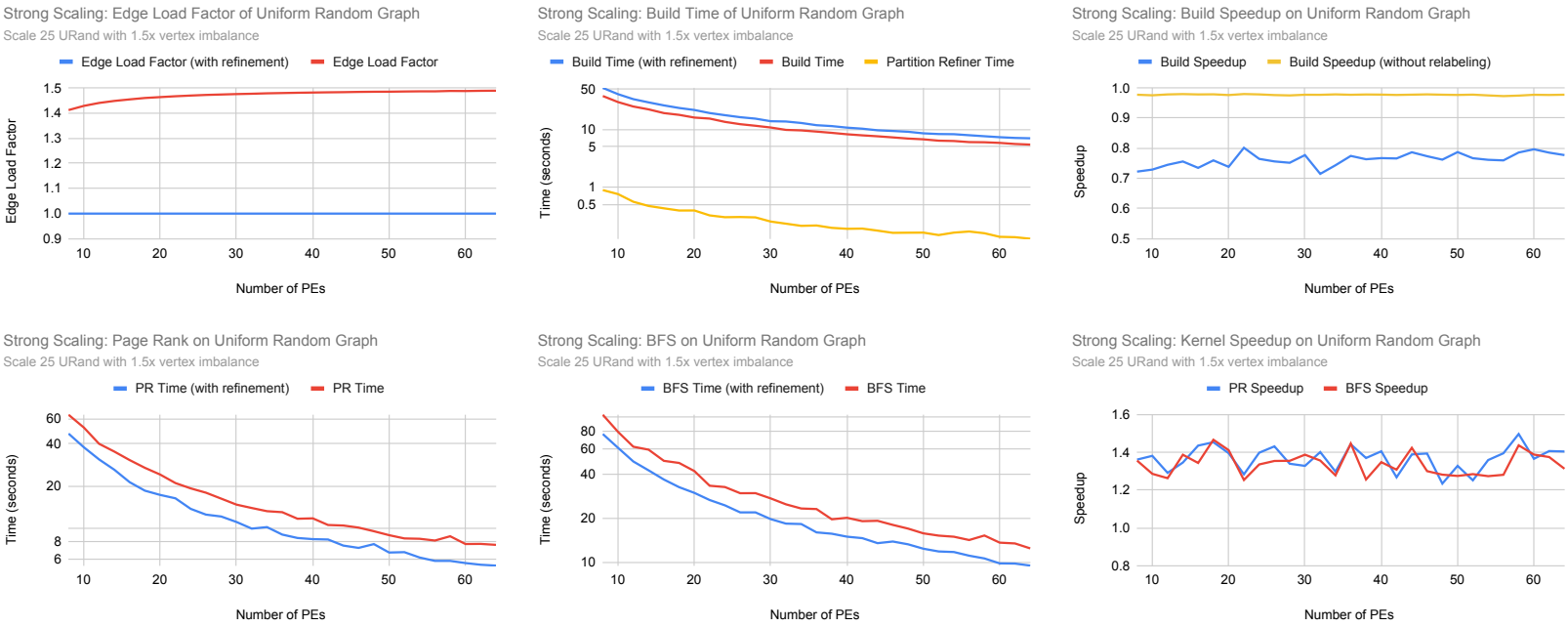
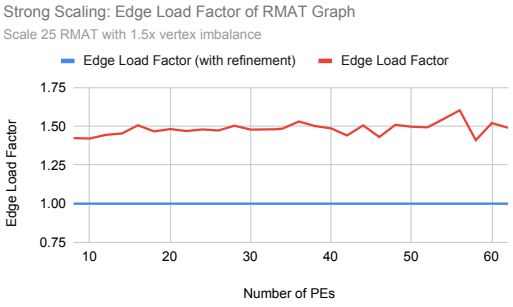
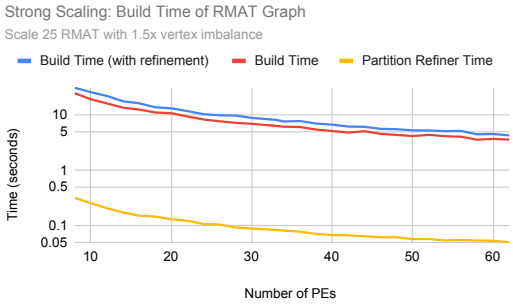
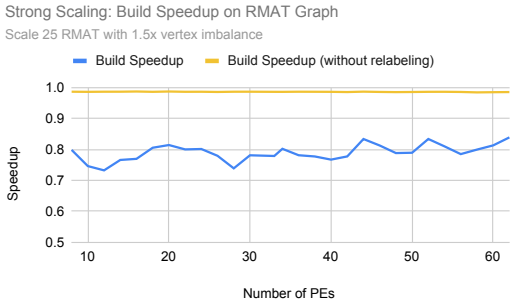
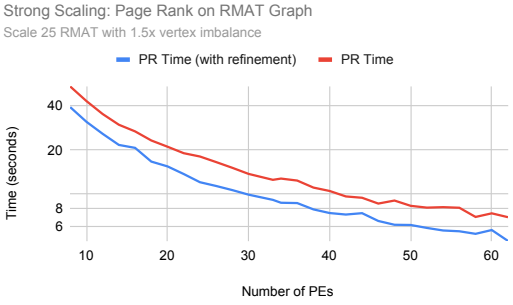


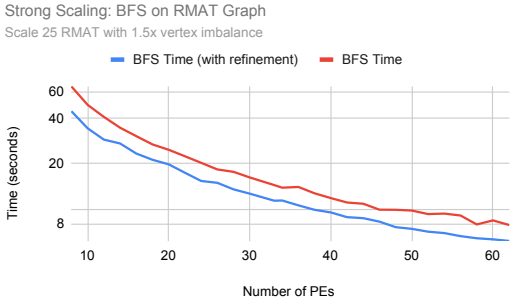
Figure 4.4: Strong scaling with 1.5x vertex imbalance on RMAT graph











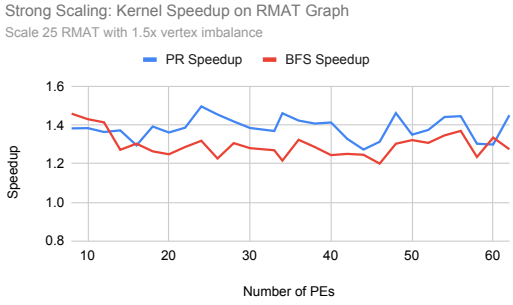
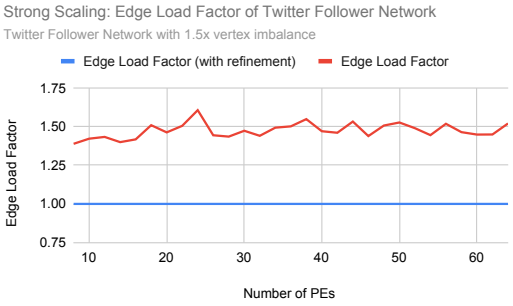
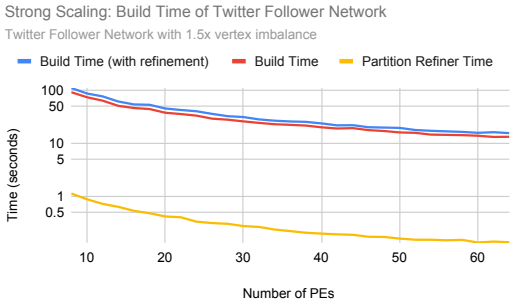
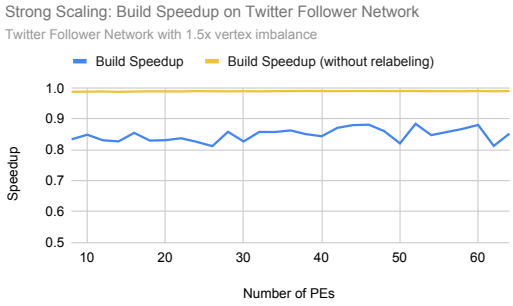
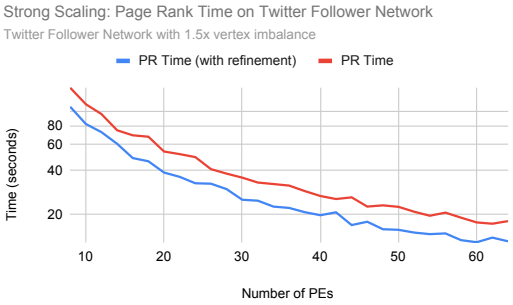


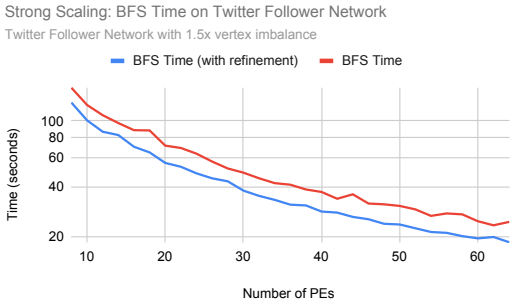
Figure 4.5: Strong scaling with 1.5x vertex imbalance on Twitter Follower Network

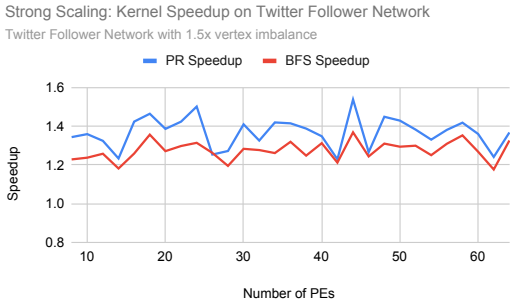












URand graphs. Specifically, URand scale 16, 17, and 18, have resulting edge load factors of 1.00018, 1.00001, and 1.00001 respectively. While the difference from 1.00000 is negligible, this demonstrates how the Partition Refiner can struggle when only being able to select from a small number of vertices with low degree skew. Overall, we see decreased build overhead and more consistent (and better) kernel speedups as the graph size increases. These benefits plateau as we reach large graph sizes.

4.2.3. Sweep Vertex Imbalance Lastly, we would like the see the effect increasing the edge load factor of a graph has on the partition refiner. We isolate this variable by keeping the number of PEs constant and by keeping the graph size constant (Figures 4.8, 4.9, and 4.10). We see that the Partition Refiner is able to perfectly balance edges on all data points to an edge load factor of 1.00000. Here we observe similar speedup outcomes

Figure 4.6: Sweep graph size with 1.5x vertex imbalance on uniform random graph

Graph has 2^{SCALE} vertices.

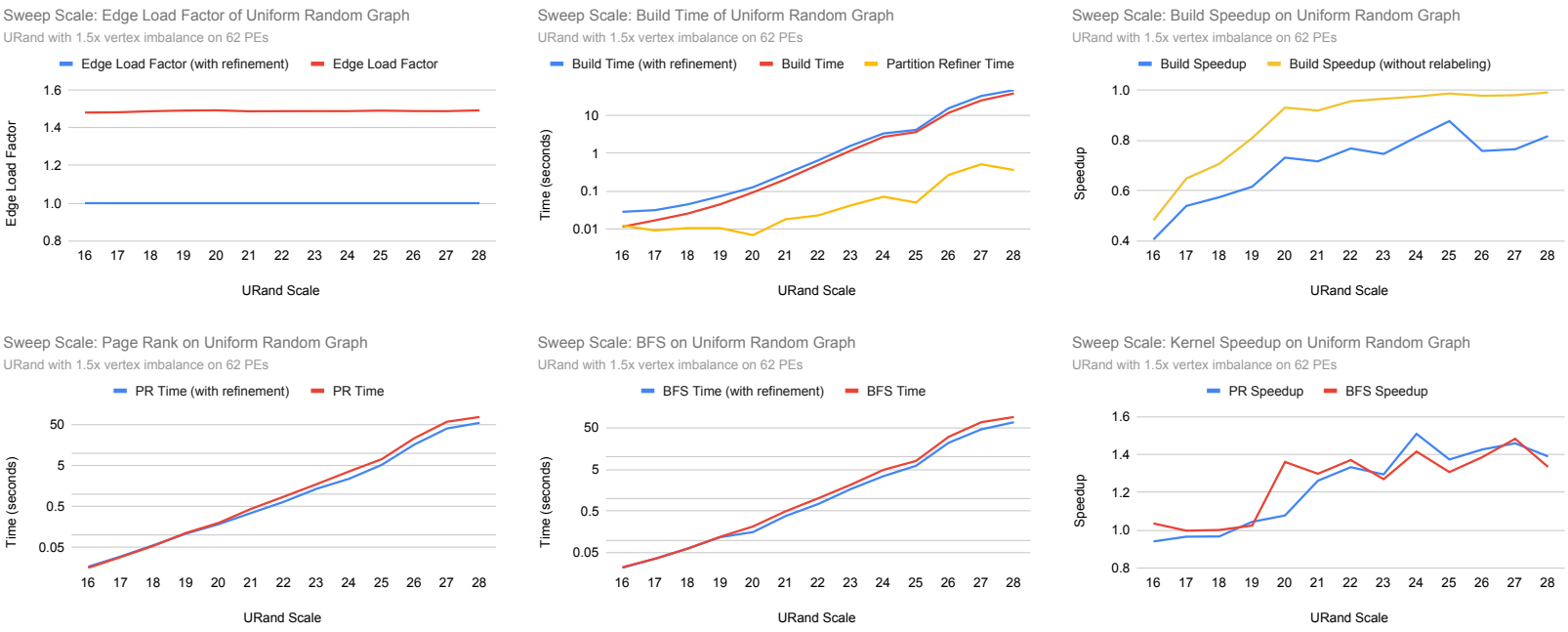
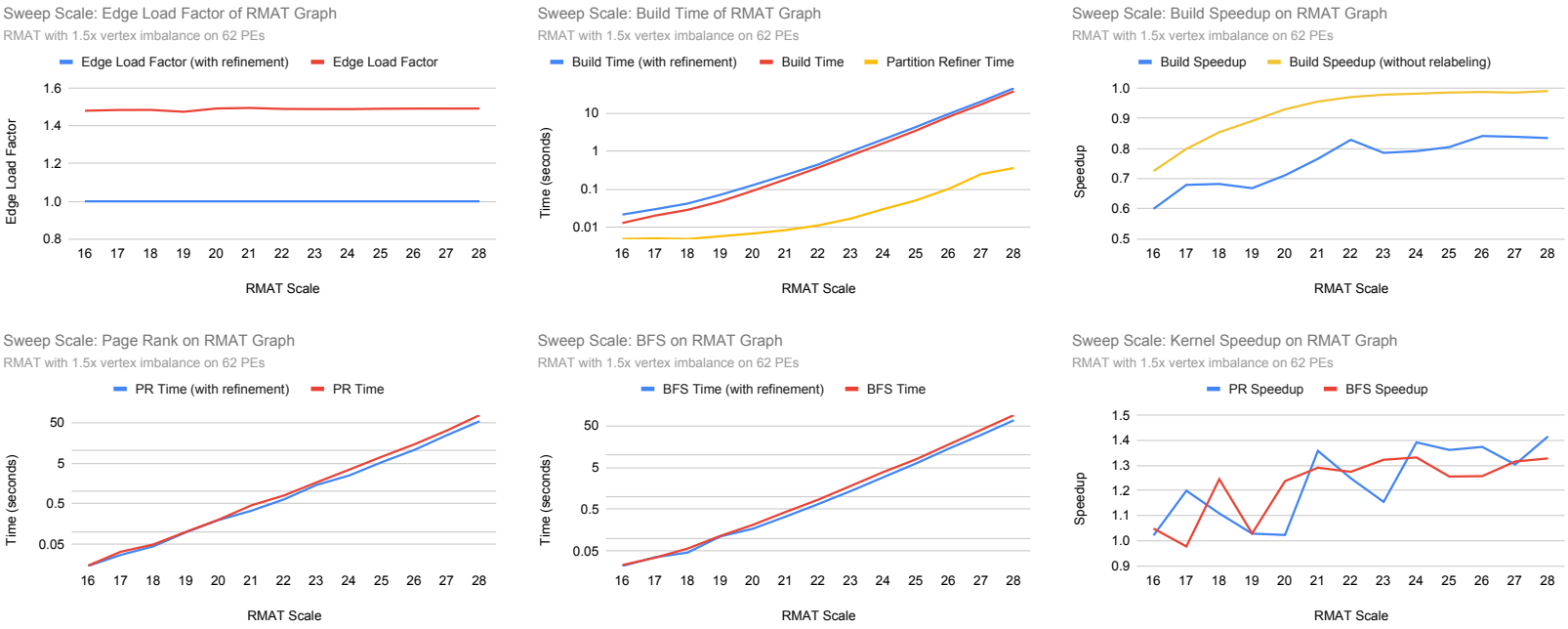


Figure 4.7: Sweep graph size with 1.5x vertex imbalance on RMAT graph

Graph has 2^{SCALE} vertices.



to strong scaling. However, we are able to more accurately see the runtime improvements of partition refinement as edge imbalance grows. Specifically, we see between a 50%-75% translation from edge imbalance of graphs above 1, to their kernel application slowdown. Meaning if we go from an edge load factor of 1.0 to 1.4 we would expect to see a 20%-30% slowdown. We also see that the build time increases as the edge imbalance does. However, only on the URand graph with low degree skew, does the partition time significantly increase with increased edge load factor. Lastly, we observe the kernel runtimes of the edge balanced partitions increase, as the initial vertex load factor increases. This is because, the Partition Refiner leaves the vertex load factor mostly unchanged, and thus, the imbalance of vertices is causing this slowdown. As such, the results in all of Section 4.2 (using inflated vertex load factors), are not fully applicable to what we would expect of AGL when using a well

designed Stateless Mapper.

Figure 4.8: Sweep vertex imbalance on uniform random graph

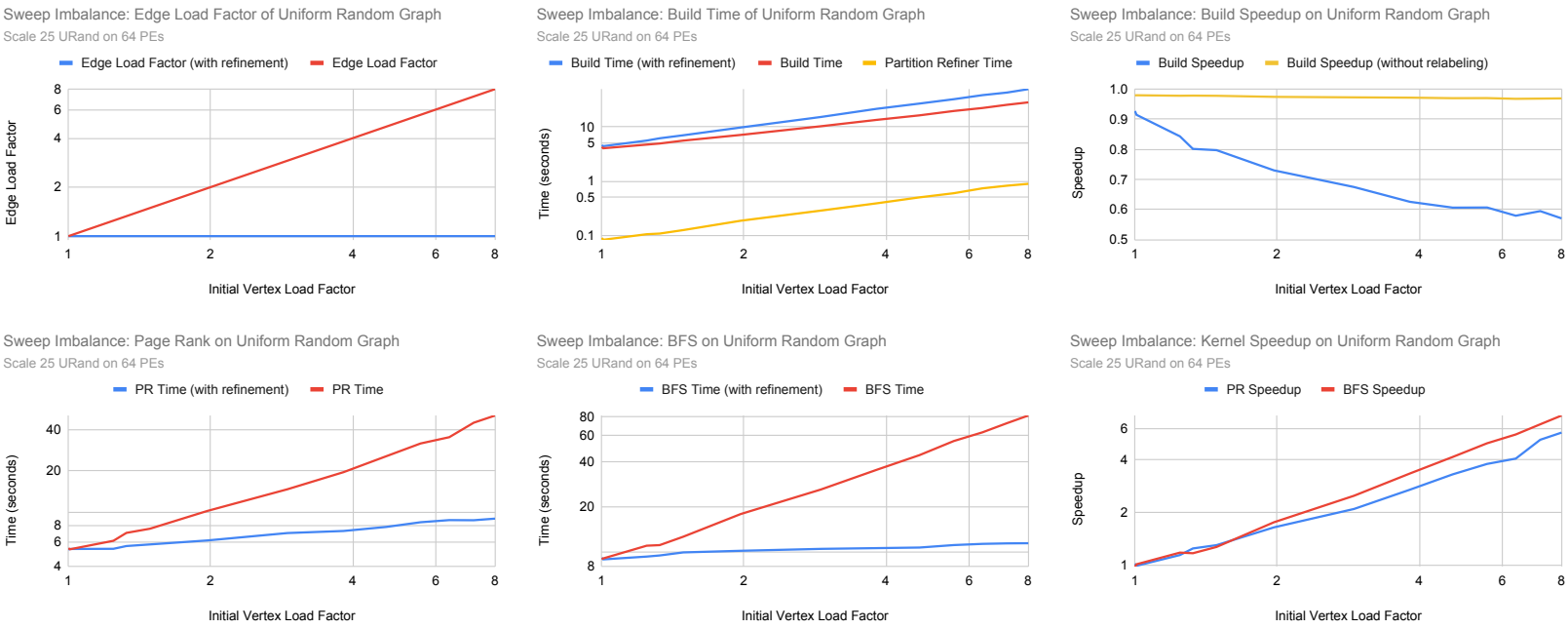


Figure 4.9: Sweep vertex imbalance on RMAT graph

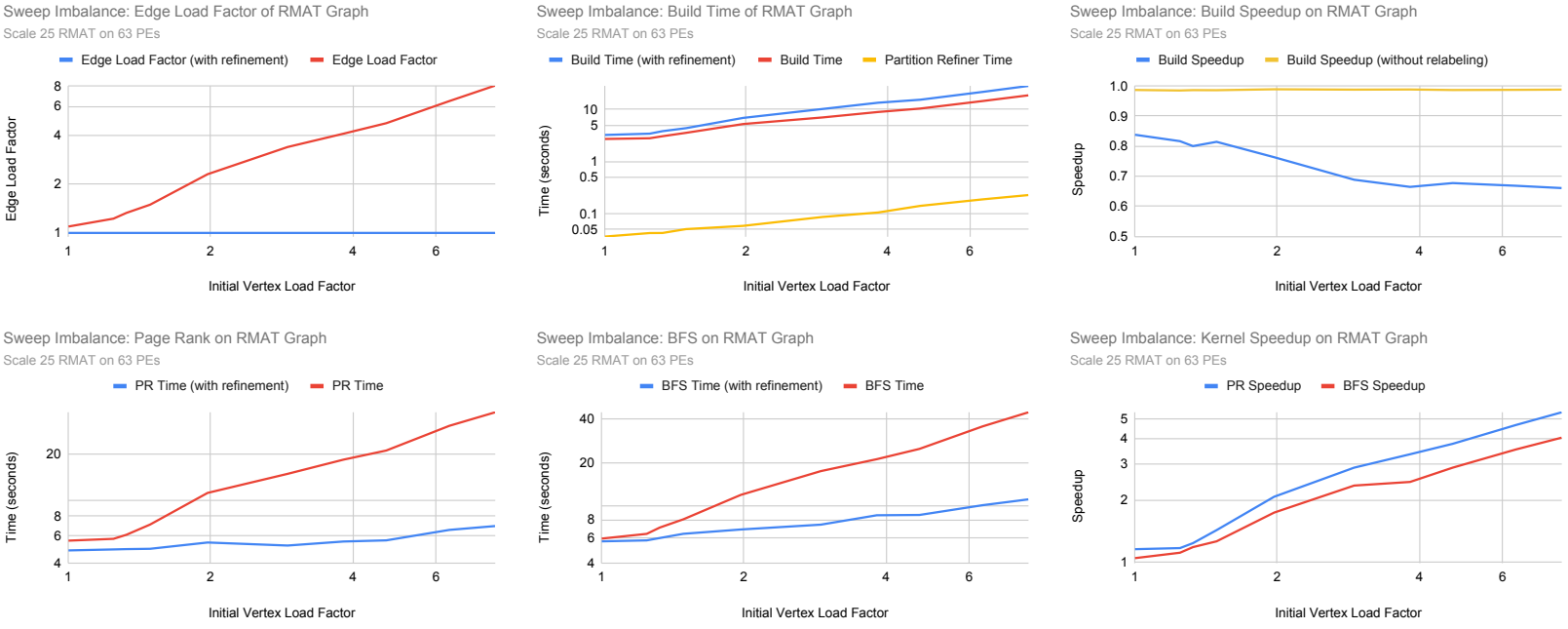
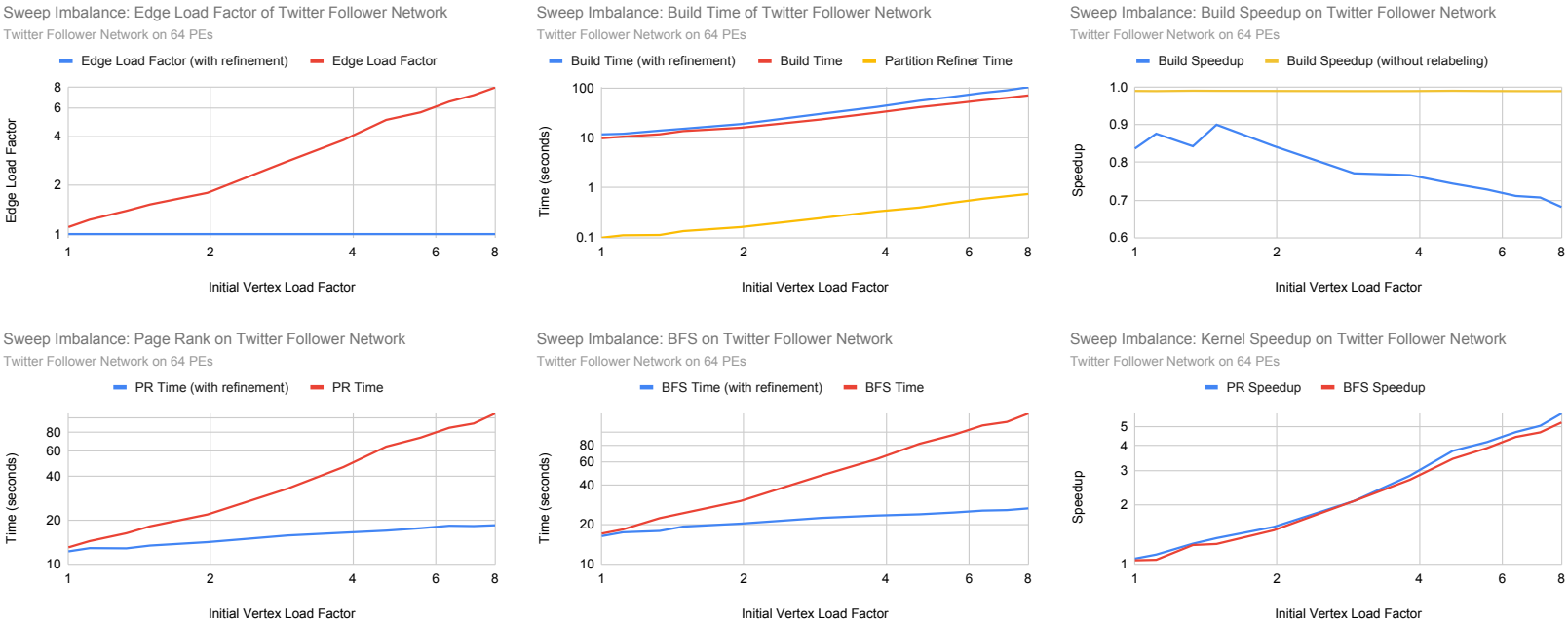


Figure 4.10: Sweep vertex imbalance on Twitter Follower Network



4.3. AGL Scaling Performance The imbalanced graph performance in the previous section is insightful to how the Partition Refiner scales with specific parameters. However, these measurements are much more useful if they can be related to what we expect from AGL as it scales. To best capture this information, we test up to 1024 PEs. This requires hardware simulation (multiple PEs per physical core) with the hardware available to us, which greatly impacts runtime. As such, we choose to limit our measurements to the following aspects of the Partition Refiner:

- the change to edge load factor
- the change to vertex load factor
- the number of vertices moved
- the number of edges moved

- the number of vertices updated (defined as vertices with at least 1 moved neighboring vertex)

We are specifically interested in how these aspects change as the number of PEs and/or graph size increases. We have also chosen to exclude the Uniform Random graph from these tests as it will never result in a significant edge imbalance when using a well designed Stateless Mapper.

4.3.1. Strong Scaling We first look strong scaling, which is keeping the problem size (graph) fixed while increasing the number of PEs. For the graphs, we have chosen RMAT scale 25 (33,554,432 vertices and 523,599,618 edges), and the Twitter Follower Network (61,578,415 vertices and 1,202,513,046 edges). To create the initial partitions, we use the RandRotation Stateless Mapper. Then, we determine the new partitions with a 4 dimensional Partition Refiner using cyclic routing. We update the vertex placement using the Partial Relabeler Stateful Mapper.

We plot the Twitter Follower Network results in Figure 4.11. Notably, the edge load factor after the Partition Refiner was 1.00000 from 4 through 512 PEs. However, it spikes to 1.27626 for 1024 PEs. On closer inspection of the partitions, we noticed that the 3 PEs with the most edges, each only had a single vertex. These vertices each have more edges than the mean number of edges for a PE, thus, it is impossible to reduce edge load factor any further. This is a fundamental limitation of AGL's edge cut partitioning. The only way to improve upon this requires splitting the largest vertices across multiple PEs. Because all

but one edges on these 3 PEs needed to be sent to other PEs, we also see a massive spike in vertex load factor (even though the actual value is still very small).

We plot the RMAT results in Figure 4.12. On every number of PEs, we achieve an edge load factor of 1.00000 after using the Partition Refiner. Notably, the vertex load factor remains unchanged since the Partial Relabeler we are using overwrites vertices with 0 degree (which are plentiful on RMAT graphs).

4.3.2. Weak Scaling Now we look at weak scaling, which is growing the problem size, while keeping the problem size per PE fixed. Since we have no accurate way to grow the size of a real world graph, we only test RMAT in this section. We will be sizing RMAT with 262,144 (scale 18) vertices per PE. At the largest (with 1024 PEs), we use RMAT scale 28 with 268,435,456 vertices. To create the initial partitions, we use the RandRotation

Figure 4.11: AGL strong scaling Twitter Follower Network

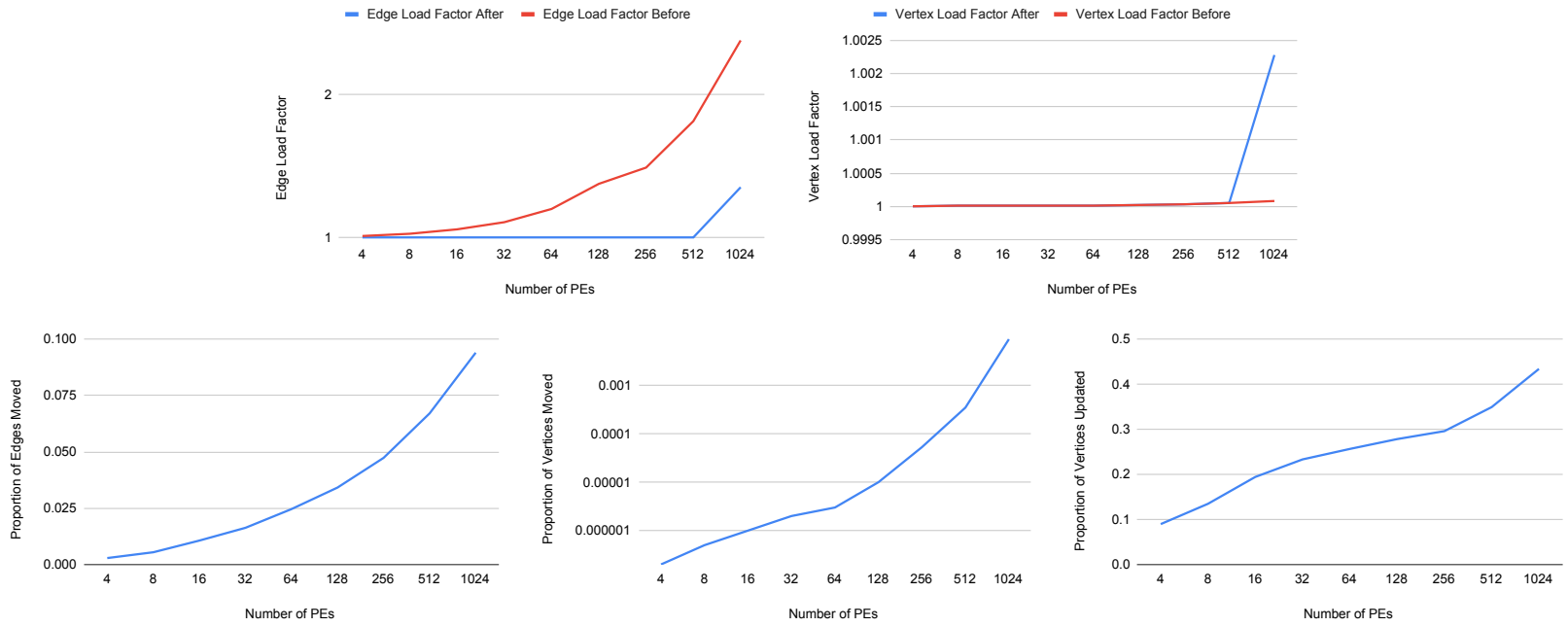
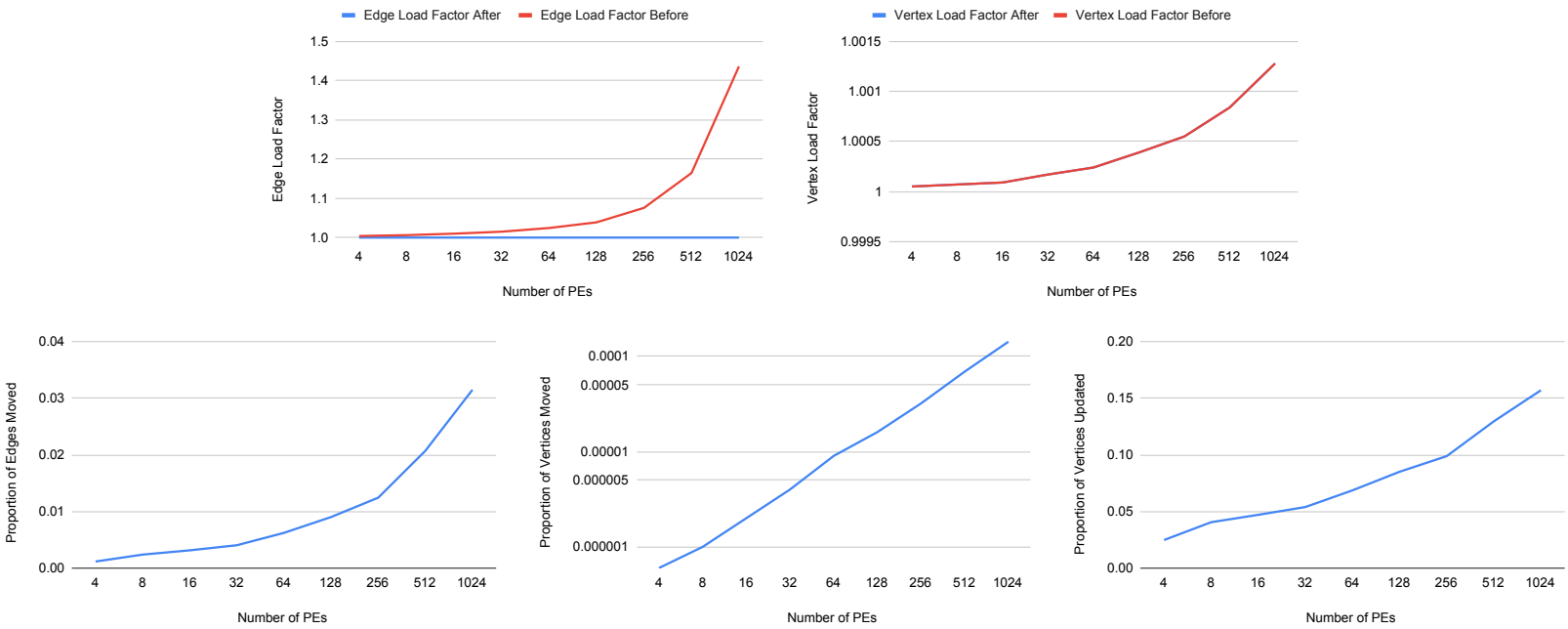


Figure 4.12: AGL strong scaling RMAT



Stateless Mapper. Then, we determine the new partitions with a 4 dimensional Partition Refiner using cyclic routing. We update the vertex placement using the Partial Relabeler Stateful Mapper. We plot these results in Figure 4.13. Once again, on every number of PEs, we achieve an edge load factor of 1.00000 after using the Partition Refiner. As with strong scaling, the vertex load factor remains the same since RMAT graphs have some 0 degree vertices to overwrite. One interesting observation, is that over 80% of vertices are updated (for 8 or more PEs) even though an incredibly small proportion of of vertices and edges actually moved.

For comparison, we rerun the same tests using the Cyclic Stateless Mapper (Figure 4.14). When using a power of two number of PEs like in these tests, partitioning an RMAT graphs with the Cyclic Stateless Mapper will result in an incredibly large edge load factor. Even with incredibly imbalanced initial partitions, we still achieve an edge load factor of 1.00000

Figure 4.13: AGL weak scaling RMAT with RandRotation Stateless Mapper

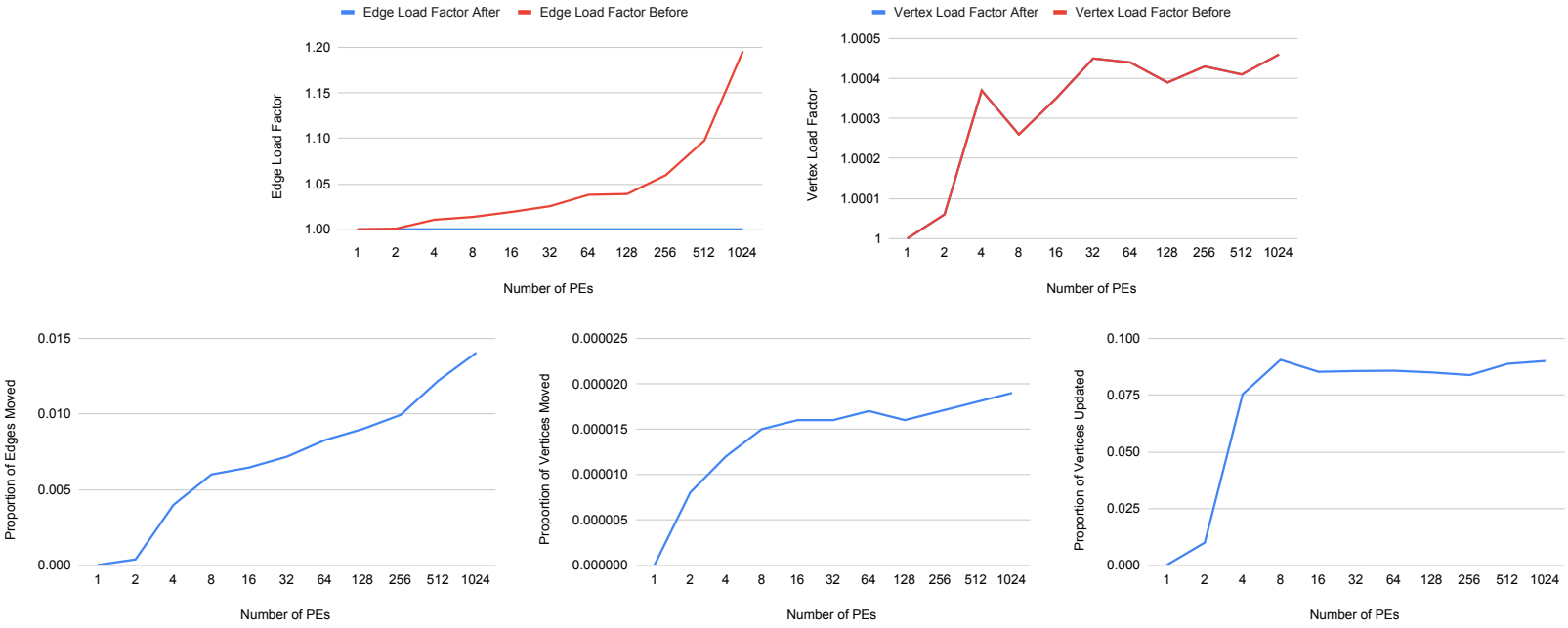
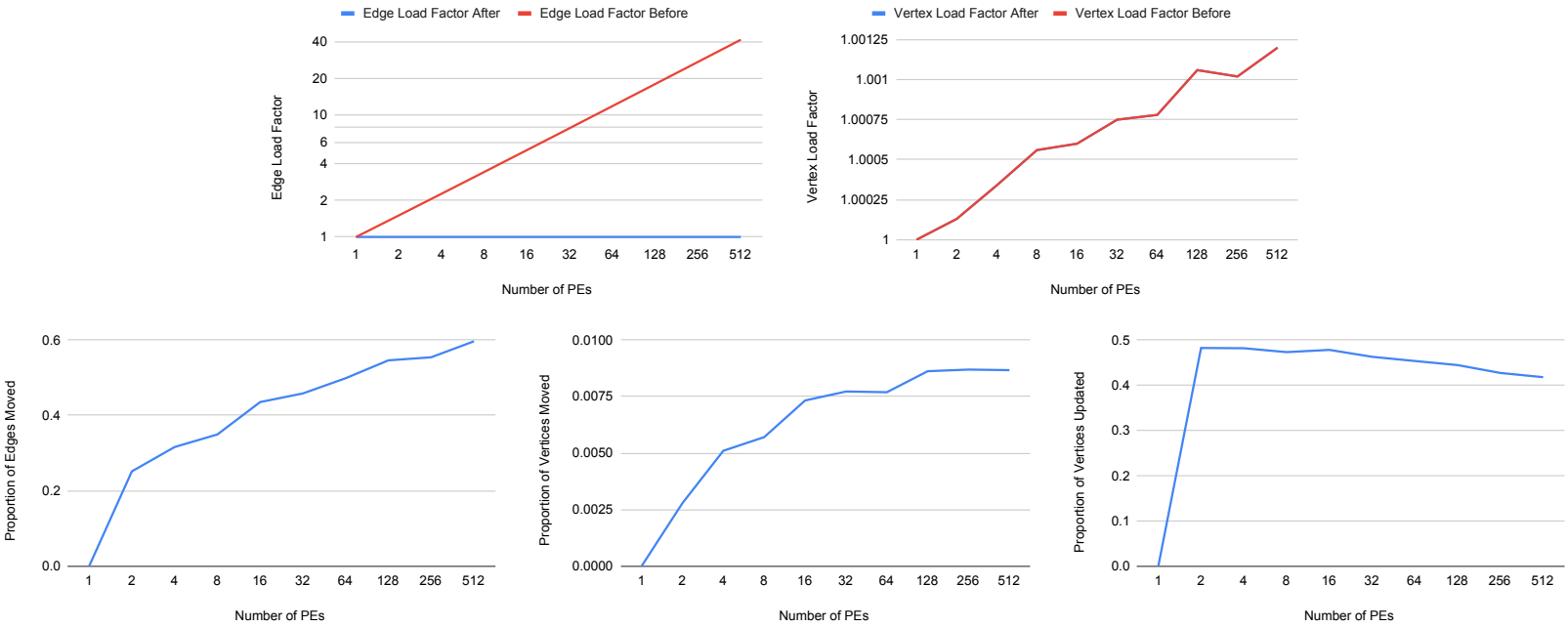


Figure 4.14: AGL weak scaling RMAT with Cyclic Stateless Mapper



on every number of PEs after using the Partition Refiner. As expected, a much higher proportion of edges are moved (almost 60% for 512 PEs). We also see a notable increase in the proportion of vertices that are moved, however, this is still a very small number being less than 1%. Surprisingly, the Partition Refiner requires updating a significantly smaller proportion of vertices (past 8 PEs) compared to using the more balanced initial partitions created by the RandRotation Stateless Mapper. This is likely a result of the vertices which are moved, having lower degree on average. One potential explanation for this, is that in the initial partitions, a much smaller number of PEs will have more edges than the mean (compared to the more balanced initial partitions from RandRotation). Another potential explanation, is that the degree distribution pattern of RMAT (which is exposed by the Cyclic Stateless Mapper to be able to get such imbalanced partitions in the first place) is somehow making the Partition Refiner extra efficient.

Chapter 5

Related Work

5.1. Communication Volume Minimizing Partitioners METIS, a popular graph partitioning program can be used to minimize the communication volume of resulting partitions (in addition to balancing edges and vertices) [14]. The communication volume is a measure of the number of unique PEs each vertex has a cut edge connected to. In graph kernel applications, this metric highly correlates with the amount of inter PE communication required. Using communication volume minimizing partitioners like METIS, requires significant amounts of time for reprocessing. While METIS is not distributed, there are some distributed partitioners, such as JA-BE-JA and DHPV, which can still achieve high quality partitions [15] [16]. Minimizing communication volume can reduce the message passing overhead, speeding up kernel applications. However, we do not directly explore this type of partitioning in this paper for a few reasons:

- Communication volume minimizing partitioners generally will not scale to the size of graphs we intend to use AGL for. If they can (like DHPV), the time required to partition will likely exceed any reduction in graph kernel runtime. For this type of partitioning to be useful, it would likely need to be done statically, unlike the dynamic Partition Refiner.

- Partitioning quality is often sub par for power law graphs like social networks. These partitioners have significantly more impact on topological graphs, such as road networks.

- A partitioning that minimizes communication volume, also has the downside of requiring almost all vertices to change their host PE (from their stateless mappings). Even excluding the time it takes to determine the partitions, moving this amount of vertices (with a Partial Relabeler) comes with an upfront cost approximately 3 times greater than initially loading a graph with a Stateless Mapper.

5.2. Distributed Data Routing The communication patterns of the Partition Refiner are related to (and inspired by) numerous other distributed routing techniques. As mentioned in Section 3.3, torus networks were an inspiration for the Partition Refiner's multidimensional communication. However, the final result is actually more similar to Chord, a distributed hash table (DHT) implementation [17]. Like the Partition Refiner and torus networks, Chord uses a ring structure to arrange nodes (PEs). Each node on the ring is responsible for a range of keys, for which it will store any corresponding values. To allow nodes to be added and removed from the system without disrupting even distribution, key ranges are assigned via random identification numbers with the same size as the keyspace. This type of randomization is very similar to how a multidimensional Partition Refiner uses randomly ordered cycles (rings) in order to prevent degrading connectivity as PEs drop out. To identify the node responsible for a given key, Chord must route queries around the ring until the correct node is found. However, if each node could only route to it's direct successor, Chord would scale very poorly, requiring $O(n)$ message hops, where n is the number of nodes.

To remedy this, each node contains a finger table of size $O(\log n)$, allowing it to route to that many other nodes. The identification numbers of such nodes are exponentially different from the current identification number (modulo keyspace). This allows Chord to identify the node responsible for a given key with only $O(\log n)$ message hops (similar to a binary search). Finger tables have similar motivation to the Partition Refiner's multidimensional communication. A one dimensional Partition Refiner would also require $O(n)$ message hops to route to a given PE. While we haven't proved it, Chord gives us a good reason to believe that the optimal dimension for a partition refiner is logarithmically related to the number of PEs.

Chapter 6

Conclusion

The current implementation of the Partition Refiner effectively balances edges among PEs in AGL. This can improve load balance, and therefore kernel performance, at the cost of some extra time to build and distribute the graph. This trade off can be further modulated by selecting different Stateful Mappers to pair with the Partition Refiner. From our experimental results, we have proven the Partition Refiner effective, and given insight into the particular scenarios that warrant its use. However, there are many opportunities for improvement and future research.

6.1. Partition Refiner Advancements

6.1.1. Adaptive Routing For multidimensional Partition Refiners, PEs have the option to send their vertices to multiple successor PEs. To make these routing decisions, we currently rely on either cyclic or random routing. However, there is a potential for more informed routing decisions based on the number of vertices PEs have. Using the requests and responses in the cycle update stage, we can easily gather this information by slightly increasing the size of each message. Specifically, each PE can know the total number of vertices each of its predecessors needs to distribute that round. Then, it can use this information (in combination with its own distance from the target number of edges) to request some number of vertices from each of its predecessor PEs (via cycle update response). Then, each PE

can route vertices based on some heuristic from the information it receives. With this extra information, it is likely we can, at least slightly, improve the efficiency of the partition refiner. However, we leave the specific heuristics of smart routing up to future research.

6.1.2. Termination Currently, the Partition Refiner uses a tolerance value to aid in termination. This works well for the graphs we have tested, but its effectiveness isn't rigorously proven. As such, further research on termination control is required before this is used in production software.

6.1.3. Dimension Optimization Partition refinement sees a reduction in both rounds and time when multidimensional communication is used. The more PEs there are, the higher the dimension that will be useful. However, unnecessarily increasing the dimension will cause a slowdown in the cycle update stage, without any benefit. One way to optimize for this is to gradually decrease the communication dimension of the Partition Refiner as PEs drop out. This will help maximize performance at all stages of the algorithm. To make this possible, further research must determine a heuristic for the ideal dimension for a given number of PEs and graph size.

6.1.4. Better Vertex Selection Selecting vertices to send is currently done via greedy approach. This is a simple yet effective method, however, there is likely room for improvement. While it is impossible too guarantee PEs getting to the target number of edges exactly, more advanced heuristics can do a better job for this variant of the bin packing problem. However,

it is also important to note that getting as close to the target, while minimizing the number of selected vertices, is not necessarily optimal for the overall algorithm. Sending smaller degree vertices than necessary can give the algorithm more flexibility in future rounds, and potentially avoid requiring a tolerance increase. This trade off is worth exploring with future research.

6.1.5. Better Cycle Ordering Determining the ideal ordering for a given dimension, while taking into account potential dropouts, is a theoretical problem left for future research. Our current solution is to randomize each cycle in relation to each other. This achieves good but sub-optimal connectivity, with the property of keeping connectivity unchanged after dropouts. However, since the cycle orders are statically determined at initialization, there is almost certainly a better approach than randomization.

6.2. Future Work

6.2.1. Balance Vertices Too One of the issues with the Partition Refiner is it only balances edges, but not vertices. Stateless Mapper partitions always have balanced vertices, but during refinement, the balance is slightly degraded. Exploring methods to balance both vertices and edges could slightly improve performance, but would likely require a full redesign of the algorithm. We leave this up to future research.

6.2.2. Vertex Splitting One issue, fundamental to AGL's edge cut partitioning, is having vertices which are too large to fit on a single PE (without causing an edge imbalance). In

fact, we observe this phenomenon when distributing the Twitter Follower Network across 1024 PEs in Figure 4.11. In such cases, the Partition Refiner can no longer help us, and we must, instead, find a way to divide vertices. To implement this in AGL, a hybrid approach could be taken with the help of a Stateful Mapper. This would allow the vast majority of vertices to benefit from the efficiency of edge cut partitioning (and likely stateless mapping), with the exception of splitting the few large vertices. This would enable the Partition Refiner to scale to well past its current limits.

6.2.3. Vertex Cut Partitioning Vertex cut partitioning is an alternative method of distributing graphs to AGL's edge cut partitioning. Instead of uniquely assigning vertices to PEs and cutting edges, it uniquely assigns edges and cuts vertices. Doing so is able to completely circumvent the edge imbalance that often occurs when using edge cut partitioning on power law graphs. In these cases, vertex cut partitioning can significantly outperform edge cut partitioning [18]. However, vertex cut partitioning suffers from the opposite issue as AGL, which is an imbalance of vertices between PEs. As we've shown in our testing, vertex imbalance is a lot less impactful than edge imbalance, but can still make a noticeable difference. One future direction of research, could be to develop a Partition Refiner to balance vertices on distributed vertex cut partitions. It would also be insightful to compare the performance of AGL with partition refinement, to a similar framework which uses vertex cut partitioning.

BIBLIOGRAPHY

[1] T. Gupta, *Actor graph library*, June, 2024. 1, 4

[2] N. K. Khanorkar, *Enabeling flexible data placement in the actor graph library*, June, 2024. 2, 8

[3] S. Beamer, *Gap benchmark suite*, June, 2024. 4

[4] Y. Elmougy, A. Hayashi and V. Sarkar, *Highly scalable large-scale asynchronous graph processing using actors*, in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, pp. 242–248, 2023. DOI. 4

[5] L. G. Valiant, *A bridging model for parallel computation*, *Commun. ACM* **33** (Aug., 1990) 103–111. 5

[6] T. Kajdanowicz, P. Kazienko and W. Indyk, *Parallel processing of large graphs*, *Future Generation Computer Systems* **32** (2014) 324–337. 5

[7] A.-L. Barabási, R. Albert and H. Jeong, *Mean-field theory for scale-free random networks*, *Physica A: Statistical Mechanics and its Applications* **272** (Oct., 1999) 173–187. 8

[8] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to algorithms, fourth edition*, 04, 2022. 11

[9] R. E. Korf, *Multi-way number partitioning*, in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, (San Francisco, CA, USA), p. 538–543, Morgan Kaufmann Publishers Inc., 2009. 11

[10] N. Karmarker and R. M. Karp, *The differencing method of set partitioning*, Tech. Rep. UCB/CSD-83-113, 1983. 11

[11] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa et al., *Blue gene/l torus interconnection network*, *IBM Journal of Research and Development* **49** (2005) 265–276. 18

[12] D. Chakrabarti, Y. Zhan and C. Faloutsos, *R-mat: A recursive model for graph mining*, vol. 6, 04, 2004. DOI. 22

[13] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." https://snap.stanford.edu/data/twitter-2010.html, June, 2014. 22

[14] G. Karypis and V. Kumar, *Metis—a software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing ordering of sparse matrices*, . 38

[15] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity and S. Haridi, *Ja-be-ja: A distributed algorithm for balanced graph partitioning*, in *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 51–60, 2013. DOI. 38

[16] W. Y. H. Adoni, T. Nahhal, M. Krichen, A. El byed and I. Assayad, *Dhpv: a distributed algorithm for large-scale graph partitioning*, *Journal of Big Data* (09, 2020) . 38

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, *SIGCOMM Comput. Commun. Rev.* **31** (Aug., 2001) 149–160. 39

[18] F. Rahimian, A. H. Payberah, S. Girdzijauskas and S. Haridi, *Distributed vertex-cut partitioning*, in *Distributed Applications and Interoperable Systems* (K. Magoutis and P. Pietzuch, eds.), (Berlin, Heidelberg), pp. 186–200, Springer Berlin Heidelberg, 2014. 44