On the Use of the `ubcdiss` Template

by

Johnny Canuck

B. Basket Weaving, University of Illustrious Arts, 1991

M. Silly Walks, Another University, 1994

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Basket Weaving)

The University of British Columbia
(Vancouver)

April 2192

© Johnny Canuck, 2192

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

On the Use of the `ubcdiss` Template

submitted by **Johnny Canuck** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Basket Weaving**.

Examining Committee:

John Smith, Professor, Materials Engineering, UBC
Supervisor

Mary Maker, Professor, Materials Engineering, UBC
Supervisory Committee Member

Nebulous Name, Position, Department, Institution
Supervisory Committee Member

Magnus Monolith, Position, Other Department, Institution
Additional Examiner

Additional Supervisory Committee Members:

Ira Crater, Professor, Materials Engineering, UBC
Supervisory Committee Member

Adeline Long, CEO of Aerial Machine Transportation, Inc.
Supervisory Committee Member

Abstract

Graph structured data is used in a variety of applications because it naturally models ubiquitous concepts such as social networks, protein structures, and supply chains [6, 17]. This has motivated the development of Graph Processing Systems (GPS) whose aim is to process analytic queries such as PageRank, Shortest Paths, or Connected Components. Previous work accelerated graph processing by modifying the graph's data layout in memory [1, 5, 14] or on disk [11, 13]. Since we typically describe a graph G in terms of its Vertex and Edge sets using the notation $G(V,E)$, most GPS can be categorized as Vertex-Centric (VC) [8, 11, 19] or Edge-Centric (EC) [15] dependant on whether the systems implement analytical queries by applying a function over each vertex or each edge.

In the VC model, algorithms rely on user-defined vertex programs to compute analytic properties of an input graph. Vertex programs are run iteratively on every vertex in the graph. In each iteration, each vertex executes a user-defined vertex program, and messages are exchanged between neighbouring vertices to propagate updated vertex values. VC systems reorder the *vertices* of the graph to improve the locality of vertices that are expected to be frequently accessed together.

In the EC model, iteration occurs over the edges of the graph. For each edge in the graph, we apply an update to either the source or destination vertex incident on that edge. EC systems reorder the *edges* of the graph to mitigate the random-access pattern of incident vertices that is common to many graph processing kernels such as PageRank.

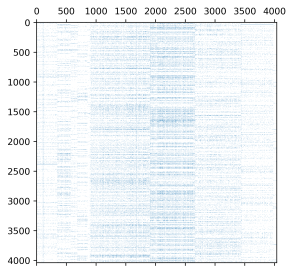
One way of representing a graph is as a square $N \times N$ matrix, A, where $N = |V|$. A non-zero element $A_{i,j}$ indicates the existence of an edge from vertex i to vertex j. Real-world graphs are typically sparse [4], meaning that the number of

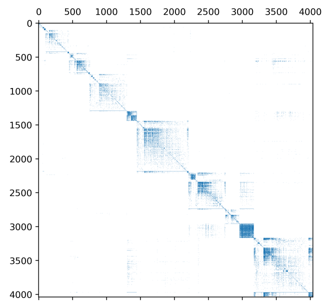
edges, $M = |E|$, is much smaller than the number of possible edges in the graph: $M \ll M_{\text{max}} = \binom{N}{2}$.

EC systems that order the edges of the graph by ascending *Source* or *Destination* ID effectively iterate over A in Row-major or Column-major order, respectively. If we traverse the adjacency matrix of a graph in Row-major order, we will have excellent locality in the source vertices (since we will process all outgoing neighbours of a source vertex before moving on to the next source vertex), but our access to the destination vertex of each outgoing edge will correspond to near-random memory accesses of the vertex array. Prior work [14] addressed this concern by using an edge ordering defined by the Hilbert Space Filling Curve (HSFC), which is a way of assigning indices to the edges of a graph that produces locality in *both* the source and destination vertices.

VC systems can use vertex reordering as a preprocessing optimization to improve the memory access locality of the vertices. For example, certain VC systems [5, 18] use the degrees of the vertices to cluster and/or sort the vertices (e.g., sorting the vertices by descending order of degree). The rationale for this is that high degree vertices (also known as "Hub" vertices) are, by definition, neighbours of a large number of vertices. These hubs will be frequently accessed when iterating over the edge set of the graph [2]. A descending degree sort collocates these hub vertices in memory and increases the likelihood of frequently accessed vertices being cached. Alternatively, Rabbit Order [1] relies on the observation that many real-world graphs such as social networks contain community structures, where vertices that belong to the same community share a larger number of edges than vertices that belong to different communities. The goal of Rabbit Order is to order the vertices in such a way that consecutive vertex IDs correspond to meaningful communities. The algorithm first detects these communities and then labels the vertices according to those communities.

A vertex reordering can be thought of as a function that maps each vertex ID to a new ID in the range: $[0, N)$. This mapping or relabeling is known as a *graph isomorphism*, since all edges between vertices are preserved and the structure of the graph is unchanged. However, vertex reordering can impose "structure" on the adjacency matrix of the graph. Figure 1.1 shows the adjacency matrix of a Facebook social network graph that was constructed from the data of survey participants using the Facebook App [12]. The graph has been reordered using 3 vertex orders: a random vertex ID assignment, and the IDs calculated using Rabbit and Cuthill-McKee orders.


(a) Random Order


(b) Rabbit Order

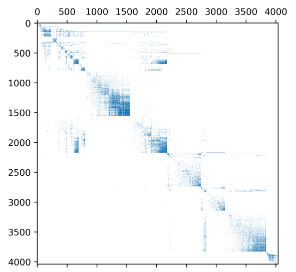

(c) Cuthill-McKee Order

Figure 1: Adjacency matrices of an undirected Facebook Social Network graph with 4,039 vertices and 88,234 edges. Each pixel denotes an undirected edge ("friend-of" relationship) between a pair of vertices (users) in the graph. Note the detected dense communities (submatrices) along the diagonal in (b) and the reduced bandwidth of the sparse matrix in (c).

To date, researchers have looked at *either* vertex or edge ordering to optimize the performance of their VC or EC systems, but there has not been an extensive evaluation of possible *vertex-and-edge* orderings. We define a **Vertex-and-Edge Ordering** of a graph as a preprocessing algorithm that:

1. Reorders the *vertices* of the graph in order to introduce structure into the adjacency matrix.

2. The *edges* are traversed using an edge ordering that leverages the structure of the isomorphic adjacency matrix. The edge ordering ignores large empty regions of the adjacency matrix and traverses the dense regions that were introduced using the vertex ordering using the HSFC.

This thesis is concerned in filling this knowledge gap and investigates the interaction between Vertex and Edge ordering. Namely, we answer the following research questions:

RQ1: It has been shown that vertex and edge ordering confer performance benefits in a variety of applications. Is it possible to combine vertex and edge ordering as a preprocessing step such that the performance benefit gained by the combination of both is compounded (i.e., is greater than using either separately)?

- Can edge-centric graph traversal be sped up by first introducing some structure into the adjacency matrix?

RQ2: Given an arbitrary input graph, is there a *vertex-and-edge* ordering combination that yields the best performance for an EC traversal (measured in speed of execution and number of cache misses)?

We answer these questions by making the following contributions. We:

1. Perform a preliminary performance evaluation on different vertex and edge orderings on single-threaded EC traversals for a variety of graph datasets. We conclude that there *does not* exist a one-size-fits-all *vertex-and-edge* ordering that outperforms all others for all types of graphs.

2. Derive an analytic model of performance using a dataset of graph features (statistical measures that summarize a graph) to identify the characteristics of a graph that help us determine which vertex-and-edge ordering performs best for EC workloads.

3. Develop a fully parallel implementation of the SlashBurn vertex ordering technique: **ParSB**.

4. Propose a novel, lock-free, multithreaded vertex-and-edge ordering technique that leverages the compressed graph representation given by ParSB and traverses the edges of the graph using the HSFC.

5. Evaluate our novel vertex-and-edge ordering and compare its performance against locking-based and merging-based multithreaded vertex-and-edge orderings.

Lay Summary

The lay or public summary explains the key goals and contributions of the research/ scholarly work in terms that can be understood by the general public. It must not exceed 150 words in length.

Preface

At University of British Columbia (UBC), a preface may be required. Be sure to check the GPS guidelines as they may have specific content to be included.

Table of Contents

List of Tables

List of Figures

Glossary

This glossary uses the handy `acroynym` package to automatically maintain the glossary. It uses the package's `printonlyused` option to include only those acronyms explicitly referenced in the LaTeX source. To change how the acronyms are rendered, change the `\acsfont` definition in `diss.tex`.

GPS	Graduate and Postdoctoral Studies
GPS	Graph Processing System
VC	Vertex-Centric
EC	Edge-Centric
HSFC	Hilbert Space Filling Curve
WWR	Wing-Width-Ratio
GNN	Graph Neural Network

Acknowledgments

Thank those people who helped you.

Don't forget your parents or loved ones.

You may wish to acknowledge your funding sources.

Chapter 1

Introduction

Graph structured data is used in a variety of applications because it naturally models ubiquitous concepts such as social networks, protein structures, and supply chains [6, 17]. This has motivated the development of Graph Processing Systems (GPS) whose aim is to process analytic queries such as PageRank, Shortest Paths, or Connected Components. Previous work accelerated graph processing by modifying the graph's data layout in memory [1, 5, 14] or on disk [11, 13]. We typically describe a graph G in terms of its Vertex and Edge sets using the notation $G(V,E)$. Correspondingly, most GPS can be categorized as Vertex-Centric (VC) [8, 11, 19] or Edge-Centric (EC) [15], depending on whether the systems implement analytical queries by applying a function over each vertex or each edge.

In the VC model, algorithms rely on user-defined vertex programs to compute analytic properties of an input graph. Vertex programs are run iteratively on every vertex in the graph. In each iteration, each vertex executes a user-defined vertex program, and messages are exchanged between neighbouring vertices to propagate updated vertex values. VC systems reorder the *vertices* of the graph to improve the locality of vertices that are expected to be frequently accessed together.

In the EC model, iteration occurs over the edges of the graph. For each edge in the graph, we apply an update to either the source or destination vertex incident on that edge. EC systems reorder the *edges* of the graph to mitigate the random-access pattern of incident vertices that is common to many graph processing kernels such as PageRank.

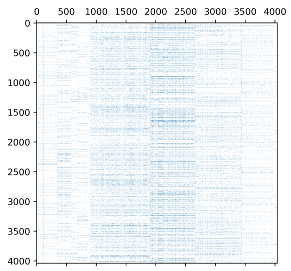
One way of representing a graph is with a square $N \times N$ adjacency matrix, A, where $N = |V|$. A non-zero element $A_{i,j}$ indicates the existence of an edge from vertex i to vertex j. Real-world graphs are typically sparse [4], meaning that the number of edges, $M = |E|$, is much smaller than the number of possible edges in the graph: $M \ll M_{\max} = \binom{N}{2}$. Equivalently, this means that the adjacency matrix of a typical real-world graph will be mostly filled with zeroes.

EC systems that order the edges of the graph by ascending *Source* or *Destination* ID effectively iterate over A in Row-major or Column-major order, respectively. If we traverse the adjacency matrix of a graph in Row-major order, we will have excellent locality in the source vertices (since we will process all outgoing neighbours of a source vertex before moving on to the next source vertex), but our access to the destination vertex of each outgoing edge will correspond to near-random memory accesses of the vertex array. Prior work [14] addressed this concern by using an edge ordering defined by the Hilbert Space Filling Curve (HSFC), which is a way of assigning indices to the edges of a graph that produces locality in *both* the source and destination vertices.

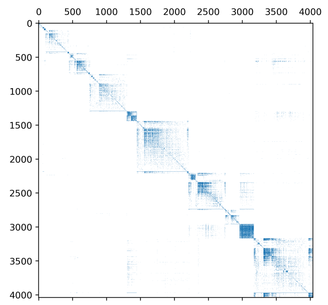
VC systems can use vertex reordering as a preprocessing optimization to improve the memory access locality of the vertices. For example, certain VC systems [5, 18] use the degrees of the vertices to cluster and/or sort the vertices (e.g., sorting the vertices by descending order of degree). The rationale for this is that high degree vertices (also known as "Hub" vertices) are, by definition, neighbours of a large number of vertices. These hubs will be frequently accessed when iterating over the edge set of the graph [2]. A descending degree sort collocates these hub vertices in memory and increases the likelihood of frequently accessed vertices being cached. Alternatively, Rabbit Order [1] relies on the observation that many real-world graphs such as social networks contain community structures, where vertices that belong to the same community share a larger number of edges than vertices that belong to different communities. The goal of Rabbit Order is to order the vertices in such a way that consecutive vertex IDs correspond to meaningful communities. The algorithm first detects these communities and then labels the vertices according to those communities.

A vertex reordering can be thought of as a function that maps each vertex ID to a new ID in the range: $[0, N)$. This mapping or relabeling is known as a *graph iso-*

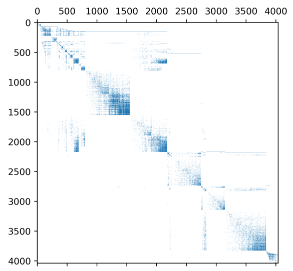
morphism, since all edges between vertices are preserved and the structure of the graph is unchanged. However, vertex reordering can impose “structure” on the adjacency matrix of the graph. Figure 1.1 shows the adjacency matrix of a Facebook social network graph that was constructed from the data of survey participants using the Facebook App [12]. The graph has been reordered using 3 vertex orders: a random vertex ID assignment, and the IDs calculated using the Rabbit and Cuthill-McKee orders.



(a) Random Order



(b) Rabbit Order



(c) Cuthill-McKee Order

Figure 1.1: Adjacency matrices of an undirected Facebook Social Network graph with 4,039 vertices and 88,234 edges. Each pixel denotes an undirected edge (“friend-of” relationship) between a pair of vertices (users) in the graph. Note the detected dense communities (submatrices) along the diagonal in (b) and the reduced bandwidth of the sparse matrix in (c).

To date, researchers have looked at *either* vertex or edge ordering to optimize the performance of their VC or EC systems, but there has not been an extensive evaluation of possible *vertex-and-edge* orderings. We define a **Vertex-and-Edge Ordering** of a graph as a preprocessing algorithm that:

1. Reorders the *vertices* to introduce structure into the adjacency matrix.

2. Traverses the *edges* using an edge ordering that leverages the structure of the isomorphic adjacency matrix. For example, the edge ordering could skip over large empty regions and traverse the dense regions using the HSFC.

This thesis is concerned in filling this knowledge gap and investigates the interaction between Vertex and Edge ordering. Namely, we answer the following research questions:

RQ1: It has been shown that vertex and edge ordering confer performance benefits in a variety of applications. Is it possible to combine vertex and edge ordering as a preprocessing step such that the performance benefit gained by the combination of both is compounded (i.e., is greater than using either separately)?

- Can edge-centric graph traversal be sped up by first introducing some structure into the adjacency matrix?

RQ2: Given an arbitrary input graph, is there a *vertex-and-edge* ordering combination that yields the best performance for an EC traversal (measured in speed of execution and number of cache misses)?

We answer these questions by making the following contributions. We:

1. Perform a preliminary performance evaluation on different vertex and edge orderings on single-threaded EC traversals for a variety of graph datasets. We conclude that there *does not* exist a one-size-fits-all *vertex-and-edge* ordering that outperforms all others for all types of graphs.

2. Derive an analytic model of performance using a dataset of graph features (statistical measures that summarize a graph) to identify the characteristics of a graph that help us determine which vertex-and-edge ordering performs best for EC workloads.

3. Develop a fully parallel implementation of the SlashBurn vertex ordering technique: **ParSB**.

4. Propose a novel, lock-free, multithreaded vertex-and-edge ordering technique that leverages the compressed graph representation given by ParSB and traverses the edges of the graph using the HSFC.

5. Evaluate our novel vertex-and-edge ordering and compare its performance against locking-based and merging-based multithreaded vertex-and-edge orderings.

<here would be an outline of the thesis, detailing the sections of it, and what each section contains.>

1. **Introduction**

2.

3. **Case Study: Does Vertex-and-Edge ordering affect performance?** To answer **RQ1**, we ran the following experiment on a dataset of $\approx$ 150 graphs whose size ranges from modest ($\approx$30MB) to large ($\approx$40GB). For each graph, we computed 8 vertex orderings. For each vertex ordering, we completed 20 iterations of PageRank using either Row-major, Column-major, or the Hilbert Order. We saw that there was not a 1-size-fits-all vertex-and-edge ordering combination. Highlight the different results observed. Visualize dataset of graphs in (reduced) high-dimensional space (t-SNE, PCA) to try to identify clusters of graphs that behave similarly. Present a simple, interpretable model (decision tree, linear regression) that identifies the features that influence the performance of different vertex-and-edge orders.

4. **Parallel Slashburn** The results of the case study show that, for certain graphs, SlashBurn produced the greatest performance improvements. Motivated by this, we propose to parallelize this algorithm, to speedup the computation of this ordering Describe the SlashBurn algorithm in depth. Highlight subroutines of the algorithm that are parallelizable. Describe the Parallel Slashburn algorithm. Evaluate:

   - The speedup between the sequential and parallel implementation.
   - Scalability: speedup when increasing the number of threads.

- Equality of results: the orderings produced by the sequential and parallel implementations should be almost identical. Show that this is the case by measuring the performance improvement that is gained by both implementations, and show how these improvements are practically identical.

4.1. **Modelling the Wing-Width-Ratio (WWR) of Real-world Graph** Define the WWR of a graph: the number of iterations to compute the slashburn ordering $\times$ the number of hubs selected at each SlashBurn iteration. Note the differences in the WWR among graphs. Observe that this metric is a deciding factor of whether to use the SlashBurn order or not. Model the WWR as a function of graph features. Use our model of WWR to predict whether to iterate over the graph using our proposed HilBurn curve.

5. **My Technique: Vertex-and-Edge Ordering using HilBurn** Given that we've seen combined speed ups using the combination of the SlashBurn Vertex Ordering and the Hilbert Edge Ordering, We propose the Hilbert+SlashBurn Vertex-and-Edge ordering, HilBurn.

   (a) First, reorder the vertices of the graph using the ordering produced by Parallel SlashBurn.

      - Use the analytic model that predicts the WWR of a graph given its graph statistics to choose the number of hubs selected at each iteration. Use this model to optimize for a Wing Width that fits in the LLC.
      - In the case where the WWR is too large, we expect that HilBurn would not perform great, so use our model to pick an appropriate Vertex-and-Edge ordering based on the input graph's graph features.

   (b) Second, separate the reordered adjacency matrix into 4 "zones":

      i. **Upper Right Quadrant - URQ**: this region should fit in the LLC. This range is defined by: $[0, ki)$, where:

- k is the number of hubs selected at each Parallel Slashburn iteration
- i is the number of iterations it took to complete Parallel Slashburn

The URQ will be computed in parallel by splitting up the $ki \times ki$ quadrant of the adjacency matrix into subquadrants, each of which contains a copy of the vertex ids in the range of its subquadrant to ensure that no contention occurs while updating vertex data. At the end of each iteration, all threads merge the updates for the vertices in their subquadrants;

ii. **Right Wing**: Can be computed in parallel with URQ. The upper right wing will be split into quadrants based on the number of edges in each region - this is done to ensure a workload balance among the threads operating on the right wing. Each thread will own a non-overlapping region of the vertex id range: $[ki, N-1)$. No locks or merging required for this step.

iii. **Left Wing**:

iv. **Tail**

6. **Evaluation**

7. **Discussion**

8. **Future Work**

9. 9.1. **SIMD Parallelism**

    9.2. **Further speedups using other state of the art (PQ, CCs, degree sorts)**

10. **Conclusion**

Thesis Overview

1. Introduction
   1.1. Fairy Tale:
      - *Context* The capacity of machines
      - *Villain*
      - *Hero*
      - *Happily Ever After*

      Contributions.
2. Background
   2.1. Notation
   2.2. In Memory Graph Data Structures
   2.3. Graph Ordering
      2.3.1. Memory Access Patterns and Bottlenecks in Graph Processing
      2.3.2. Vertex Reordering
      2.3.3. Edge Reordering
   2.4. Modes of Graph Computation
      2.4.1. Push vs. Pull
      2.4.2. Parallel Computation
         2.4.2.1. Propagation Blocking
3. Related Work
4. Case Study: Single-threaded Vertex-and-Edge Ordering
5. Parallel Slashburn
6. RHuBarb - Recursive Hilbert Blocking
7. Evaluation
8. Discussion

Chapter 2

Background

2.1 In Memory Graph Data Structures

We introduce various methods for storing graphs in memory, including the traditional Adjacency Matrix and List representations. We explain how vertices and edges are stored using these data structures. We then present the efficient Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) representations commonly used for large graphs and highlight the challenges that arise when using these compressed representations.

2.1.1 Adjacency Matrix

2.1.2 Adjacency List

2.1.3 Compressed Representations

Compressed Sparse Row, Column

2.1.4 Memory Access Patterns and Bottlenecks in Graph processing

2.2 Graph Ordering

To address the problems mentioned above, there has been a significant amount of research on Graph Ordering Techniques. These techniques rearrange either the vertices or edges of a graph to enhance the memory access locality during graph traversal. We first define spatial locality and demonstrate how vertex reordering can improve it. We also categorize vertex reordering techniques into light, mid, and heavyweight techniques and provide examples. Next, we define temporal locality and explain how edge reordering can optimize it. Finally, we introduce the Hilbert Space Filling Curve and highlight its benefits in improving the temporal locality of edge traversals

2.2.1 Spatial Graph Transforms: Vertex Reordering

LightWeight, HeavyWeight, MidWeight, Slashburn

2.2.2 Temporal Graph Transforms: Edge Reordering

Hilbert Curve

2.3 PageRank

In the next section, we examine the PageRank algorithm and the difficulties associated with parallelizing it. We present the algorithm and explore its widespread use as a benchmark in graph processing. We distinguish between the two modes of computation that can be used to compute a graph's PageRank algorithm. Then, we focus on the challenges in parallelizing the algorithm. Finally, we introduce Propagation Blocking, a recent optimization aimed at enhancing the spatial locality in parallel PageRank computation.

2.3.1 Pull vs. Push

2.3.2 Parallelizing Computation

Propagation Blocking

2.4 Symbols

Symbol	**Definitions**	**Equivalences**
V	Vertex Set	-
E	Edge Set	-
$G(V,E)$	A graph G, with vertex set V, and edge set E	-
n	The number of vertices in a graph	$\|V\|$
m	The number of edges in a graph	$\|E\|$

Chapter 3

Related Work

Related Work

1. **Vertex Centric vs. Edge Centric Graph processing systems**

    - Define and describe the differences between Vertex Centric and Edge Centric GPS;
    - Define CSR in the context of in-memory graph processing systems, and explain how this in-memory representation can cause many cache misses by iterating over the edges of the graph in a vertex centric manner:
        - For each source vertex, access via the destination vertex to the `vdata` array is close to random.
    - Covers: Ligra, GraphChi, Pre-select Static Caching, Mosaic, X-stream, FlashGraph

2. **Vertex Ordering**

    2.1. **Lightweight Reordering Techniques**; Define and motivate the concept of vertex reordering to speed up graph processing. Define Lightweight reordering. Introduce HubCluster, HubSort, Degree Sort, Degree-Based-Grouping. Introduce techniques that are computationally expensive: e.g., Gorder. Discuss techniques that are not lightweight, and not heavy-weight: Rabbit-order, Parallel SlashBurn, Parallel Cuthill-McKee. The cost of a preprocessing should be jointly considered with the runtime of the application: that is, for what size of graph and for how many executions of a graph algorithm does the speedup balance out the ordering time?

3. **Edge Ordering / Hilbert Ordering** Introduce the concept of edge ordering. Introduce COST, and discuss the speedups they observed when using the Hilbert curve. Describe other fields in computer science that have used the Hilbert curve to speed up performance (e.g., Image/video rendering). Define the Hilbert Curve, how to compute it, the computational cost of computing it, how computing it is easily parallelizable, and why it produces an improvement in performance for (certain) graph algorithms.

4. **Vertex-and-edge Ordering: Leveraging the potential Interaction between Vertex and Edge Ordering** Discuss how vertex ordering can produce adjacency matrices with inherent structure that can be leveraged with efficient edge-centric traversals that avoid large empty regions, and traverse filled regions using the Hilbert space filling curve. Introduce in-Hub Temporal Locality - a similar technique that identifies blocks in the adjacency matrix, and applies a suitable traversal direction (push or pull) based on the block's content.

5. **Graph Statistics** Introduce the question: how can we model a graph using features? If we want to model the performance of different reordering algorithms on different graphs, it would be helpful to have a representation of a graph using statistical features: We would expect that two graphs that are "close" to each other in this feature space, to perform similarly when using a combination of vertex-and-edge ordering. Briefly introduce Graph Representation Learning and argue that while recent advances may have produced Graph Neural Network (GNN) architectures that can process large scale graphs, the produced models are generally opaque, and will not yield much insight as to *why* a certain vertex-and-edge ordering produces performance improvement. Introduce Konect and their statistics. Present a table of statistics. Statistics are grouped into categories: Algebraic, Distance, De-

gree, Powerlaw, and Motif counts.

1. **What is the body of work out of which your work grew?** AT ►

   - *"When is Lightweight Reordering an Optimization".*
     - *Showcases the benefit that can be gained by vertex reordering, but also that there exist light and heavyweight reorderings, and that heavyweight techniques may sometimes not be worth the effort.*
   - *SlashBurn*
   - *"Propagation Blocking" - mentioned that parallelizing edge-centric computation using the Hilbert curve may be complicated.*
   - *"Cagra" - did parallelize the hilbert curve, but reported horrible scaling. I still think the reason for this was was a näive implementation – not that it was impossible (in fact, Rhubarb shows that it is possible!).*
   - *Finally, the recent "Spray: Sparse Reductions of Arrays in OPENMP" publication, which addressed the issue with OpenMP dense array reductions being unable to scale.*

   ◄

2. **What work inspired you?** AT ►

   - *McSherry et al. [14]'s use of the Hilbert curve and the notable Single-threaded speedup they achieved simply by iterating over the edges of the graph in a different edge-order.*

   ◄

3. **To which work should you be comparing your approach?** AT ►

   - *Since Rhubarb only supports edge-centric algorithms, Rhubarb should be compared to Graph Processing Systems that either provide or enable implementation of such algorithms. In our evaluation, we've chosen to use the following EC algorithms: PageRank, Connected Components, and Collaborative Filtering. As such, we've chosen the following GPS to compare Rhubarb to:*

   (a) ***Ligra*** *- a de-facto main-memory, parallel graph processing system that provides implementations of PR, CC, and CF. Ligra is a great baseline to compare against for the following reasons:*

i. *It uses Cilk's work stealing scheduler to provide efficient, parallel implementations of the algorithms used in my evaluation.*

ii. *It does not use any optimization with regards to the graph representation (i.e. like GPOP and Syze do with their graph partitioning schemes). Specifically, each iteration of PR or CF in Ligra involves iterating over all the edges in the graph – which is exactly what Rhubarb does, too. So, the amount of work done per iteration is comparable between Ligra and Rhubarb.*

(b) ***GPOP*** *- was the SOTA when it was published (2020). Uses Partition-Centric Processing Methodology (PCPM) to partition the graph into cacheable subgraphs. In PCPM, edges from the same source vertex that point out to multiple destination vertices inside the same subgraph are compressed into one inter-edge to reduce memory traffic [3]. For this reason, GPOP does substantially less work (measured in the number of updates written to a vertex data array in each iteration) than Rhubarb.*

(c) ***Syze*** *- is a recently published optimization of GPOP which addressed an issue in GPOP: the subgraphs produced by GPOP may have been imbalanced in the number of edges they contained - so Syze identifies demanding subgraphs (ones that contain an unequal number of edges) and further subdivides them until a certain threshold is met.*

◀

AT ▶*Here is the outline of the Related Work chapter which will answer the questions above and place Rhubarb in the current landscape of Graph Reordering.*◀

This chapter provides an overview of the current state of Graph Reordering as an optimization being used for graph processing. We highlight key research questions that arise from recent advancements in this field, along with the techniques that facilitated the development of Rhubarb. Additionally, we describe the GPS we used to compare Rhubarb's performance and explain our rationale for selecting them as the benchmark for comparison.

3.1 When is Vertex Reordering an Optimization?

This section is an introduction to Vertex Reordering and its use in accelerating graph processing. We reference the work of Balaji and Lucia [2] to define lightweight and heavyweight reorderings, and highlight how certain heavyweight Vertex Reordering techniques may result in significant preprocessing overhead that outweighs any potential performance gains.

3.2 SlashBurn

AT ► *I actually think it makes more sense to describe SlashBurn in the chapter devoted to my implementation of Parallel Slashburn.* ◄

3.3 Using the Hilbert Curve to Speed Up Edge-Centric Graph Algorithms

This section introduces the work of McSherry et al. [14], which used the Hilbert curve to ameliorate the poor read and write locality of graph traversals. This finding inspired the development of Rhubarb. We also discuss related work that has explored parallelization of edge-centric computation using the Hilbert curve, and highlight how the recent work of Hückelheim and Doerfert [7] has facilitated the development of Rhubarb.

3.4 The Current Main-Memory Graph Processing Landscape

We conclude this chapter by discussing Graph Processing systems that benefit from Vertex ordering and provide implementations of popular Edge-centric algorithms. First, we introduce Ligra [16], a de-facto main-memory GPS that is commonly used as a baseline for comparison for new optimizations and systems. Next, we discuss GPOP [10], and their use of Partition Centric Processing Methodology (PCPM) [9] to improve the cache behaviour of graph algorithms. We conclude this section with a discussion of Syze [3], a recently published optimization of GPOP.

Chapter 4

Evaluation

In the previous chapters we:

1. Described the results of our single-threaded PageRank microbenchmark using different Vertex-and-Edge ordering combinations. We saw that specific Vertex-and-Edge orderings (e.g. Slashburn Vertex Order and Hilbert Edge Order) consistently yielded the greatest speedup. This motivated us to parallelize the SlashBurn Vertex Reordering algorithm.

2. Described the Parallel SlashBurn algorithm and evaluated how well it scales by increasing the number of cores. **AT** ▶*I think it makes sense to discuss and show the evaluation of how well Parallel Slashburn scales right after I describe how I've implemented it - i.e. at the end of that chapter*◀.

3. Described and analyzed RHuBarb (Recursive Hilbert Blocking).

Now, we answer the following Research Questions:

RQ1 *How expensive is RHuBarb's preprocessing?*

RQ1.1 RHuBarb benefits from compressed graph representations (those with a *smaller* number of *denser* blocks, as opposed to a *larger* number of *sparser* blocks.) **How long do these vertex reorderings take to compute?**

RQ1.2 How long does RHuBarb's divide-and-conquer blocking algorithm take?

RQ1.3 How well does RHuBarb's divide-and-conquer blocking algorithm scale with an increasing number of cores?

RQ2 How does RHuBarb compare to State-of-the-Art Graph Processing Systems (GPS)? AT ►*This will involve an evaluation of:*◄

1. Systems: GPOP, Syze, Ligra, GraphMat.
2. Graphs:
   - Social Networks: Twitter, Twitter-MPI, Friendster.
   - Hyperlink Networks: UK Domain, Wikipedia.
   - Road Networks: US Road Network.
   - Bipartite User-Item Rating Networks (Specific to Collaborative Filtering benchmark): Yahoo Songs, Amazon.
3. Edge-Centric Algorithms:
   - PageRank, Connected Components, Collaborative Filtering.

RQ3 The performance of RHuBarb depends on 2 user-defined parameters:

1. *d, Dynamic Group Size*: How many consecutive Hilbert blocks should be dynamically assigned to cores during Edge-Centric traversal?
2. *m, Maximum Number of Edges per Block*: each Hilbert block must contain *at most* this many edges.

How should users choose the values for d,m?

RQ4 In answering **RQ2**, we identified a performance bottleneck for SOTA systems (GPOP, SYZE) on graphs whose vertices have been relabelled using the SlashBurn or Descending Degree Sort vertex reorderings (i.e. graphs with "concentrated edge densities"). Since, given an arbitary input graph, one has no a priori knowledge of the edge density of the graph's "Original" vertex ID assignment (except for known examples of Hyperlink networks in the literature), we answer the following: **For graphs whose "Original" Vertex**

ID assignment is "close" to a Descending Degree Sort, does RHuBarb perform well out-of-the-box?

AT ►*Or maybe?*◄ **How often are the Vertex IDs of real-world graphs close to a Descending Degree Sort?**

We begin this chapter by describing the GPS we'll compare against and the graph datasets and algorithms we'll use in our comparison. Table 4.1 lists the graph datasets we used in our evaluation and Sections 4.1 and 4.2 briefly describe the algorithms and systems, respectively, and why we chose these specifically. Sections 4.3 to 4.6 answer **RQ1-4**, respectively.

Table 4.1: Graph Datasets

Graph	Description	n	m
twitter	Twitter Follower	41,652,230	1,468,365,182
road	USA Road	23,947,347	57,708,624
uk	Hyperlink Network of .uk Domain	105,153,952	3,301,876,564
amazon	Rating Network	31,050,733	82,677,131
...	...	...	...

4.1 Edge-Centric Graph Algorithms

4.1.1 PageRank

4.1.2 Connected Components

4.1.3 Collaborative Filtering

4.2 Graph Frameworks

4.2.1 Ligra

4.2.2 GraphMat

4.2.3 GPOP and Syze

4.3 Preprocessing Overhead

This section lists the preprocessing time of various vertex reorderings AT ▶*Parallel-SlashBurn, Descending Degree Sort, Degree-Based-Grouping, Rabbit Order, COrder*◀and Recursive Hilbert-Blocking. We find that certain costly vertex reorderings take longer to compute than certain graph algorithms. In such cases, we also list the number of times the computation must be repeated to amortize the total cost of preprocessing.

4.3.1 Cost of Vertex Reorderings

4.3.2 Cost of Recursive Hilbert-Blocking

4.3.3 Scaling of Recursive Hilbert-Blocking

4.3.4 Time to Amortize Costly Preprocessing Steps

4.4 Multicore Comparison

This section compares RHuBarb to State-of-the-Art, main-memory GPS. We compare the performance of RHuBarb to these systems using a representative set of Edge-Centric graph algorithms and graph datasets. Using an increasing number of cores, we measure runtime performance and L2 and LLC cache miss rate.

AT ►*A figure like* 4.1 *for PageRank, Connected Components, and Collaborative Filtering. (More coloured lines to show the performance of the different systems.) Also, similar figures to show the L2, LLC Cache-miss rates as we increase the number of cores.* ◄

4.5 Effect of Dynamic Group Size and Minimum Number of Edges per Block

In this section, we evaluate the effect of the 2 user-defined parameters of RHuBarb on performance.

1. *d, Dynamic Group Size*: How many consecutive Hilbert blocks should be dynamically assigned to cores during Edge-Centric traversal?
2. *m, Maximum Number of Edges per Block*: each Hilbert block must contain *at most* this many edges.

We perform this performance analysis in order to find a heuristic value for these user-defined parameters. We evaluate RHuBarb using the combination of $d \in \{1,2,4,8,16,32,64\}$ and $m \in \{8192, 16{,}384, 32{,}768, 65{,}536, 131{,}072, 262{,}144\}$.

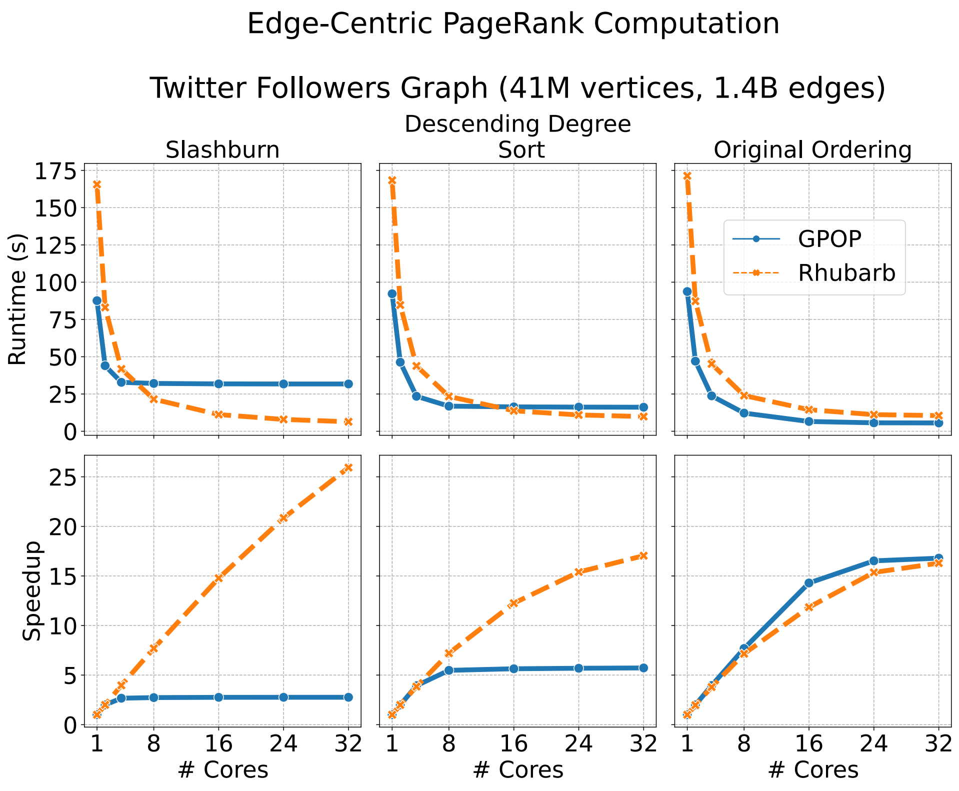


Figure 4.1: Runtime and Scaling for different vertex reorderings (Original, Descending (Out) Degree Sort, and Slashburn) for PageRank computation using GPOP and Hilbert Blocking.

Since the combination of *d* and *m* dictates the amount of work assigned to each core, we find that the size of the private L2 Cache acts as a good heuristic for a rough upper bound of values to assign to *d* and *m*. As a result, RHuBarb uses the L2 cache-size at runtime to assign values to *d* and *m*.

4.6 On which graphs does RHuBarb perform well "out-of-the-box"?

AT ► *This is the "fuzziest" section in my mind, and is my attempt to answer your note about "Default orders" in your email response. I'm not sure how useful the distance met-*

rics I found are for the purpose of comparing the similarity between two vertex reorderings. My current hypothesis is the following: ◀

1. I observed that RHuBarb outperformed SOTA on specific graphs **without** performing any vertex reordering (i.e. using the original vertex ID assignment).

2. I hypothesize that, for these graphs, the "distance" between the "Original" order and a Descending Degree Sort should be **smaller** than the distance between Original and Descending Degree Sort for graphs for which we *did* need to sort them to see any performance improvement using RHuBarb.

AT ▶*The distance metrics I'm considering:*◀

- **Weighted Kendall Tau Distance**: measures the number of pairwise disagreements between two rankings. It counts the number of times that two items are ranked differently between two rankings. AT ▶*I can weigh vertex rankings by the vertices' out-degrees. i.e. Vertices with high degrees will contribute more to the final distance scores than vertices with low degree. I don't think this is a good metric, because it simply counts the number of times vertex ID assignment were out of order, not "how" out of order they were. For this reason, I think Manhattan distance (below), makes more sense.*◀

- **Manhattan distance**: calculates the distance between two rankings based on the sum of the absolute differences between the ranks of each vertex.

Bibliography

[1] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31. IEEE, 2016. → pages iii, iv, 1, 2

[2] V. Balaji and B. Lucia. When is graph reordering an optimization? In *IEEE International Symposium on Workload Characterization (IISWC)*, 2018. → pages iv, 2, 17

[3] Y. Chen and Y.-C. Chung. An unequal caching strategy for shared-memory graph analytics. *IEEE Transactions on Parallel and Distributed Systems*, 2023. → pages 17, 18

[4] M. Danisch, O. Balalau, and M. Sozio. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*, pages 589–598, 2018. → pages iii, 2

[5] P. Faldu, J. Diamond, and B. Grot. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–13. IEEE, 2019. → pages iii, iv, 1, 2

[6] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Computing Surveys (CSUR)*, 51(3):1–53, 2018. → pages iii, 1

[7] J. Hückelheim and J. Doerfert. Spray: Sparse reductions of arrays in openmp. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 475–484. IEEE, 2021. → page 18

[8] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi:Large-Scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012. → pages iii, 1

[9] K. Lakhotia, R. Kannan, and V. Prasanna. Accelerating pagerank using partition-centric processing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 427–440, 2018. → page 18

[10] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna. Gpop: A cache and memory-efficient framework for graph processing over partitions. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 393–394, 2019. → page 18

[11] E. Lee, J. Kim, K. Lim, S. H. Noh, and J. Seo. Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 459–474, 2019. → pages iii, 1

[12] J. Leskovec and J. Mcauley. Learning to discover social circles in ego networks. *Advances in neural information processing systems*, 25, 2012. → pages v, 3

[13] 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 Twelfth European Conference on Computer Systems*, pages 527–543, 2017. → pages iii, 1

[14] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015. → pages iii, iv, 1, 2, 16, 18

[15] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013. → pages iii, 1

[16] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013. → page 18

[17] Y. Yan, S. Zhang, and F.-X. Wu. Applications of graph theory in protein structure identification. *Proteome science*, 9(1):1–10, 2011. → pages iii, 1

[18] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017. → pages iv, 2

[19] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015. → pages iii, 1

Appendix A

Supporting Materials

This would be any supporting material not central to the dissertation. For example:

- additional details of methodology and/or data;
- diagrams of specialized equipment developed.;
- copies of questionnaires and survey instruments.