Vertex-and-Edge Ordering for Faster Parallel Graph Processing

by

Alex Trostanovsky

BCS, Carleton University, 2020

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

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

October 2023

© Alex Trostanovsky, 2023
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Vertex-and-Edge Ordering for Faster Parallel Graph Processing**

submitted by Alex Trostanovsky in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

**Examinining Committee:**

Margo Seltzer, Professor, Department of Computer Science, UBC  
*Supervisor*

Arpan Gujarati, Assistant Professor, Department of Computer Science, UBC  
*Supervisory Committee Member*
Abstract

Graph structured data, which models complex relationships, describes data in myriad domains, such as social network analysis, protein structure analysis, and supply chain management. As the size of graph data grows, researchers develop modern systems to handle massive datasets with trillions of entries. To accelerate processing, systems optimize the data layout of vertices and edges in memory or on disk, but, to date, researchers have treated the performance improvements due to vertex and edge orderings separately. This work investigates the interaction between vertex and edge orderings. We explore whether combining these orderings improves performance. An extensive performance study of different orderings finds that a specific combination, the SlashBurn vertex order and the Hilbert edge order, consistently provides the fastest runtimes for scale-free graphs. These results motivate our development of a parallelized SlashBurn and the Rhubarb edge ordering and blocking technique. Using 14 cores, Parallel SlashBurn is up to 11.96× faster than the sequential implementation. Rhubarb enables the scalable implementation of parallel edge-centric algorithms using the Hilbert curve and offers end-to-end performance speedup for the Collaborative Filtering application of up to more than 2.5× over modern parallel Graph Processing Systems.
Lay Summary

Graphs are used to represent complex networks such as friends in social networks or molecules in protein structures. Companies use specialized systems called Graph Processing Systems to make sense of these graphs and answer questions such as: who is the most influential person in a network? or, what is the degree of separation between two people? As graphs get larger, researchers work on ways to process them more quickly. One common approach is to change the way the graph’s data is stored in the computer’s memory. We explore how two things - the ways in which the parts of a graph are arranged and the ordering of the connections between those parts - affect the speed of graph processing systems. We find that combining two specific techniques, the SlashBurn and Hilbert methods, yields substantial performance gains. Based on these results, we implement optimized and parallel versions of both techniques, enabling faster graph processing.
Preface

The work presented in this thesis was conducted in the Systopia Lab at the University of British Columbia, Vancouver campus. All work presented in this thesis was written and produced by Alex Trostanovsky under the supervision of Professor Margo Seltzer. Computing resources used throughout my studies were generously provided by Professor Margo Seltzer, Compute Canada, and the Cloud Innovation Centre (CIC) at the University of British Columbia (UBC).
# Table of Contents

Abstract ........................................................................................................ iii
Lay Summary .................................................................................................. iv
Preface ........................................................................................................ v
Table of Contents ........................................................................................ vi
List of Tables ............................................................................................... ix
List of Figures ............................................................................................. xi
List of Symbols ............................................................................................ xix
List of Abbreviations .................................................................................. xxi
Acknowledgments ....................................................................................... xxii
Dedication ................................................................................................. xxiv

1 Introduction ................................................................. ................................. 1

2 Background ................................................................. ................................. 5
   2.1 Preliminaries ................................................................. ........................... 6
   2.2 In Memory Graph Data Structures ......................................................... 7
   2.3 Memory Access Patterns and Bottlenecks in Graph Processing ............ 8
      2.3.1 Vertex Partitioning for Parallel Graph Processing ......................... 10
      2.3.2 Modes of Computation for Parallel Graph Processing ................. 10
   2.4 Graph Ordering ................................................................. ........................ 11
      2.4.1 Vertex Reordering ................................................................. 11
      2.4.2 Edge Reordering ................................................................. 14

3 Case Study: Single-Threaded Performance Evaluation of Vertex-and-Edge Orderings .... 18
   3.1 Graph Analytic Task: PageRank .......................................................... 20
6.6 Comparison to Other Edge-Centric Implementations ........................................... 76
  6.6.1 Preprocessing ................................................................................................. 76
  6.6.2 Runtime ......................................................................................................... 78
  6.6.3 Scaling ............................................................................................................ 91
6.7 Locality Preservation of Rhubarb .......................................................................... 99
6.8 Takeaways ............................................................................................................ 100

7 Discussion ............................................................................................................. 102
  7.1 Other Uses of Hilbert Ordering .......................................................................... 102
  7.2 Limitations and Future Work ........................................................................... 103
    7.2.1 Sparse vs. Dense Blocks and Differentiated Modes of Computation for Rhubarb 103
    7.2.2 Improving Parallel SlashBurn ...................................................................... 106

8 Conclusion ............................................................................................................. 108

Bibliography .............................................................................................................. 110

A Supporting Materials .............................................................................................. 117
  A.1 End-to-end performance of Rhubarb as a function of its hyperparameters .............. 117
List of Tables

| Table 3.1 | Real World Graphs evaluated in the case study. | 23 |
| Table 3.2 | Uniform Random Graphs evaluated in the case study. The uniform random graphs are symmetric: for each edge \( u, v \), the inverse edge \( v, u \) exists and is explicitly represented in the graph. Thus, \( m = 2dn \). | 24 |
| Table 3.3 | Kronecker Random Graphs evaluated in the case study. Note that \( n < 2^{22} \) for all Kronecker graphs. This is due to the removal of singleton vertices (vertices with no neighbours) after the random graph generation. | 24 |
| Table 3.4 | Case study results for the Konect graphs from Table 3.1. Each cell lists the average runtimes (across five trials) for 20 iterations of edge-centric PageRank for a given graph, vertex order, and edge order. Standard deviations are listed in parenthesis below each value. Bolded text corresponds to the best runtime for a graph and vertex ordering. Bolded, red text corresponds to the best runtime for a graph. | 26 |
| Table 3.5 | Case study results for the uniform random graphs from Table 3.2. Each cell lists the average runtimes (across five trials) for 20 iterations of edge-centric PageRank for a given graph, vertex order, and edge order. Standard deviations are listed in parenthesis below each value. Bolded text corresponds to the best runtime for a graph and vertex ordering. Bolded, red text corresponds to the best runtime for a graph. | 27 |
| Table 3.6 | Case study results for the Kronecker graphs from Table 3.3. Each cell lists the average runtimes (across five trials) for 20 iterations of edge-centric PageRank for a given graph, vertex order, and edge order. Standard deviations are listed in parenthesis below each value. Bolded text corresponds to the best runtime for a graph and vertex ordering. Bolded, red text corresponds to the best runtime for a graph. | 27 |
Table 4.1 Comparison of execution times for the various implementations of SlashBurn. Run-
times are in seconds. 1-Core is a sequential implementation of SlashBurn that uses
QuickSort and BFS. The Optimized parallel implementation of SlashBurn uses IPS4o,
Afforest, BlockReduction and MapReduction. Optimized Speedup is the ratio be-
tween 1-Core and 1-Core Optimized execution time. Parallel Speedup is the ratio
between 1-Core Optimized to the 14-Core Optimized execution time. Total Speedup
is the combined speedup gained by using the Parallel SlashBurn optimizations and
the 14 available cores.

Table 6.1 Graph Datasets

Table 6.2 Hardware specification of systems used in evaluation.

Table 6.3 Runtimes of Parallel Vertex Orderings using Xeon with 14 cores. Given 128GB of
RAM, Rabbit order was unable to reorder sd-arc and friendster due to insuf-
ficient memory. Although Parallel SlashBurn is consistently slower than Rabbit order
(the other comparably “midweight” reordering algorithm), its memory consumption
is smaller, allowing it to reorder the larger sd-arc and friendster graphs.

Table 6.4 Preprocessing times for GraphMat (GM) and Rhubarb (RHB) using Xeon with 14
cores. Times are in seconds. With 128 GB of RAM, we were unable to preprocess
sd and fr using GraphMat. On average, Rhubarb’s preprocessing is 3.07× faster
than the time required to construct the doubly-compressed sparse column (Doubly
Compressed Sparse Column (DCSC)) representation used by GraphMat.

Table 6.5 The number of PageRank iterations to converge for $\varepsilon = 10^{-9}$.

Table 6.6 PageRank runtimes, excluding preprocessing, using Xeon with 14 cores. The fastest
runtime per graph is highlighted using bold, red text. Cells labeled OOM indi-
cate that the system ran out of memory. For almost all scale-free networks (sx-
stackoverflow, soc-LiveJournal1, wikipedia_link_en, twitter, and sd-arc), using Rhubarb and SlashBurn results in the fastest runtime to compute
the PageRank of the graph. For sx-stackoverflow, soc-LiveJournal1, wikipedia_link_en, and twitter, the next best performing combination is
Rhubarb and Descending Degree Sort. This is due to the similarity between De-
scending Degree Sort and SlashBurn as well as Rhubarb’s ability to efficiently tra-
verse dense colocated adjacency matrix blocks. We use Figure 6.2 to illustrate the
similarity between the sparsity patterns of the adjacency matrices of SlashBurn and
Descending Degree Sort.

Table 6.7 Connected Components Runtimes, excluding preprocessing, using Xeon with 14
cores. The fastest runtime per graph is highlighted using bold, red text. Cells con-
taining OOM indicate that the system ran out of memory, while cells containing OOT
indicate that the tests ran longer than five minutes.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Adjacency matrices of a directed, user-to-user, trust network with 4,658 users and 40,133 trust ratings [60]. Each pixel denotes a directed edge (“trusts” relationship) between a pair of vertices (users) in the graph. We ignore edge weights (ratings). Note the detected dense communities (submatrices) along the diagonal for Rabbit-Order and the dense clustering of edges for SlashBurn.</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>In-memory Graph Representations</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Irregular access to the $v_{data}$ array by traversing the out-neighbourhood of $u$. The circled numbers refer to the order in which elements from $v_{data}$ are retrieved. Due to the distance between the IDs of $u$’s out-neighbours, this traversal suffers from poor spatial locality. Figure adapted from [10].</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Degree-based Vertex Orderings. The top of the figure shows an undirected graph whose vertices are labelled using an arbitrary ordering. The arrays in Subfigures (a)-(d) store the degrees of the vertices and illustrate the different vertex ID assignments for the different vertex orderings. High-degree vertices are denoted by blue entries. Figure adapted from [10].</td>
<td>13</td>
</tr>
</tbody>
</table>
The adjacency matrix of the CiaoDVD Social Network graph [60] after performing vertex reordering using the methods described in Section 2.4.1. A marked blue pixel in the \((i, j)\)th coordinate of the plot corresponds to the existence of an edge \((i, j)\) in the graph. Expectedly, the Random vertex ordering produces a sparse matrix with a uniform distribution of edges. The degree-based techniques (Descending Degree Sort, Hub-Cluster, Hub-Sort) all produce a concentration of edges in the upper-left of the adjacency matrix as a result of the colocation of the hub vertices of the graph. Note that the boundary between the low- and high-degree vertices of the graph in the Hub-Cluster and Hub-Sort plots produces close-to-identical lower-right quadrants. Rabbit-Order produces clusters along the diagonal of the matrix that correspond to the hierarchical communities in the graph. SlashBurn concentrates edges to the left, top, and diagonal (the wings and tail, respectively). This results in an adjacency matrix that requires the smallest number of square blocks to cover the edges among the vertex reorderings discussed. Lastly, the Original vertex ID assignment is far from random. While the method of vertex ID assignment is typically opaque to the users, and varies depending on the type of network being modelled, certain graph datasets use default vertex ID assignments that exhibit excellent locality (e.g., hyperlink networks from the Laboratory for Web Algorithmics [22]).
Figure 3.3 Performance of edgelist traversal using the Hilbert curve on the real-world graphs from Table 3.1 as a function of vertex ordering and Average Read/Write Distance (ARWD). The title of each plot lists the range of the number of edges in the graphs of the plot. Graphs are distinguished by colours within a plot. Vertex orders are distinguished by marker types. The shared y-axes show performance measured in Millions of Edges traversed per Second (MEPS) and calculated using Equation (3.3). The shared logarithmic x-axes show the Average Read/Write Distance (ARWD) calculated using Equation (3.6). Highlighted marks with a bold black outline indicate the fastest vertex order for that graph.

Figure 4.1 Sequential out-degree computation using an array reduction.

Figure 4.2 Concurrent out-degree computation using atomic updates.

Figure 4.3 Concurrent out-degree computation using the OpenMP reduction clause.

Figure 4.4 A comparison of OpenMP array reduction and Spray BlockReduction. Each box represents a memory location. The boxes for Threads 1 and 2 are private. The locations updated by Threads 1 and 2 are coloured using blue and green, respectively. The Spray BlockReduction avoids the expensive initialization of thread-private arrays. Using BlockReduction, each thread initializes only the memory it requires. Figure adapted from [54].

Figure 4.5 Concurrent out-degree computation using a BlockReduction with block size = 512.

Figure 4.6 Scaling of Parallel SlashBurn with an increasing number of cores.

Figure 4.7 Runtime (in ms) and number of active vertices per iteration of Parallel SlashBurn using 14 cores. The coloured regions represent the cumulative time performing each subroutine in every iteration. Degree sort refers to the IPS^o sort step; Afforest refers to the Connected Components (CC) step; Spoke-sizing computes the sizes of spokes using a MapReduction; Spoke-sorting aggregates and sorts the spokes; and Degree-decrement uses a BlockReduction to decrement the degrees of vertices. Other refers to time spent during initialization and clearing of metadata during execution.

Figure 5.1 Potential race condition due to parallel processing of edges using the Hilbert curve. (Figure adapted from [43]).
Figure 5.2  Locality of access to shared vertex data by HMerge [97]. Subfigure (a) shows the adjacency matrix of a social network. Each pixel in the \((i, j)\)th coordinate corresponds to the edge \((i, j)\) in the graph. Subfigure (b) shows the division of the Hilbert-ordered edges between four threads. Edges are coloured according to their assigned threads. Although each thread processes roughly the same number of edges, the area of the adjacency matrix covered by each thread varies significantly due to the uneven distribution of edges in the matrix. For example, a clustering of edges in the upper-left quadrant of the matrix corresponds to the small area covered by Thread 1. The left and top rectangular axes of Subfigures (b-c) highlight the regions of the vertex data array from which each thread will read and to which each thread will write, respectively. Subfigure (c) overlays the Hilbert curve (starting from the top-left) over Subfigure (b) to show the order of edge traversal for each thread. The start and endpoints of each thread’s edgelist segment are annotated. While there is spatial overlap in the vertex data regions accessed by each thread, there is no guarantee that the threads will access the overlapping vertices in the shared data array at the same time (i.e., temporal overlap), which can lead to competitive sharing of the cache.

Figure 5.3  Uneven distribution of edges due to quadrant partitioning.

Figure 5.4  Parallel Recursive Hilbert Blocking using OpenMP tasks.

Figure 5.5  The recursive divide-and-conquer function of Recursive Hilbert Blocking, parallelized using OpenMP tasks.

Figure 5.6  OpenMP static vs. dynamic schedules for static and dynamic workloads and different chunksizes. On the left of Plots (a) and (b), we show static and dynamic input workloads, split into 32 iterations, where each block represents an iteration. The amount of work per block is proportional to the darker filled area of that block. Each thread assignment corresponds to a schedule type and chunksize. For example, “Static, 4” uses a Static schedule with a chunksize of 4. Not specifying a chunksize is equivalent to using the OpenMP default. On the right of each plot, we show the cumulative amount of work per thread for an input workload, schedule, and chunksize.

Figure 6.1  Preprocessing and execution times for PageRank on twitter using Xeon with 14 cores as a function of the maximum number of edges per block, \(\mu\), dynamic chunk size, \(\delta\), and Spray Block Width. All times are rounded to one decimal place. Cells with blue borders denote the fastest execution time, omitting the preprocessing time (shown on the left). The cell with the red border denotes the fastest overall execution time.
Figure 6.2 Recursive Hilbert Blocks of wikipedia link en with SlashBurn and Descending Degree Sort vertex ID assignments, color-coded by the number of edges within each block (maximum 131,072 edges per block). Both Rhubarb and Descending Degree Sort cluster the majority of the edges in the graph in top-left quadrant of the adjacency matrix. Rhubarb leverages this clustering by dividing the dense region into smaller blocks than those in the sparser regions of the matrix. This facilitates parallel traversal using the available cores with a finer granularity of core-to-block assignment compared to the sparser regions of the graph. Specifically, since Rhubarb uses a constant block chunksize for thread assignment, concurrent threads that process the blocks in the dense upper-left quadrant share the read and write working set for the vertex data of the hub vertices. The sparser regions in the adjacency matrix, represented by lighter and larger blocks, are assigned to separate threads through individual, non-overlapping BlockReduction regions. This reduces the memory overhead required per iteration to aggregate thread-private contributions from all threads.

Figure 6.3 Recursive Hilbert Blocks of wikipedia link en with Original and Rabbit-order vertex ID assignments, color-coded by the number of edges within each block (maximum 131,072 edges per block). The figure shows edge clustering in the upper-left quadrant of the Original adjacency matrix. Rabbit-order uses the detected communities to reorder vertices, shown in the clustered blocks along the diagonal. Using Rabbit-order for CC in Rhubarb proves suboptimal since it increases the number of iterations required for convergence. In the original order, the lower ID vertices account for the majority of the edges in the graph. Rabbit-order relabels these hub vertices and assigns them IDs that are closer to the IDs of the vertices in their communities. Since many of these hierarchical communities are themselves connected to the other communities in the graph, Rabbit-order causes more frequent minimum Component ID relabellings as the Shiloach-Vishkin algorithm progresses, which directly translates to an increased number of iterations to converge (58 with Rabbit vs. 3 with Original), and a longer runtime.

Figure 6.4 PageRank (PR) and Connected Components (CC) total runtimes for sx-stackoverflow and soc-LiveJournal using Xeon with 14 cores. Preprocessing and execution times are shown as darker and lighter bars, respectively.

Figure 6.5 PageRank (PR) and Connected Components (CC) total runtimes for dimacs9-USA and wikipedia link en using Xeon with 14 cores. Preprocessing and execution times are shown as darker and lighter bars, respectively. GraphMat's Connected Components runtime on dimacs9-USA exceeded 5 minutes, so we omit it from the plot.
Figure 6.6 PageRank (PR) and Connected Components (CC) total runtimes for twitter and sd-arc using Xeon with 14 cores. Preprocessing and execution times are shown as darker and lighter bars, respectively. Both GraphMat and Rabbit Order ran out of memory trying to preprocess the larger sd-arc graph on our machine with 128 GB of RAM.

Figure 6.7 PageRank (PR) and Connected Components (CC) total runtimes for friendster using Xeon with 14 cores. Preprocessing and execution times are shown as darker and lighter bars, respectively. Both GraphMat and Rabbit Order ran out of memory trying to preprocess the larger friendster graph on our machine with 128 GB of RAM.

Figure 6.8 Tradeoffs between runtime, accuracy, and latent factor dimension ($k$). Although we use Graph-Based Benchmark Suite (GBBS) for this evaluation, we verify that given the same hyperparameter configuration (e.g., dimension of latent vector, stepsize, and regularization constant), all systems exhibit the same error curves. Figures 6.8a and 6.8b show that, for both bipartite graphs ys and ml, as $k$ grows, the time required for Collaborative Filtering grows linearly, while the final error decreases exponentially.

Figure 6.9 Tradeoffs between runtime, accuracy, and latent factor dimension ($k$). Figures 6.9a and 6.9b present the normalized mean squared error per iteration of Collaborative Filtering for the same graphs and values of $k$. A larger $k$ allows the latent factors to better represent the user-item interactions in the ratings datasets, resulting in a lower initial error. Although larger $k$ values correspond to longer execution times per iteration, increasing the dimensionality of the latent factors enables faster training. For example, the error curve for $k = 48$ in Figure 6.9a takes approximately 400 iterations to achieve the minimal error for this hyperparameter configuration. In Figure 6.9b, although the error curve for $k = 8$ has a higher initial error than $k = \in [5, 6, 7]$, it eventually achieves the lowest error among all values of $k$. For both graphs, achieving the same accuracy using fewer dimensions is either impossible or would require a significantly greater number of Gradient Descent iterations.

Figure 6.10 Collaborative filtering runtimes using Xeon with 14 cores.

Figure 6.11 Runtime and Parallel Scaling of 20 iterations of PageRank on twitter using the Original and Descending Degree Sort Vertex Orderings for \{1, 2, 4, 8, 16, 32\} cores on AWS.

Figure 6.12 Runtime and Parallel Scaling for 20 iterations of PageRank on twitter using the SlashBurn and Rabbit Vertex Orderings for \{1, 2, 4, 8, 16, 32\} cores on AWS.

Figure 6.13 Recursive Hilbert Blocks and distribution of block sidelengths for twitter using the Original Vertex Ordering.

Figure 6.14 Recursive Hilbert Blocks and distribution of block sidelengths for twitter using the Rabbit Vertex Ordering.
Figure 6.15  Recursive Hilbert Blocks and distribution of block sidelengths for twitter using the Descending Degree Sort Vertex Ordering.  

Figure 6.16  Recursive Hilbert Blocks and distribution of block sidelengths for twitter using the Parallel SlashBurn Vertex Ordering.  

Figure 6.17  Cache miss rates for 20 iterations of PageRank on twitter using Pull-Direction and Hilbert Edge Orderings, with 1 and 14 cores on Xeon. Each bar represents an edge-ordering (Pull vs. Hilbert) and number of cores. “Hilbert-14” refers to using Rhubarb with 14 cores. When using a single core, we see that the Hilbert curve’s locality preservation property leads to significantly fewer L2 and Last Level Cache (LLC) cache misses, resulting in faster runtimes in our single-threaded case study. For the multicore experiments, we measure the miss rate for each core separately and report the mean and standard deviation. Even after accounting for the standard deviation, using the Hilbert curve and all cores results in a decrease in L2 cache miss rate. This improvement is due to the partitioning of Hilbert-ordered edges and the grouping of contiguous Hilbert-ordered blocks using Rhubarb’s Recursive Hilbert Blocking and dynamic scheduling, which reduces the range of reads and writes assigned to individual cores. This trend does not extend to the LLC. While the multicore Hilbert LLC miss rate is substantially lower than the multicore pull miss rate, it is nearly double than the single-core Hilbert LLC miss rate. While Rhubarb mitigates the competition for the LLC by assigning consecutive groups of blocks to cores, the concurrent access of multiple cores to shared vertex data fills up the shared cache, resulting in a slightly higher number of misses.  

Figure 7.1  The top plots show the Recursive Hilbert Blocks of wikipedia_link_en with the SlashBurn vertex ID assignment. Blocks whose density is \( \geq \rho = 10^{-6} \) and whose sidelength is \( \leq 65,536 \) are color-coded by their density. All other sparse blocks are blank. The top-left plot shows the entire adjacency matrix, and the top-right plot zooms in on the region containing the majority of the densest blocks. The bottom plot shows the distribution of Recursive Hilbert Blocks sidelengths by anchor x-coordinate. Bars corresponding to block sidelengths that are greater than 65,536 are hatched.  

Figure 7.2  Basic vs. Greedy \( k \)-hub selection. Using Basic \( k \)-hub selection and \( k = 6 \), the vertices labelled with 1 – 5 are removed. Vertex 6 is then selected as the next hub due to its high degree in the original graph, even though its degree is 0. Greedy \( k \)-hub selection addresses this “incorrect” removal by updating degrees sequentially. (Figure adapted from [66]).
Figure A.1 Preprocessing and execution times for 77 iterations of PageRank on sx-stackoverflow as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.

Figure A.2 Preprocessing and execution times for 77 iterations of PageRank on sx-stackoverflow as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.

Figure A.3 Preprocessing and execution times for 91 iterations of PageRank on dimacs9-USA as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.

Figure A.4 Preprocessing and execution times for 91 iterations of PageRank on dimacs9-USA as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.

Figure A.5 Preprocessing and execution times for 69 iterations of PageRank on wikipedia-link_en as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.

Figure A.6 Preprocessing and execution times for 69 iterations of PageRank on wikipedia-link_en as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.

Figure A.7 Preprocessing and execution times for 83 iterations of PageRank on twitter as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.

Figure A.8 Preprocessing and execution times for 83 iterations of PageRank on twitter as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.
## List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>The adjacency matrix of the directed graph, $G$</td>
<td></td>
</tr>
<tr>
<td>$AID_v$</td>
<td>The Neighbour-to-Neighbour Average ID Distance of vertex $v$</td>
<td>$\sum_{i=2}^{N(v)}</td>
</tr>
<tr>
<td>$B(V_1, V_2, E)$</td>
<td>A bipartite graph $B$, with disjoint and independent vertex sets $V_1, V_2$, and edge set $E \subseteq V_1 \times V_2$</td>
<td></td>
</tr>
<tr>
<td>$b_{i,j}^s$</td>
<td>The Recursive Hilbert Block anchored at $(i, j)$ with a sidelength $s$</td>
<td></td>
</tr>
<tr>
<td>$\bar{d}$</td>
<td>The average degree of the graph</td>
<td>$\frac{m}{n}$</td>
</tr>
<tr>
<td>$deg^{+/−}(u)$</td>
<td>The out/in-degree of vertex $u$</td>
<td>$</td>
</tr>
<tr>
<td>$deg(u)$</td>
<td>The degree of vertex $u$</td>
<td>$</td>
</tr>
<tr>
<td>$E$</td>
<td>Edge Set</td>
<td></td>
</tr>
<tr>
<td>$(e_1, \ldots, e_m)$</td>
<td>An ordered list of $m$ pairwise distinct (unique) edges</td>
<td></td>
</tr>
<tr>
<td>$f(G)$</td>
<td>The fill (edge density) of a graph</td>
<td>$\frac{m}{n(n−1)}$</td>
</tr>
<tr>
<td>$G(V, E)$</td>
<td>A graph $G$, with vertex set $V$, and edge set $E$</td>
<td></td>
</tr>
<tr>
<td>$G(V, E, w(E))$</td>
<td>A weighted graph $G$, with vertex set $V$, edge set $E$, and a weight function $w(E)$</td>
<td></td>
</tr>
<tr>
<td>$h(u, v), h$</td>
<td>The Hilbert Index of the edge $(u, v)$</td>
<td>Assuming 32 bit unsigned int, $h(u, v)$ will be a 64 bit unsigned int</td>
</tr>
<tr>
<td>$k$</td>
<td>The number of hubs removed at each iteration of SlashBurn</td>
<td>$k = pn$, where $p$ is a user-defined hyperparameter. e.g., $p = 0.005$, from [66]</td>
</tr>
<tr>
<td>$m$</td>
<td>The number of edges in a graph</td>
<td>$</td>
</tr>
<tr>
<td>$N^{+/-}(u)$</td>
<td>The out/in-neighbourhood of vertex $u$</td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td>Definition</td>
<td>Notes</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>-------</td>
</tr>
<tr>
<td>$N(u)$</td>
<td>The neighbourhood of vertex $u$</td>
<td>$N^+(u) \cup N^-(u)$</td>
</tr>
<tr>
<td>$n$</td>
<td>The number of vertices in a graph</td>
<td>$</td>
</tr>
<tr>
<td>$s$</td>
<td>The sidelength of a block for blocked Sparse Matrix-Vector Multiplication</td>
<td></td>
</tr>
<tr>
<td>$(u,v)$</td>
<td>The edge between vertices $u,v$</td>
<td>May be directed or undirected</td>
</tr>
<tr>
<td>$u_{e_i}$</td>
<td>The source vertex of edge $e_i$ in the ordered edge list $(e_1,\ldots,e_m)$</td>
<td></td>
</tr>
<tr>
<td>$V$</td>
<td>Vertex Set</td>
<td></td>
</tr>
<tr>
<td>$v_{e_i}$</td>
<td>The destination vertex of edge $e_i$ in the ordered edge list $(e_1,\ldots,e_m)$</td>
<td></td>
</tr>
<tr>
<td>$y = Ax$</td>
<td>$A$ a sparse matrix (graph), $x$ dense input vector, $y$ dense output vector</td>
<td></td>
</tr>
<tr>
<td>$\delta$</td>
<td>The dynamic chunksize used by the OpenMP dynamic scheduler of Rhubarb</td>
<td></td>
</tr>
<tr>
<td>$\mu$</td>
<td>The maximum number of edges in a Recursive Hilbert Block</td>
<td></td>
</tr>
</tbody>
</table>
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AID</td>
<td>Neighbour-to-Neighbour Average ID Distance</td>
</tr>
<tr>
<td>ARD</td>
<td>Average Read Distance</td>
</tr>
<tr>
<td>ARWD</td>
<td>Average Read/Write Distance</td>
</tr>
<tr>
<td>AWD</td>
<td>Average Write Distance</td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth-First Search</td>
</tr>
<tr>
<td>CC</td>
<td>Connected Components</td>
</tr>
<tr>
<td>CSC</td>
<td>Compressed Sparse Column</td>
</tr>
<tr>
<td>CSR</td>
<td>Compressed Sparse Row</td>
</tr>
<tr>
<td>DCSC</td>
<td>Doubly Compressed Sparse Column</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth-First Search</td>
</tr>
<tr>
<td>GBBS</td>
<td>Graph-Based Benchmark Suite</td>
</tr>
<tr>
<td>GCC</td>
<td>Giant Connected Component</td>
</tr>
<tr>
<td>GD</td>
<td>Gradient Descent</td>
</tr>
<tr>
<td>GPS</td>
<td>Graph Processing System</td>
</tr>
<tr>
<td>HSFC</td>
<td>Hilbert Space Filling Curve</td>
</tr>
<tr>
<td>LLC</td>
<td>Last Level Cache</td>
</tr>
<tr>
<td>MEPS</td>
<td>Millions of Edges per Second</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Accesss Memory</td>
</tr>
<tr>
<td>RSS</td>
<td>Resident Set Size</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>SPMV</td>
<td>Sparse Matrix-Vector Multiplication</td>
</tr>
<tr>
<td>SSSP</td>
<td>Single Source Shortest Path</td>
</tr>
<tr>
<td>SV</td>
<td>Shiloach-Vishkin</td>
</tr>
</tbody>
</table>

xxi
Acknowledgments

I would like to express my heartfelt gratitude to the following individuals, without whom I would not have completed my thesis.

Puneet, I fondly remember our first chat on the bench in front of ICICS, which was a wonderful start to what ended up being a personal and professional relationship that I cherish. Your openness to collaboration, our countless conversations (technical or otherwise), and your support were crucial in helping me reach this point. Thank you for always being keen to chat.

Milad, it feels like we’ve shared so much of our respective journeys - the “Graph Calculator” project seems like it was ages ago. Thank you for always being open to discussing and sharing your technical expertise, even while being on the other side of the continent.

To my supervisor, Margo, I am indebted for your guidance. It is daunting to think how different I was as a person and researcher at the beginning of my thesis. Thank you for your sincere openness to my speculative (and possibly meandering) ideas of where my research should go, and what my thesis should be about. Our weekly meetings kept pushing me forward, refining and improving what ended up being this work. It has been a pleasure and an honour to be supervised by you, and I am thankful that you were always receptive and open to explore, while still offering helpful critiques and pushing back and asking me the hard questions that made me a stronger researcher - this research would not have been possible without you.

To Arpan, thank you for reviewing this dissertation and offering helpful comments and revisions to improve the quality of this work.

To my friends and family, for providing emotional support and encouragement in every step along the way. To Vadim, Andrei, and Shane, our camping trips to Bowron, Clayoquot, and Albert Edward punctuated this journey with much-needed respites into nature, where I was able to take a step back from work to take the time for introspection and come back rejuvenated with the energy needed to complete this effort.

And to my mother, for your encouragement on my academic journey, from Carleton to UBC. Our calls and your advice and guidance have been invaluable to me and have always kept me going. Thank you for always being there for me.

Finally, I am truly lucky and privileged to have been able to call Systopia my home for the past three years. Thanks to the Faculty and Students, this lab ended up being way more than my place of work and study, but was also my social hub and a source of inspiration.

This research was enabled in part by support provided by Compute Canada. We acknowledge the
support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Nous remerciions le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) de son soutien.
Dedication

To my mother, Sveta.
Chapter 1

Introduction

A variety of applications rely on graph structured data because it naturally models ubiquitous concepts such as social networks, protein structures, and supply chains [50, 95]. The pervasiveness of graph data has motivated the development of Graph Processing Systems (GPS) whose goal is to process analytic queries such as PageRank, Shortest Paths, or Connected Components. As graph data proliferates through different applications and domains the scale of graph datasets increases; modern systems process datasets with trillions of entries [67]. To speedup graph processing on these tera-scale graphs, previous work accelerated graph processing by modifying the graph’s data layout in memory [7, 40, 71] or on disk [64, 67]. As every graph fundamentally consists of two components - the vertices representing entities and the edges representing interactions between these entities - GPS typically process graphs by operating on either the vertices or edges of the graph. It is these components that are often reordered to improve the runtime of graph analysis.

In the Vertex-centric model, graph algorithms are expressed through user-defined vertex programs that compute values associated with a vertex. These programs execute iteratively on each vertex in the graph. In each iteration, a vertex executes its assigned vertex program and exchanges messages between its neighbours to propagate updated vertex values. To enhance the memory access locality of vertices, Vertex-centric systems use vertex reordering as a preprocessing optimization. A vertex reordering is a function that maps each vertex ID to a new ID within the range \([0, n]\), where \(n\) is the number of vertices in the graph. This mapping, also known as a graph isomorphism, preserves all edges between vertices and maintains the graph’s structure. However, vertex reordering can introduce structure to the adjacency matrix of the graph. For instance, Figure 1.1 illustrates the adjacency matrix of a social network. The graph was preprocessed using three different vertex orders: a random vertex ID assignment, the IDs calculated using Rabbit-order [7], and those calculated using the SlashBurn [66] order.
Figure 1.1: Adjacency matrices of a directed, user-to-user, trust network with 4,658 users and 40,133 trust ratings [60]. Each pixel denotes a directed edge (“trusts” relationship) between a pair of vertices (users) in the graph. We ignore edge weights (ratings). Note the detected dense communities (submatrices) along the diagonal for Rabbit-Order and the dense clustering of edges for SlashBurn.

Some Vertex-centric systems [40, 97] use the degrees of the vertices to cluster and/or sort the vertices (e.g., sorting the vertices by descending order of degree). Since high degree vertices (also known as “Hub” vertices) are, by definition, neighbours of a large number of vertices, these hubs are accessed frequently when iterating over the edge set of the graph [10]. Sorting the vertices by descending degree collocates these hub vertices in memory and increases the likelihood of frequently accessed vertices being cached. Alternatively, Rabbit Order [7] 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. Rabbit Order orders the vertices so that consecutive vertex IDs correspond to meaningful communities. The algorithm first detects these communities and then labels the vertices according to those communities.

In the Edge-centric model, we compute analytic properties of graphs by iterating over the edges. For each edge in the graph, we apply an update to either the source or destination vertex incident to that edge. To speedup graph kernels such as PageRank, Edge-centric systems use edge ordering to address the random-access pattern of incident vertices, which is commonly observed in many graph processing tasks. Edge-centric systems can order the edges of the graph by ascending Source or Destination ID, effectively iterating over the graph’s adjacency matrix in Row-major or Column-major order, respectively. When traversing the adjacency matrix in Row-major order, we achieve excellent locality in the source vertices, since all outgoing neighbours of a source vertex are processed before moving on to the next source vertex. But, at the same time, Row-major order may result in near-random memory accesses of the vertex array when accessing the destination vertex of each outgoing edge. Prior work [71] addressed this concern by using an edge ordering defined by the Hilbert Space Filling Curve [52]. The Hilbert Curve is a method of assigning indices to the edges of a graph that provides locality in both the source and destination vertices, thereby improving the overall efficiency of Edge-centric systems during graph processing.

Until now, researchers have primarily focused on either vertex or edge ordering to enhance the performance of their Vertex-centric or Edge-centric systems. However, there exists no comprehensive evaluation of potential vertex-and-edge orderings. We define a vertex-and-edge ordering of a graph as
a preprocessing pipeline that:

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

2. Relabels the edges of the graph using the new vertex IDs, and

3. Reorders the edges of the graph.

This thesis addresses the existing knowledge gap and investigates the interaction between Vertex and Edge orderings. We first ask whether the benefits of these orderings compound. That is, it has been shown that using either a vertex ordering or an edge ordering accelerates graph processing. So, do we see a combined performance improvement by using both techniques? To answer this question, we conduct a comprehensive performance study of 18 vertex-and-edge orderings using a dataset we construct from the cross product of 45 graphs (31 real-world and 14 synthetic), 6 vertex orders, and 3 edge orders. We discover that a specific vertex-and-edge ordering, using the SlashBurn vertex order [66] and the Hilbert edge order [52, 71], consistently yields the fastest runtime for 30 out of 45 graphs in our dataset. However, we also encounter two main issues with the combined use of these orderings:

1. The SlashBurn vertex order leads to notable performance improvements for computing the PageR-rank of different graphs in our dataset. However, the implementation of the SlashBurn vertex ordering is sequential, resulting in significant computational overhead that offsets the performance gains achieved by the preprocessing technique.

2. While computing the Hilbert edge ordering is trivially parallelizable, incorporating the Hilbert edge order into parallel graph processing applications is challenging. Previous attempts to parallelize edge-centric processing using the Hilbert curve [97] resulted in suboptimal performance due to poor scalability.

In response to these two issues, we present the two novel contributions of this thesis:

1. We develop a parallel implementation of the SlashBurn algorithm that optimizes and parallelizes crucial subroutines, namely, degree decrements and connected components computation. Using a single core, our optimizations deliver a notable speedup over the existing sequential implementation, ranging from $4.33 \times -11.96 \times$. Moreover, our implementation is scalable, achieving a parallel speedup of $6.78 \times -8.19 \times$ when using 14 cores.

2. We introduce RHuBarb, an edge ordering and blocking technique that uses Recursive Hilbert Blocking to accelerate parallel, edge-centric graph algorithms. RHuBarb is highly scalable, achieving an impressive parallel speedup of $25.82 \times$ with 32 cores. RHuBarb outperforms state-of-the-art PageRank implementations from Ligra [84], GraphMat [89], and GBBS [33], but due to significant preprocessing overhead, does not provide an overall speedup compared to these systems. However, for the Collaborative Filtering application, RHuBarb does offer end-to-end speedups of $1.22 \times - 2.33 \times$ over GBBS, $1.05 \times - 2.64 \times$ over GraphMat, and $1.89 \times - 2.61 \times$ over Ligra for the yahoo-song and movielens datasets.
Chapter 2 provides essential background information and introduces the notation we use throughout this thesis. We review common data structures used for graph representation, explore typical modes of computation used in parallel graph processing, discuss the challenges commonly encountered in this domain, and explain how graph ordering is used to address these challenges. In Chapter 3, we present our case study on vertex-and-edge ordering performance and discuss the results that serve as the motivation for Chapters 4 and 5. Chapter 4 delves into our parallel implementation of SlashBurn. We describe the adaptations made to the key subroutines of SlashBurn to enable parallel execution and evaluate the resulting performance improvements. Chapter 5 introduces Rhubarb and Recursive Hilbert Blocking. We outline the Recursive Hilbert Blocking algorithm and explain how we parallelize edge-centric computation in Rhubarb using Sparse Array Reductions [54] and dynamic thread scheduling. We evaluate Rhubarb in Chapter 6, where we describe the implementation of parallel, edge-centric PageRank, Connected Components, and Collaborative Filtering using Rhubarb and compare the performance and scalability of our implementation against Ligra [84], GraphMat [89], and the Graph-Based Benchmark Suite (GBBS) [33]. Chapter 7 distinguishes Rhubarb from previous systems that used the Hilbert curve, identifies limitations in our approach, and proposes potential improvements for both Rhubarb and Parallel SlashBurn. Finally, Chapter 8 serves as the conclusion of this thesis, summarizing our key findings and contributions.
Chapter 2

Background

We begin by introducing common data structures that are used to represent graphs in memory. We define our notation in the List of Symbols on p. xix. In Section 2.2, we review the conventional graph data structures: the Adjacency List and Adjacency Matrix, and show how they are inadequate for representing large, real-world, sparse graphs. Compressed graph representations such as Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) address these limitations. In Section 2.3, we introduce the different modes of parallel graph computation and illustrate the challenges that arise when using the CSR representation for parallel graph processing. We conclude by defining graph ordering, distinguishing between two common methods: Vertex and Edge ordering, and showing how graph ordering can address these challenges.
2.1 Preliminaries

A graph $G(V,E)$ is an ordered pair of a vertex set, $V$, and a subset of edges $E$ from $V \times V$. Graphs may be directed and/or weighted. If a graph is directed, $E$ consists of ordered pairs of vertices, whereas if $G$ is undirected, edges are unordered. $n$ is the number of vertices in the graph ($|V|$) and $m$ is the number of edges in the graph ($|E|$). We assume vertex IDs lie in the range: $[0,n)$. Given a directed graph and an edge $e(u,v)$ such that $u,v \in V$, we say that $v$ is an out-neighbour of $u$; $u$ is an in-neighbour of $v$; $u$ and $v$ are adjacent; and $u$ and $v$ are incident to the edge $(u,v)$. Unless explicitly stated, we treat all graphs as directed in the remainder of this thesis. If $G$ is a weighted graph, a weight function, $w(E) \rightarrow \mathbb{R}$, maps each edge in the graph to a real value. A graph $B$ is said to be bipartite if $V$ can be decomposed into two disjoint vertex sets $V_1,V_2$, such that no two vertices within a vertex set are adjacent. A tree is an undirected graph in which any two vertices are connected by exactly one path [16]. A forest is a disjoint union of trees [16]. Finally, graphs $G$ and $H$ are isomorphic if there is a bijection between the vertex sets of $G$ and $H$

$$f : V(G) \rightarrow V(H)$$

such that any two vertices $u$ and $v$ of $G$ are adjacent in $G$ iff $f(u)$ and $f(v)$ are adjacent in $H$ [94]. Figure 2.1a shows a directed graph, which we will use as a running example to show how the same graph is represented using either an Adjacency Matrix, Adjacency List, or the CSR / CSC representations.
A graph is *scale-free* if the degree distribution of its vertices follows a *power-law*, where a small fraction of vertices are incident to a disproportionately large fraction of the edges of the graph. The vertices of power-law graphs are typically categorized according to their degree. We follow the convention adopted by Esfahani et al. [37]: the *average degree* of the graph ($\bar{d}$) acts as the threshold between *low-degree* and *high-degree* vertices, and vertices whose degree is larger than $\sqrt{n}$ are called *hub* vertices.

A graph is *sparse* if the number of edges in the graph is substantially smaller than the total number of possible edges (i.e., $m \ll n(n-1)$).

### 2.2 In Memory Graph Data Structures

A graph can be represented using an **Adjacency Matrix**, $A$ (Figure 2.1b), with $A_{i,j}$ corresponding to the edge $(i, j)$. If the graph is unweighted, $A$ is an $n \times n$ Boolean matrix, with $A_{i,j} = 1$ indicating the existence of an edge $(i, j)$. If the graph is weighted, the nonzero values of $A \in \mathbb{R}^{n \times n}$ correspond to the weight function, $w(E)$. The adjacency matrices of real world graphs are themselves sparse matrices, with a majority of the entries in the matrix containing zero, or null, values (we will often use the terms “graph”...
and “sparse matrix” interchangeably). This adjacency-matrix representation led some researchers to frame graph processing in the form of matrix operations such as Sparse Matrix-Vector Multiplication [89]. For example, the out- (in-) degrees of the vertices in the graph can be computed using the matrix-vector product: \( A^T (A^T 1^T) \). However, representing real world graphs as sparse matrices is usually infeasible, due to the \( O(n^2) \) memory footprint of the entire adjacency matrix.

Alternatively, a graph can be represented using an Adjacency List (Figure 2.1c), with the vertices of the graph stored in an array. Depending on the graph application, the neighbourhood of vertex \( u \) (either \( N^+(u) \) or \( N^-(u) \) for the out- or in-neighbourhood of \( u \)) are stored in an array, linked list, or set data structure [87]. The neighbours of each vertex are often sorted by ascending Vertex ID. The memory requirements of the adjacency list \( O(n + m) \) are lesser than that of the adjacency matrix, which is particularly crucial when dealing with sparse graphs. However, using a linked list to represent vertex neighbourhoods raises an issue. Graph algorithms typically rely on the repeated traversal of the neighbourhoods of vertices (e.g., Breadth-First Search (BFS), Single Source Shortest Path (SSSP)). If the graph is stored using a linked list adjacency list, this traversal follows neighbouring pointers to iterate over the neighbourhood of each vertex. This poses problems for hardware prefetchers, since a prefetcher cannot prefetch the data of the next vertex until it has read the neighbour pointer of the current vertex. This “pointer-chasing” issue can be alleviated by laying out the neighbourhoods in a sorted, contiguous array.

The Compressed Sparse Row/Column (CSR/CSC) representations (Figure 2.1d) achieve this data layout by using two arrays, one to store vertices and one to store edges. The Offsets Array, \( OA \), is an array of size \( n + 1 \) that corresponds to the vertices of the graph. Vertices are stored in the Offsets Array in ascending order of their vertex ID. The Neighbours Array, \( NA \), is an array of size \( m \) that corresponds to the edges of the graph. To iterate over the neighbourhood of vertex \( u \), we first retrieve the beginning and end offset of \( u \)'s neighbourhood from \( OA[u] \) and \( OA[u + 1] \). These offsets tell us the range of indices in the Neighbours Array that contain the Vertex IDs of the neighbours of \( u \). Vertices within a neighbourhood are sorted by ascending order of their vertex ID. If the graph is weighted, an additional Weights Array, \( WA \), of size \( m \) stores edge weights. A CSR stores the out-edges of the graph, while a CSC stores the in-edges of the graph. Using a compressed representation to store the graph achieves the benefits of both the adjacency matrix and adjacency list. First, fast Matrix-Vector operations are enabled, since we can access the nonzero values of row \( i \) (stored in \( NA[OA[i]:OA[i + 1]] \)) sequentially and in constant time. Second, we no longer require any pointer chasing to iterate over the neighbourhood of vertex \( i \) (also stored in \( NA[OA[i]:OA[i + 1]] \)).

2.3 Memory Access Patterns and Bottlenecks in Graph Processing

Due to its memory efficiency, the CSR representation has become the de-facto data structure for storing graphs in-memory. However, traversals of sparse graphs suffer from poor-locality when using the CSR representation, and prior work shows that graph algorithms on CSRs spend a majority of execution time stalled on memory accesses [96].

Consider Algorithm 1, which is an example of a common graph traversal, where we iterate over
the out-neighbourhoods of the vertices of the graph and retrieve metadata for each vertex. For each out-neighbour, we access an array of size $n$, $v_{\text{data}}$, that stores application-specific vertex data.

**Algorithm 1** Out-neighbourhood Graph Traversal

```plaintext
1: for $u \in V$: do
2:     for $v \in N^+(u)$: do
3:         work($v_{\text{data}}[v]$)
```

Figure 2.2 illustrates the irregularity of access of this common graph traversal pattern.

**Figure 2.2:** Irregular access to the $v_{\text{data}}$ array by traversing the out-neighbourhood of $u$. The circled numbers refer to the order in which elements from $v_{\text{data}}$ are retrieved. Due to the distance between the IDs of $u$'s out-neighbours, this traversal suffers from poor spatial locality. Figure adapted from [10].

Using this figure as reference, we can distinguish between two different types of locality:

1. **Spatial Locality:** a vertex $u$’s neighbours exhibit spatial locality if the ID range of its neighbours is consecutive and/or contiguous. That is, we require a minimal number of cache lines to retrieve the metadata for the vertices adjacent to $u$. For example, in Figure 2.2, we can retrieve the metadata for $N^+(u)$ using 4 cache lines, but if $u$’s neighbours were packed closely together, it would be possible to retrieve all of their metadata using a single cache line.

2. **Temporal Locality:** a pair of vertices $u, v$ exhibit temporal locality if there exists substantial overlap between $N(u)$ and $N(v)$. This overlap implies that the vertices adjacent to both $u$ and $v$ are likely to be retained in the shared cache after accessing the neighborhood of one vertex, making it more efficient to access the neighborhood of the other vertex. For example, $u, v$ exhibit perfect temporal locality if $N(u) = N(v)$.

While locality of access dictates the performance of graph applications, the size of application-specific vertex data determines the volume of memory traffic. For example, a BFS accesses only the IDs
of its neighbours whereas applications such as Collaborative Filtering access and update large vectors of per-vertex data [10]. The issue of poor locality is exacerbated in the context of modern Graph Processing Systems (GPS) that use multiple threads to iterate over the graph concurrently. To explain exactly how this issue is exacerbated, we discuss how GPS typically perform parallel graph analytics by picking a Vertex Partitioning strategy and Mode of Computation.

### 2.3.1 Vertex Partitioning for Parallel Graph Processing

To iterate over the vertices of the graph in parallel, a GPS must choose a partitioning strategy to divide the vertices among the available threads. Given $t$ threads, Balanced Vertex Partitioning partitions the Vertex Set $V$ into $t$ disjoint sets of size $\frac{n}{t}$. During parallel iteration, thread $i$ processes the vertices whose IDs lie in the range $[i\frac{n}{t}, (i+1)\frac{n}{t})$. This simple partitioning is easy to compute but suffers from work imbalance when processing power-law graphs. The threads that process the vertex partitions that contain hub vertices process many more edges and therefore take a longer time to complete their traversal. This work imbalance causes other threads to wait idly, degrading parallel performance by not using all available threads at all times.

This work imbalance can be solved by using Balanced Edge Partitioning. Here, $V$ is partitioned into $t$ partitions, but the number of vertices in each partition may vary. Instead, we add vertices to a partition until the number of incident (either incoming/outgoing) edges in a partition reaches $m \frac{t}{7}$. This ensures that the amount of work that is assigned to each thread is roughly proportional. We note that the greedy and exclusive assignment of hub vertices to partitions may still result in potential imbalance between partitions. Prior work into producing balanced graph partitioning [47, 68] is extensive but out-of-scope for this dissertation.

### 2.3.2 Modes of Computation for Parallel Graph Processing

Once a partitioning strategy is selected, a GPS must decide the direction of traversal each thread should take when iterating over the neighbourhoods of its assigned vertices. The two natural modes of parallel graph computation are Push and Pull [12, 18, 84]. In pull mode, a thread iterates over the in-neighbours of a vertex in Line 2 of Algorithm 2 and pulls updates from a vertex’s neighbours to update the global $v_{data}$ array. In push mode a thread iterates over the out-neighbours in Line 2 of Algorithm 2 and pushes updates to $v_{data}$. Deciding on the mode of computation is application dependant, and some systems [12, 84] alternate between modes for iterative algorithms. It is important to note that in the context of parallel graph processing, there is a distinct advantage to using the pull mode. Specifically, since each thread is assigned a non-overlapping region of the $v_{data}$ array, in the pull mode, each thread can safely write to its assigned region without synchronization. However, in the push mode, threads can arbitrarily write to the same destination vertex’s metadata in the $v_{data}$ array, which typically requires a form of locking to ensure correctness of execution, and can hinder parallel performance and scalability.

Regardless of the choice of partitioning and mode of computation, parallel graph algorithms generally suffer from poor locality. As threads iterate over their assigned vertices, there is no guarantee of spatial and/or temporal locality within and between vertex neighbourhoods or threads. Poor locality
is exacerbated in modern, multicore architectures, where the different CPU cores share a Last Level Cache (LLC), which serves as the final level of cache in a multi-level cache system, residing between the CPU cores and the main memory Random Access Memory (RAM). Since threads can be operating concurrently on non-overlapping regions of \texttt{v.data}, it is likely they compete for the capacity of the shared LLC. This competition causes a large number of cache evictions and misses, expensive requests from RAM, and an overall degradation in parallel performance.

\begin{algorithm}
\caption{Parallel neighbourhood Graph Traversal}
\begin{algorithmic}[1]
\State \textbf{parallel} for $u \in V$ : \textbf{do}
\State \hspace{1em} for $v \in N(u)$ : \textbf{do}
\State \hspace{2em} $\text{work}(v_{\text{data}}[v])$
\State \hspace{1em} \textbf{end for}
\State \textbf{end parallel for}
\end{algorithmic}
\end{algorithm}

\subsection*{2.4 Graph Ordering}

To address the problems mentioned above, there has been a significant amount of research on Graph Ordering Techniques \cite{10, 38, 40, 57, 64, 66, 71}. These techniques rearrange either the vertices or edges of a graph to enhance memory access locality during graph traversal. We categorize and provide a few examples of vertex reordering techniques that address the poor locality of graph traversals by laying out the vertices of the graph using different heuristics and methods. Then, we introduce edge ordering and the Hilbert Space Filling Curve (HSFC), and explain how McSherry et al. \cite{71} used this technique to alleviate the poor locality of the push/pull modes of computation.

\subsubsection*{2.4.1 Vertex Reordering}

A \textit{Vertex Reordering} is a relabelling of vertex IDs that improves the access pattern for parallel graph traversal. It does not alter the graph structure. Reordering techniques exploit known properties of real-world graphs, such as a power-law degree-distribution \cite{11} and community structure \cite{44}. In graph processing, a \textit{community} refers to a subset of vertices that exhibit a higher degree of interconnectedness relative to the rest of the vertices in the graph. Communities are often characterized by dense intra-community connections and relatively sparse inter-community connections. Vertices in a community can be assigned contiguous IDs \cite{7}, given the higher likelihood that the vertices reference each other.

To calculate the performance improvement due to a vertex reordering, we must take into account the cost and complexity of the reordering computation since it acts as a preprocessing step to improve performance. So, \textit{lightweight} vertex reorderings that improve overall performance even after accounting for the overhead of preprocessing \cite{10} are preferred. Sophisticated algorithms with high computational costs might not be worth it if the downstream analytic tasks require only a few passes through the graph or a small number of iterations, which will not amortize the high cost of preprocessing. Below, we introduce a few common vertex orderings that have been shown to speed up graph analytics.
Degree-Sort

The degree-sort is the canonical example of a degree-based vertex reordering algorithm. It sorts the vertices of the graph by their degrees and assigns the vertices new IDs based on their index in either the ascending or descending degree order. A degree-sort groups the hub vertices together so that they fit in the smallest number of cache lines. This improves the temporal locality of access to the hub vertices: since most of the vertices of the graph are adjacent to hub vertices, accesses to the (now contiguous) region of the v.data array that references the hub vertices’ metadata can benefit from potential reuse as the hubs’ vertex metadata will more likely remain in the shared LLC cache. Certain graph algorithms (e.g., triangle counting) [35, 58] and GPS [26, 92] benefit from degree-sorting as a preprocessing step to optimize their performance.

Degree-based Methods

Faldu et al. [40] raise the following issue with degree-sort. Certain graph datasets have a “default” vertex ID assignment that yields high spatio-temporal locality. For example, hyperlink networks that were crawled by the Laboratory for Web Algorithmics use a lexicographical URL ordering to assign IDs to vertices [22]. This hierarchical ordering of vertices usefully groups together URLs that belong to the same domain and subdirectories. A degree-sort will destroy this locality present in the ordering of vertices by ignoring the hierarchical grouping of sites, using only the degree information of nodes to rearrange the vertices of the graph. To address this concern, researchers developed the following degree-based reorderings that preserve the structural properties of the original vertex ID assignment (if they exist) and cluster, and optionally reorder, the high-degree vertices of the graph.

Hub-sort [96] maintains the original vertex IDs of low-degree vertices, while clustering and sorting the high-degree vertices of the graph. As a result, accesses to the v.data array will now benefit from the same spatio-temporal benefits of the degree-sort, without perturbing the structure of the original input graph. Relatedly, Hub-cluster [10] clusters only the high-degree and low-degree vertices of the graph into separate regions of v.data, without reordering the vertices based on their degrees. This reordering closely packs the high-degree vertices, maintains the graph’s original structure, and incurs a lower reordering overhead since it does not require a sorting step. Figure 2.3 illustrates the degree-based vertex reorderings.
Figure 2.3: Degree-based Vertex Orderings. The top of the figure shows an undirected graph whose vertices are labelled using an arbitrary ordering. The arrays in Subfigures (a)-(d) store the degrees of the vertices and illustrate the different vertex ID assignments for the different vertex orderings. High-degree vertices are denoted by blue entries. Figure adapted from [10].

Rabbit-order

Rabbit-order is a vertex reordering that leverages the assumption that the input graph contains a hierarchical community structure with large communities of vertices that contain nested communities of more densely connected subgraphs. Rabbit-order identifies the hierarchical communities of the graph by iterating over the vertices in ascending degree order. Rabbit-order searches the neighbourhood of each vertex in an attempt to merge the neighbour that yields the greatest gain in Modularity, $Q$. Arai et al. [7] define the improvement in modularity gained by merging two adjacent vertices $u, v$ as:

$$
\Delta Q_{u,v} = 2 \left( \frac{w(u,v)}{2n} - \frac{\text{deg}(u)\text{deg}(v)}{2n^2} \right),
$$

where $w(u,v)$ is the weight of the edge $(u,v)$, $\text{deg}(u)$ is the degree of vertex $u$, and $n = |V|$. This iterative graph refinement repeats as long as there exists a neighbour $v$ adjacent to $u$ for which $\Delta Q_{u,v} > 0$. If $u$ has no neighbours to merge, it is added to a set of vertices that act as roots of communities. Rabbit-order completes the hierarchical community detection by performing a parallel Depth-First Search (DFS) starting from the root (top-level vertex) of each community. Once the vertices within each nested community
are identified, they are assigned consecutive vertex IDs. Arai et al. [7] claim that Rabbit-order improves locality by mapping hierarchical communities into hierarchical caches. The authors use lightweight atomic operations to incrementally compute the hierarchical communities, with the goal of reducing the runtime of the reordering and providing end-to-end speedup.

**SlashBurn**

SlashBurn relies on the ubiquity of hub vertices in real-world graphs and their role in connecting the connected components of a graph. SlashBurn is an iterative algorithm that consists of two phases:

1. **Slash**: identify the hub vertices of the graph, remove them from the graph, and assign the removed hubs consecutive vertex IDs in descending order of degree.

2. **Burn**: once the hub vertices have been deleted, (re)compute the new graph’s connected components. The graph’s connected components now consist of a Giant Connected Component (GCC) and potentially other, smaller, connected components, named spokes. Sort the spokes by descending order of size. Assign the vertices in the spokes new Vertex IDs based on the size of their respective connected component. Recursively repeat steps 1 and 2 on the GCC.

This iterative process continues until the number of hub vertices is smaller than \( k \), a user-defined hyperparameter. SlashBurn is unique among the vertex ordering techniques we have discussed in that it reorders the vertices of the graph to minimize the storage cost of the graph’s adjacency matrix. Specifically, Lim et al. [66] are interested in large-scale matrix-vector multiplication operations (See Section 2.2), which form the basis for graph algorithms such as PageRank, diameter estimation, and connected components [56]. At the time of publication, the state-of-the-art method for Sparse Matrix-Vector Multiplication was the block multiplication method, where \( A \) is split into \( l \times l \) square matrix blocks, the vector to be multiplied, \( x \), is split into vector blocks of length \( l \), and the matrix-vector blocks are multiplied [56]. The goal of SlashBurn is to produce a vertex ordering that clusters the edges of the graph and produces a compressed representation: a smaller number of denser blocks is better than a larger number of sparser blocks [66]. Among the vertex orderings we discussed above, SlashBurn is the only heavyweight vertex ordering, with a time complexity of \( O(m + n \log n)i \), where \( i \) is the number of iterations to compute the reordering. Figure 2.4 illustrates the effect of the different vertex reorderings we described in Section 2.4.1 on the adjacency matrix of a social network.

**2.4.2 Edge Reordering**

Next, we divert our attention away from vertex ordering to focus on edge ordering. We discuss the common methods of matrix traversals: Row-major and Column-major, introduce the Hilbert Space Filling Curve (HSFC), and discuss how McSherry et al. [71] used the Hilbert curve to improve the memory access locality of both source and destination vertices in edge-centric traversal.

The two common matrix traversals are Row-major and Column-major (Figure 2.5a., b., respectively).
Figure 2.4: The adjacency matrix of the CiaoDVD Social Network graph [60] after performing vertex reordering using the methods described in Section 2.4.1. A marked blue pixel in the \((i, j)\)th coordinate of the plot corresponds to the existence of an edge \((i, j)\) in the graph. Expectedly, the Random vertex ordering produces a sparse matrix with a uniform distribution of edges. The degree-based techniques (Descending Degree Sort, Hub-Cluster, Hub-Sort) all produce a concentration of edges in the upper-left of the adjacency matrix as a result of the colocation of the hub vertices of the graph. Note that the boundary between the low- and high-degree vertices of the graph in the Hub-Cluster and Hub-Sort plots produces close-to-identical lower-right quadrants. Rabbit-Order produces clusters along the diagonal of the matrix that correspond to the hierarchical communities in the graph. SlashBurn concentrates edges to the left, top, and diagonal (the wings and tail, respectively). This results in an adjacency matrix that requires the smallest number of square blocks to cover the edges among the vertex reorderings discussed. Lastly, the Original vertex ID assignment is far from random. While the method of vertex ID assignment is typically opaque to the users, and varies depending on the type of network being modelled, certain graph datasets use default vertex ID assignments that exhibit excellent locality (e.g., hyperlink networks from the Laboratory for Web Algorithmics [22]).
These are identical to the push and pull modes of computation described in Section 2.3.2. In the context of dense graphs (matrices), Row-major traversal traditionally dominates in terms of performance, due to C-style 2D arrays being laid out in this form in memory. This allows kernels that use row-major order to iterate over a dense matrix to benefit from compiler optimizations such as prefetching and vectorization. However, real-world graphs are rarely dense, and the sparser a graph is, the less likely that such optimizations will affect performance. Specifically, if we iterate over the edges of a sparse graph using the push mode, we experience excellent temporal locality in the dimension of the source vertices, but nearly random access in the dimension of the destination vertices, and vice versa for the pull mode.

McSherry et al. [71] observed this issue with vertex-centric traversals and used the Hilbert curve to order the edges of the graph, reporting improved single-threaded performance for edge-centric algorithms such as PageRank and Connected Components. Hilbert [52] first proposed the Hilbert curve as a continuous fractal space-filling curve. The Hilbert curve provides a bidirectional mapping between 2D and 1D space that preserves locality across the transformation: points that are close to each other in 2D space will remain close in 1D space. This property of locality preservation has led to the use of the Hilbert curve in different areas of computer science, such as computer graphics [31] and storage [63].

Reordering the edges of the graph using the Hilbert curve requires computing the Hilbert Index of each edge in the graph. The mapping of the edge \((u,v)\) to its Hilbert Index, \(h(u,v)\), can be done using a recursive procedure [23] whose complexity is \(O(\lg(n))\). The mapping can be thought of as an interleaving of the bits of \(u, v\), which produces the 1D coordinate that preserves the locality of \(u, v\) [71]. This mapping is computed for all the edges of the graph, and then the edges are sorted by ascending \(h\), for an overall complexity of \(O(m\lg m)\). The Hilbert curve sacrifices the excellent 1D locality of the traditional pull/push based traversals, but improves the locality in the dimensions of the source and destination vertices of the edges of the graph.

We introduced different techniques for reordering graphs, which can improve performance in various ways depending on the dataset and algorithm. Vertex reordering reorders vertices to improve temporal and spatial locality by grouping vertices that are more likely to reference each other, such as hubs and communities, together in memory. On the other hand, edge reordering using the Hilbert curve sacrifices sequential access in either the source or destination vertices but improves memory access locality for
edges in both source and destination dimensions. We note that these two techniques are not mutually exclusive. Combining vertex and edge ordering could potentially lead to more significant performance improvements, which naturally leads to the first Research Questions we answer in this thesis:

<table>
<thead>
<tr>
<th>Research Question 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do the benefits of Vertex and Edge orderings compound?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Research Question 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>For a given graph and algorithm, is there a vertex-and-edge ordering pair that outperforms all other vertex-and-edge orderings and produces significant performance improvements?</td>
</tr>
</tbody>
</table>
Chapter 3

Case Study: Single-Threaded
Performance Evaluation of
Vertex-and-Edge Orderings

We concluded the last chapter by asking whether the benefits of vertex and edge orderings compound. To answer Research Question 1, we conducted a performance case study on 45 graphs, six vertex orders, and three edge orders:

1. We downloaded 31 real-world graphs representing social, hyperlink, and road networks from the Konect Project [60] and generated 14 synthetic graphs.

2. We used the vertex orderings described in Section 2.4.1 on the graphs. This resulted in six distinct isomorphisms for each graph, including the one using the original vertex IDs.

3. We relabelled the edges of each graph using the six vertex orders.

4. We reordered the edges using the Row-major, Column-major, and Hilbert orderings.

5. By reordering both the vertices and edges, we obtained 18 representations for each graph. For each representation, we used the time to complete 20 iterations of PageRank to evaluate the efficacy of the representation.

6. We compared the execution times within each graph to identify combinations of vertex and edge orderings that consistently produced the best running time.

We named this pipeline of:

\[
\text{Vertex Reordering} \rightarrow \text{Edge Relabelling} \rightarrow \text{Edge Reordering}
\]

a vertex-and-edge ordering. To ensure the reliability of our findings, we established the following requirements for our chosen datasets and task:
**Requirement 1**

*Benchmark Invariablility:*

1. The benchmark should iterate over all the edges of the graph.

2. The benchmark should perform a constant number of iterations.

3. The amount of work per iteration should remain constant, regardless of the vertex or edge ordering.

While previous work has shown that vertex ordering can affect the convergence properties of graph analytic tasks such as Connected Components [10], we chose not to evaluate this effect in our case study and opted to evaluate the performance improvements gained by the improved locality conferred by vertex and edge ordering instead.

By using a benchmark that met these requirements, any performance differences between the experimental runs would be due only to the different vertex and edge orderings, rather than other properties of the input graph or the benchmark. For the sake of simplicity, we used a single-threaded benchmark implementation.

**Requirement 2**

*Dataset Representativeness:*

To accurately assess the combined effect of vertex and edge ordering, we selected a dataset of graphs that represent real-world applications of graph analytics. Due to an overrepresentation of hyperlink networks in our dataset, we included synthetic graphs as well.

**Requirement 3**

*Stressing Cache Capacity:*

After selecting the benchmark and dataset, we calculated the Resident Set Size of our benchmark as a function of the space complexity of the benchmark and the size of the graph. Our goal was to ensure that the Resident Set Size exceeded the size of the LLC. This choice was made to align our study with real-world scenarios commonly encountered in graph processing. In practice, many graph datasets and algorithms exhibit Resident Set Sizes that exceed the LLC. If the graph is small enough to fit in the CPU cache, the opportunity for performance improvement would be minimal, since repeated, costly memory requests from RAM are the main cause of performance degradation in graph traversals [96]. Our entire study is premised on the hypothesis that reordering the vertices and edges of a graph can reduce the number of cache misses. However, running the benchmark on a graph that fits in the CPU cache (shared or private) would not result in any expensive memory requests, rendering the graph unsuitable for our study. Therefore, we used graphs whose Resident Set Size exceeded the size of our LLC. We describe how we selected the graphs in Section 3.2.
3.1 Graph Analytic Task: PageRank

A natural candidate task for our benchmark was PageRank, proposed by Brin and Page [24], which ranks the relative importance of webpages. PageRank uses two hyperparameters: a damping factor $0 \leq \gamma \leq 1$ and a convergence constant $\varepsilon$. The algorithm initializes the PageRank value for all the vertices to $1/n$ and iteratively applies the following equation to each vertex:

$$PR[v] = \frac{1 - \gamma}{n} + \gamma \sum_{u \in \mathcal{N}^-(v)} \frac{PR[u]}{\deg^+(u)}$$

(3.1)

The iterative application of the PageRank formula continues until the sum of differences of PageRank values between iterations drops below $\varepsilon$.

To meet the criteria in Requirement 1, we:

1. Ignored the convergence criteria of PageRank and performed a constant number of iterations, and
2. Modified the algorithm to iterate over the edges in an arbitrary order (instead of the in-neighbourhood iteration seen in Equation (3.1)).

We adapted the PageRank implementation of McSherry [70] for our purposes (See Algorithm 3). This implementation takes as input an ordered edge list $(e_1, \ldots, e_m)$, which is traversed in each PageRank iteration. We computed the three edge orderings by:

1. **Row-major/Push**: sorting $E$ by ascending $(u,v)$
2. **Column-major/Pull**: sorting $E$ by ascending $(v,u)$
3. **Hilbert Order**: sorting $E$ by ascending $h(u,v)$

For each PageRank iteration, we:

1. Iterated over the vertices of the graph (lines 9-11 in Algorithm 3) and used the accumulated sum of PageRank values to update the PageRank of each vertex for the next iteration.
2. Iterated over the edges of the graph (lines 12-13 in Algorithm 3). For each edge, we accumulated the source vertex’s PageRank value into the destination vertex. This accumulation corresponded to the sum in the second term of Equation (3.1).
Algorithm 3 Single-threaded PageRank Benchmark

1: procedure PAGE\_RANK\((n, (e_1, \ldots, e_m), \gamma = 0.85, \text{num\_iters} = 20)\)

2: \(p_{\text{curr}} = [0.0, \ldots, 0.0] \quad \triangleright \text{initialize arrays of size } n (n = |V|) \text{ to store the PageRank of vertices}\)

3: \(p_{\text{next}} = [0.0, \ldots, 0.0] \quad \triangleright \text{use } p_{\text{next}} \text{ to accumulate } p_{\text{curr}} \text{ values from neighbours}\)

4: \(\text{inv\_degs} = [0.0, \ldots, 0.0] \quad \triangleright \text{populate inverse out-degrees}\)

5: for \(u \in V:\) do

6: \quad if \(n^+(u) > 0\) then \quad \triangleright \text{ignore vertices whose out-degree is zero (i.e. } n^+(u) = 0)\)

7: \quad \text{inv\_degs}[u] = \frac{1}{n^+(u)}

8: for \(i \leftarrow 1 \text{ to } \text{num\_iters}:\) do

9: \quad for \(u \leftarrow 1 \text{ to } n\) do

10: \quad \quad \(p_{\text{curr}}[u] = \gamma \times p_{\text{next}}[u] \times \text{inv\_degs}[u]\)

11: \quad \quad \(p_{\text{next}}[u] = \frac{1-\gamma}{n}\)

12: \quad for \(j \leftarrow 1 \text{ to } m\) do \quad \triangleright \text{iterate over ordered edges}\)

13: \quad \quad \(u, v = e_j\)

14: \quad \quad \(p_{\text{next}}[v] += p_{\text{curr}}[u]\)

return \(p_{\text{curr}}\)

For each edge \((u, v)\) in line 13 in Algorithm 3, we read from \(p_{\text{curr}}[u]\) and write to \(p_{\text{next}}[v]\). The locality of reads and writes is a function of the vertex-and-edge ordering. As discussed in Section 2.4.1, vertex ordering affects the spatial/temporal locality of neighbouring vertices and vertices whose IDs are adjacent. The edge ordering affects the read and write locality of consecutive edges. Since edges are sorted by ascending source ID, push will have excellent read locality and nearly random write locality. For the same reason, pull will have excellent write locality and nearly random read locality. The Hilbert curve will sacrifice the locality in one of the dimensions to improve the locality in the other dimension.

Since our PageRank implementation performs a constant amount of work \(O(n+m)\) in each iteration, and the number of iterations is constant, we meet all the criteria of Requirement 1 and can safely attribute any difference in performance to the different vertex-and-edge orderings.

3.2 Graph Dataset

To construct a representative dataset of real-world graphs, we selected the 33 largest directed graphs available from the Konect dataset [60]. We omitted the two largest directed graphs, since their single-threaded benchmark runtime exceeded 5 minutes. We list the graphs in Table 3.1. Our dataset consists of the following:

1. 2 social networks (twitter, soc-LiveJournal1),
2. 2 road networks (dimacs9-CTR/USA),
3. 1 contact network (sx-stackoverflow), and
4. 21 hyperlink networks. 20 hyperlink networks were either from different top-level domains of Wikipedia or from different aggregators of the site. The 21st hyperlink graph was an aggregated network of articles from the three Chinese online encyclopedias Baidu [2], Hudong [3] and Wikipedia in Chinese, as released by the Zhishi project [76].

While this dataset contains four categories of real-world graphs, there was an overrepresentation of hyperlink networks in the dataset. So, we augmented our dataset by including an additional 14 synthetic networks. We used the GAP Benchmark Suite [13] to generate seven Erdős-Rényi (Uniform Random) graphs [36] and seven Kronecker graphs [65]. We used the Kronecker graph generator with the same parameters as used by Graph 500 [74] ($A = 0.57, B = C = 0.19, D = 0.05$) to generate random graphs with a Power-law degree distribution. We generated each random graph (both Kronecker and uniform random) by using a constant number of nodes, $n = 2^{22}$, and an increasingly larger average degree $\bar{d} \in \{8, 12, 16, 20, 24, 28, 32\}$. We list the random networks in Tables 3.2, 3.3.
<table>
<thead>
<tr>
<th>Graph</th>
<th>Code</th>
<th>$n$</th>
<th>$m$</th>
<th>Network Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbpedia-link</td>
<td>db</td>
<td>18,268,991</td>
<td>136,537,566</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>dimacs9-CTR</td>
<td>ctr</td>
<td>14,081,816</td>
<td>33,866,826</td>
<td>Road</td>
</tr>
<tr>
<td>dimacs9-USA</td>
<td>usa</td>
<td>23,947,347</td>
<td>57,708,624</td>
<td>Road</td>
</tr>
<tr>
<td>link-dynamic-dewiki</td>
<td>ldd</td>
<td>2,166,661</td>
<td>43,245,190</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>lj</td>
<td>4,846,609</td>
<td>68,475,391</td>
<td>Social</td>
</tr>
<tr>
<td>sx-stackoverflow</td>
<td>sx</td>
<td>7,549,234</td>
<td>24,007,765</td>
<td>Contact</td>
</tr>
<tr>
<td>twitter</td>
<td>tw</td>
<td>41,652,230</td>
<td>1,468,364,884</td>
<td>Social</td>
</tr>
<tr>
<td>wikipedia-growth</td>
<td>gr</td>
<td>1,870,709</td>
<td>39,953,145</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_ar</td>
<td>ar</td>
<td>1,073,810</td>
<td>66,456,691</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_ca</td>
<td>ca</td>
<td>926,636</td>
<td>38,965,095</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_ceb</td>
<td>ceb</td>
<td>7,891,015</td>
<td>69,540,144</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_de</td>
<td>de</td>
<td>3,603,726</td>
<td>96,832,299</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_en</td>
<td>en</td>
<td>13,593,032</td>
<td>437,167,958</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_es</td>
<td>es</td>
<td>3,033,373</td>
<td>46,652,467</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_fa</td>
<td>fa</td>
<td>2,105,723</td>
<td>66,968,791</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_fr</td>
<td>fr</td>
<td>3,333,397</td>
<td>123,690,096</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_hu</td>
<td>hu</td>
<td>621,500</td>
<td>46,356,886</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_it</td>
<td>it</td>
<td>2,148,791</td>
<td>104,689,918</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_ja</td>
<td>ja</td>
<td>1,767,268</td>
<td>83,195,203</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_nl</td>
<td>nl</td>
<td>2,622,756</td>
<td>59,165,624</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_pl</td>
<td>pl</td>
<td>1,688,608</td>
<td>70,253,994</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_pt</td>
<td>pt</td>
<td>1,744,781</td>
<td>57,668,866</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_ru</td>
<td>ru</td>
<td>3,370,462</td>
<td>93,349,659</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_sh</td>
<td>sh</td>
<td>3,924,218</td>
<td>106,265,127</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_sr</td>
<td>sr</td>
<td>3,175,009</td>
<td>139,585,489</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_sv</td>
<td>sv</td>
<td>6,100,692</td>
<td>106,748,310</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_uk</td>
<td>uk</td>
<td>1,201,408</td>
<td>59,134,144</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_vi</td>
<td>vi</td>
<td>1,372,590</td>
<td>51,074,939</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_war</td>
<td>war</td>
<td>2,093,449</td>
<td>38,631,841</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>wikipedia_link_zh</td>
<td>zh</td>
<td>1,786,381</td>
<td>72,610,512</td>
<td>Hyperlink</td>
</tr>
<tr>
<td>zhishi-all</td>
<td>zhi</td>
<td>7,825,669</td>
<td>64,637,949</td>
<td>Hyperlink</td>
</tr>
</tbody>
</table>

**Table 3.1:** Real World Graphs evaluated in the case study.
<table>
<thead>
<tr>
<th>Graph</th>
<th>n</th>
<th>m</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = $2^2$, $d = 8$</td>
<td>$2^2$</td>
<td>67,108,730</td>
<td>Uniform-Random</td>
</tr>
<tr>
<td>n = $2^2$, $d = 12$</td>
<td>$2^2$</td>
<td>100,663,010</td>
<td>Uniform-Random</td>
</tr>
<tr>
<td>n = $2^2$, $d = 16$</td>
<td>$2^2$</td>
<td>134,217,182</td>
<td>Uniform-Random</td>
</tr>
<tr>
<td>n = $2^2$, $d = 20$</td>
<td>$2^2$</td>
<td>167,771,318</td>
<td>Uniform-Random</td>
</tr>
<tr>
<td>n = $2^2$, $d = 24$</td>
<td>$2^2$</td>
<td>201,325,342</td>
<td>Uniform-Random</td>
</tr>
<tr>
<td>n = $2^2$, $d = 28$</td>
<td>$2^2$</td>
<td>234,879,352</td>
<td>Uniform-Random</td>
</tr>
<tr>
<td>n = $2^2$, $d = 32$</td>
<td>$2^2$</td>
<td>268,433,288</td>
<td>Uniform-Random</td>
</tr>
</tbody>
</table>

Table 3.2: Uniform Random Graphs evaluated in the case study. The uniform random graphs are symmetric: for each edge $u, v$, the inverse edge $v, u$ exists and is explicitly represented in the graph. Thus, $m = 2dn$.

<table>
<thead>
<tr>
<th>Graph</th>
<th>n</th>
<th>m</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = $2^2$, $d = 8$</td>
<td>$2^2$</td>
<td>2,009,335</td>
<td>Kronecker</td>
</tr>
<tr>
<td>n = $2^2$, $d = 12$</td>
<td>$2^2$</td>
<td>2,236,281</td>
<td>Kronecker</td>
</tr>
<tr>
<td>n = $2^2$, $d = 16$</td>
<td>$2^2$</td>
<td>2,396,233</td>
<td>Kronecker</td>
</tr>
<tr>
<td>n = $2^2$, $d = 20$</td>
<td>$2^2$</td>
<td>2,518,144</td>
<td>Kronecker</td>
</tr>
<tr>
<td>n = $2^2$, $d = 24$</td>
<td>$2^2$</td>
<td>2,616,620</td>
<td>Kronecker</td>
</tr>
<tr>
<td>n = $2^2$, $d = 28$</td>
<td>$2^2$</td>
<td>2,698,283</td>
<td>Kronecker</td>
</tr>
<tr>
<td>n = $2^2$, $d = 32$</td>
<td>$2^2$</td>
<td>2,768,308</td>
<td>Kronecker</td>
</tr>
</tbody>
</table>

Table 3.3: Kronecker Random Graphs evaluated in the case study. Note that $n < 2^2$ for all Kronecker graphs. This is due to the removal of singleton vertices (vertices with no neighbours) after the random graph generation.

### 3.3 Stressing Cache Capacity

After selecting a graph analytic task and a dataset of graphs on which to run the task, we verified that our benchmark will stress our machine’s cache capacity (Requirement 3). We ran the benchmark on an Intel® Xeon® W-2275 CPU @ 3.30GHz with 128 GB of RAM. The sizes of the L1, L2, and LLC caches were 32 KB, 1 MB, and 19.25 MB, respectively.

To calculate the Resident Set Size (RSS) of Algorithm 1, we distinguished between the access patterns of the two loops performed for each iteration of PageRank. **Loop 1** (lines 9-11 in Algorithm 3) sequentially accesses $p_{\text{curr}}, p_{\text{next}}$, and $\text{inv\_degs}$. **Loop 2** (lines 12-13 in Algorithm 3), sequentially reads the ordered edges in $(e_1, \ldots, e_m)$ and reads and writes to the indices defined by the source and destination IDs. Note that there is no guarantee that consecutive source and destination IDs will be adjacent in memory. The goal of vertex-and-edge ordering is to improve the locality of read access to the $p_{\text{curr}}$ array and the locality of write access to the $p_{\text{next}}$ array. The better the improvement in the locality of
consecutive reads and writes, the faster the runtime. Due to the completely sequential memory access pattern of Loop 1, we omitted it from the Resident Set Size calculation and focused only on Loop 2. We used float for the datatype of the $p_{curr}, p_{next}$ arrays, and represented edges using a vector of pair<uint32_t, uint32_t>. Since both float and uint32_t are 4 byte quantities, we could calculate the Resident Set Size (in bytes) for the PageRank benchmark on a given graph using:

$$RSS_{PR}(G) = 4 \cdot 2 \cdot n + 4 \cdot 2 \cdot m$$  \hspace{1cm} (3.2)

Finally, we verified that the Resident Set Size of all the graphs in Tables 3.1 - 3.3 was greater than the size of the LLC. We also verified that running our benchmark on a graph whose Resident Set Size is smaller than our LLC incurred a minimal number of LLC misses.

### 3.4 Implementation

We implemented Algorithm 3 in C++ and used the Intel® icpx Compiler Version 2023.1.0. For each graph in Tables 3.1 - 3.3, we computed 18 different edge-lists (6 vertex orderings $\times$ 3 edge orderings) to produce a total of $45 \times 18 = 810$ experiments. For each experiment, we measured the runtime of 20 iterations of PageRank (Lines 8-13 in Algorithm 3). We repeated each experiment 5 times and report the mean and standard deviation of the runtime in our results. For each experiment, we computed an equivalent metric, Millions of Edges per Second (MEPS), using the following formula:

$$MEPS = \frac{m \times num\_iters}{\text{runtime in seconds}} \times 10^6$$  \hspace{1cm} (3.3)

### 3.5 Results

By analyzing the results of our case study, we answer the following questions.

1. Is there an edge ordering that consistently performs best?
2. Is there a vertex-and-edge ordering that consistently performs best?
Table 3.4: Case study results for the Konect graphs from Table 3.1. Each cell lists the average runtimes (across five trials) for 20 iterations of edge-centric PageRank for a given graph, vertex order, and edge order. Standard deviations are listed in parenthesis below each value. Bolded text corresponds to the best runtime for a graph and vertex ordering. Bolded, red text corresponds to the best runtime for a graph.
produced performance improvements, graphs. These results show that the improved read and write locality due to Hilbert Ordering consistently
for our benchmark. If we compare the runtimes across the different vertex-and-edge orderings for
3.5.1 Edge Ordering
the original findings of McSherry et al. [71].

Table 3.5: Case study results for the uniform random graphs from Table 3.2. Each cell lists the
average runtimes (across five trials) for 20 iterations of edge-centric PageRank for a given graph,
vertex order, and edge order. Standard deviations are listed in parenthesis below each value.
Bolded text corresponds to the best runtime for a graph and vertex ordering. Bolded, red
text corresponds to the best runtime for a graph.

Table 3.6: Case study results for the Kronecker graphs from Table 3.3. Each cell lists the average
runtimes (across five trials) for 20 iterations of edge-centric PageRank for a given graph,
vertex order, and edge order. Standard deviations are listed in parenthesis below each value.
Bolded text corresponds to the best runtime for a graph and vertex ordering. Bolded, red text
- corresponds to the best runtime for a graph.

3.5.1 Edge Ordering
For 244 out of 270 (90%) graph isomorphisms, the Hilbert edge ordering produced the fastest runtime
for our benchmark. If we compare the runtimes across the different vertex-and-edge orderings for
the graphs, we see that using the Hilbert curve produced the best runtimes for 42 out of 45 (93%)
graphs. These results show that the improved read and write locality due to Hilbert Ordering consistently
produced performance improvements, regardless of graph or vertex ordering, and are consistent with
the original findings of McSherry et al. [71].
3.5.2 Vertex-and-Edge Ordering

For our next set of analyses, we restricted our experimental results to the experiments that used the Hilbert ordering. This refined our question to: “What is the best vertex ordering that complements the improved read/write locality of the Hilbert curve?”

Figure 3.1 illustrates that for 30 out of 45 graphs whose edges were ordered using the Hilbert curve, the best performing vertex ordering was SlashBurn.

![Distribution of Best Performing Vertex Orderings for Edgelists Ordered Using the Hilbert Curve](image)

**Figure 3.1:** The best performing vertex ordering for 30/45 graphs whose edges were ordered using the Hilbert curve was SlashBurn. For seven out of those 30 graphs, the percent improvement due to SlashBurn over the next best vertex ordering was greater than 10%.

3.6 Discussion

To analyze the consistent performance improvement achieved by using the SlashBurn and Hilbert vertex-and-edge ordering on a large number of graphs, we added another metric to our dataset. This metric was inspired by the Neighbour-to-Neighbour Average ID Distance (AID) proposed by Esfahani et al. [37]. The AID quantifies the spatial locality of a vertex ordering by averaging the absolute distance between IDs of neighboring vertices.

We shift our focus from vertex orderings to edge orderings and define a similar metric called the Average Read/Write Distance (ARWD), which quantifies the spatial locality of edges given an edge ordering. For an ordered edge list \((e_1, \ldots, e_m)\), where the source and destination vertices of edge \(e_i\) are denoted as \(u_{e_i}\) and \(v_{e_i}\) respectively, we define the Average Read Distance (ARD) and Average Write Distance (AWD) as follows:

\[
ARD = \frac{\sum_{i=2}^{m} |u_{e_{i-1}} - u_{e_i}|}{m - 1} \quad (3.4)
\]
\[ AWD = \frac{\sum_{i=2}^{m} |v_{e_i} - v_{e_{i-1}}|}{m - 1} \]  

(3.5)

The ARWD (Average Read/Write Distance) is the product of ARD and AWD:

\[ ARWD = ARD \times AWD \]  

(3.6)

Expectedly, for Row-major edge orderings, the Average Read Distance is minimized while the Average Write Distance can be large, and vice versa for Column-major orderings. These averages are useful when considering the Hilbert curve or any other edge ordering that improves the locality of accesses based on source and/or destination IDs of consecutive edges. In the case of the Hilbert curve, its locality-preserving property (See Section 2.4.2) should try to minimize the Average Read/Write Distance. In our benchmark, a low Average Read/Write Distance directly translated to improved read and write locality of consecutive edges and, consequently, faster runtimes.

To illustrate the relationship between the performance improvement due to Hilbert Ordering and Vertex Ordering and the Average Read/Write Distance, we plotted the results of our case study in Figures 3.2 and 3.3.

**Figure 3.2:** Performance of edgelist traversal using the Hilbert curve on Uniform Random and Kronecker graphs as a function of vertex ordering and Average Read/Write Distance (ARWD). Graphs are distinguished by colours and vertex orders are distinguished by marker types. The shared y-axis shows performance measured in Millions of Edges traversed per Second (MEPS) and calculated using Equation (3.3). The logarithmic x-axis shows the Average Read/Write Distance (ARWD) calculated using Equation (3.6). Highlighted marks with a bold black outline indicate the fastest vertex order for that graph.
3.6.1 Uniform Random Graphs

The left plot in Figure 3.2 presents the performance on the Uniform Random graphs from Table 3.2. The Average Read/Write Distances (ARWD) of the Uniform Random graphs range from approximately 23,000 to 100,000. These distances are about one order of magnitude larger than the ARWD range observed for the Kronecker graphs. Generally, the ARWD of Uniform Random graphs exceeds that of Kronecker graphs, due to the uniform distribution of the edges of Uniform Random graphs, where each vertex has an equal probability of being the neighbor of any other vertex. As a result, no specific vertex ordering consistently yields additional performance improvement. As expected, increasing the density of the Uniform Random graphs corresponds to a decrease in the ARWD.

3.6.2 Kronecker Graphs

The right plot in Figure 3.2 displays the performance on the Kronecker graphs from Table 3.3. Despite using the same seed number of nodes and average degree as the Uniform Random graphs, the Kronecker graphs have a significantly smaller number of nodes due to the removal of the large number of singleton vertices that are characteristic of Kronecker graphs [6]. However, the number of edges remains approximately the same.

Figure 3.2 shows that we get the best performance (greatest MEPS) when the ARWD is smallest. That is, performance (measured in MEPS) is inversely related to ARWD. Also, a noticeable pattern emerges in terms of decreasing ARWD with respect to vertex ordering, as follows:

Original > Rabbit > Hub-Cluster > Hub-Sort > Desc. Degree Sort > SlashBurn

This pattern is due to the power-law degree distribution of the Kronecker graphs and the clustering of hub vertices. SlashBurn achieves an even lower ARWD than the degree-based vertex orderings, because it groups the hub vertices and separates them from the “spokes” and singletons, concentrating all the edges of the graph to the dense “wings”. Except for one case (a Kronecker graph with \( n = 2^{22} \) and \( \overline{d} = 28 \)), SlashBurn consistently produces the lowest ARWD for the Hilbert edge ordering, resulting in the overall fastest runtime for Kronecker graphs.

3.6.3 Real-World Graphs

Figure 3.3 illustrates the interaction between ARWD and performance improvement due to vertex-and-edge ordering for the graphs in Table 3.3.
Figure 3.3: Performance of edgelist traversal using the Hilbert curve on the real-world graphs from Table 3.1 as a function of vertex ordering and Average Read/Write Distance (ARWD). The title of each plot lists the range of the number of edges in the graphs of the plot. Graphs are distinguished by colours within a plot. Vertex orders are distinguished by marker types. The shared y-axes show performance measured in Millions of Edges traversed per Second (MEPS) and calculated using Equation (3.3). The shared logarithmic x-axes show the Average Read/Write Distance (ARWD) calculated using Equation (3.6). Highlighted marks with a bold black outline indicate the fastest vertex order for that graph.

The results are not as clear-cut as with the Uniform and Kronecker Random graphs. However, the pattern remains consistent: **Performance is inversely related to ARWD. SlashBurn minimizes the ARWD for a given graph, leading to the fastest single-threaded edge traversals using the Hilbert curve.** The SlashBurn and Hilbert vertex-and-edge ordering achieve the fastest runtime for 25 out of 31 real-world graphs. Out of those 25 graphs, SlashBurn and Hilbert also produce the lowest ARWD for
20 of them. This observation is evident by examining each cluster of points corresponding to the same graph individually (marked with the same color) and identifying the top-leftmost point, representing the lowest ARWD and highest MEPS (Million Edges Processed per Second). There are two notable exceptions to this observation:

1. The two road networks, dimac9-CTR and USA: The lowest ARWD for these graphs was not achieved using SlashBurn but rather with the Original and Rabbit ordering. This can be attributed to the non-power-law degree distribution of these networks. Unlike scale-free networks, these road networks have relatively low minimum, maximum, and average degrees, making it challenging for SlashBurn to create an efficient compressed representation of the graph’s edges. The original vertex ordering, which is based on geographical location, assigns IDs that result in a relatively low ARWD, by default. In the case of the complete USA road network, Rabbit-order improves upon the original vertex ID assignment by detecting communities within the network, leading to slightly better performance and a lower ARWD.

2. The hyperlink networks, wikipedia link ar, pt, uk, and zh: For these graphs, the best performing vertex order is not SlashBurn, but Descending Degree Sort (marked with a star). Although SlashBurn achieves a slightly lower ARWD, Descending Degree Sort yields a slightly better runtime. In these graphs, 24-39% of the vertices have no in-neighbours. Descending degree sort clusters these vertices to produce a wide empty vertical “stripe” of the adjacency matrix. This reduces the region of write access for the PageRank kernel (when compared to the SlashBurn ordering), and results in the faster runtimes for Descending Degree Sort.

3.7 Conclusion

It is not surprising that the SlashBurn vertex ordering and Hilbert edge ordering work so well together and produce the fastest running times for various scale-free graphs such as social, contact, and hyperlink networks. The objective of SlashBurn is to create a compressed graph representation by clustering the edges into fewer denser blocks, as opposed to a larger number of sparser blocks [66]. The Hilbert edge ordering sacrifices locality in either the source or destination vertex dimensions, while drastically improving the locality of the other dimension. This improvement in spatial locality is quantified in our case study using the Average Read/Write Distance metrics. Complementarily, SlashBurn produces a compressed clustering of edges, whose spatial locality can be further leveraged by the Hilbert curve.

However, it is important to remember that this performance improvement is not the whole story. We have not considered the preprocessing costs of either SlashBurn or Hilbert ordering in our analysis. This detail is particularly relevant since SlashBurn is the most computationally expensive vertex ordering technique we examined, with a complexity of $O(m + n \log n)i$, where $i$ represents the number of iterations required to complete the algorithm. Recognizing the improved performance due to these two techniques, we decided to improve their practicality. In the following chapter, we describe our parallelization of the SlashBurn vertex reordering algorithm, and in Chapter 5, we extend our efforts to develop a scalable, multicore edge reordering and blocking technique that we designed for the Hilbert curve.
Chapter 4

Parallel SlashBurn

Based on the results from our case study in Chapter 3, we found that:

1. The SlashBurn vertex ordering minimizes the Average Read-Write Distance (ARWD) for scale-free networks, and, as a result,

2. Using the Slashburn+Hilbert vertex-and-edge ordering consistently produces the fastest running times to compute the PageRank of power-law graphs.

These results position SlashBurn as a strong candidate to include in preprocessing for graph algorithms that repeatedly iterate over all the edges of the graph. However, we did not include the preprocessing cost of computing the SlashBurn ordering in our evaluation. For example, our single-threaded SlashBurn implementation in C++ using the igraph [30] library took 4.55 minutes to complete on the soc-LiveJournal1 graph. If we compare the time to compute the PageRank of the graph using the Original+Hilbert vertex-and-edge ordering (7.72s) vs. the SlashBurn+Hilbert vertex-and-edge ordering (6.11s), we see a 1.61s improvement that is due to the SlashBurn reordering 1. The preprocessing time of 4.55 minutes dwarfs the 1.61s improvement in runtime achieved by SlashBurn, rendering it unsuitable as a lightweight vertex reordering technique. To recover the preprocessing cost, we would need to repeat the task 170 times, which is unrealistic, especially for one-off graph analytic tasks such as PageRank.

It is worth noting that many main-memory graph processing systems (e.g., GPOP [61], Cagra [97], and GraphMat [89]) justify the cost of preprocessing by treating it as a one-time step. While there are scenarios where preprocessing costs may be acceptable, we focus on improving the practicality of SlashBurn for one-off graph analytic tasks. Further, while we treat PageRank as a one-off task when computing the end-to-end speedup due to graph ordering, we acknowledge that algorithms such as Incremental PageRank [9] address this concern, and leverage previously computed properties to answer dynamic queries. We consider the use of dynamic graph algorithms on ordered graphs and the efficient

---

1The runtimes for the PageRank computation are larger than the runtimes listed in Table 3.4. Table 3.4 lists the time to complete 20 iterations whereas the values listed correspond to the time required to converge for the soc-LiveJournal1 graph. Specifically, this requires the following tolerance to be met: \( \sum_{v \in V} |PR_{i+1}(v) - PR_i(v)| < 10^{-6} \). This took 68 iterations for soc-LiveJournal1.
update of vertex ordering in the context of vertex and edge insertions/deletions out of scope for this
thesis, but acknowledge these questions as avenues for further exploration in future work. Nonethe-
less, even in instances where preprocessing is a one-time cost that is amortized over future executions,
reducing that cost is still valuable.

To address the prohibitive cost of SlashBurn, we decided to parallelize the algorithm. Specifically,
we parallelize key subroutines, namely: degree/spoke sorting and connected components computation.
In Section 4.1, we review the SlashBurn algorithm, highlighting the subroutines that can be parallelized.
Section 4.2 presents **Parallel SlashBurn** and our optimizations:

1. Parallelizing degree sort using a state-of-the-art, parallel samplesort [8].

2. Parallelizing connected components using Afforest [90].

3. Parallelizing degree-decrement and spoke-sizing operations using OpenMP [79] and Sparse Array
   Reductions [54].

Finally, in Section 4.3, we evaluate the performance of our parallel implementation by measuring
the improvement that is gained by our optimizations.

4.1 SlashBurn Analysis

Algorithm 4 uses pseudocode to describe the SlashBurn algorithm from the original paper [66]. Al-
gorithm 5 describes the SlashBurn algorithm in more detail. The input to SlashBurn is an undirected
directed graph \( G(V, E) \). The goal of the vertex ordering is to output a permutation \( \pi : V \rightarrow [n] \) such that the
edges of \( G \) are grouped together in the adjacency matrix of \( G \). To order the vertices of a directed graph
using SlashBurn, we ignore the directionality of edges. Lim et al. [66] describe two optimizations to
their algorithm: **Greedy k-hubset selection** and **Hub-ordering of spokes**, but we chose to parallelize the
simplest version of SlashBurn and used basic \( k \)-hubset selection and order spokes by their size.

**Algorithm 4** SlashBurn Algorithm Pseudocode [66].

```
Input Edge set \( E \) of a graph \( G = (V, E) \), a constant integer \( k \geq 1 \) (default = 0.005\( n \)).
Output Array \( \pi \) containing the ordering \( V \rightarrow [n] \)

1: Remove the \( k \)-hubset from \( G \) to make the new graph \( G' \). Add the removed \( k \)-hubset to the front of \( \pi \).
2: Find connected components in \( G' \). Add nodes in non-giant connected components to the back of \( \pi \),
in the decreasing order of sizes of connected components they belong to.
3: Set \( G \) to be the giant connected component (GCC) of \( G' \). Go to step 1. Continue until the number of
   nodes in the GCC is smaller than \( k \).
```
Algorithm 5 SlashBurn Algorithm [66].

**Input** Edge set $E$ of a graph $G = (V,E)$, a constant integer $k \geq 1$ (default = 0.005$n$).

**Output** Array $\pi$ containing the ordering $V \rightarrow [n]$

1: Initialize an empty array $\pi$ of size $n$
2: $hub\_start \leftarrow 0$
3: $spoke\_start \leftarrow n$
4: while $G$ is non-empty do
5: Sort vertices by descending degree $\triangleright$ Compute $k$-hubset
6: Remove $k$ highest degree vertices and their incident edges from $G$ to create $G'$
7: Insert $k$ highest degree vertices into $\pi$ in the range $[hub\_start, hub\_start + k)$
8: $hub\_start + = k$
9: Find connected components of $G'$
10: Let GCC be the largest connected component; call all other connected components spokes
11: Sort spokes by decreasing size
12: $l \leftarrow \text{len}(spokes)$
13: $m \leftarrow$ total number of vertices in spokes
14: $spoke\_start -= m$
15: $j \leftarrow spoke\_start$
16: for $i$ in $0, \ldots, l - 1$ do
17: $spoke \leftarrow spokes[i]$
18: for $v \in spoke$ do
19: $\pi[j] = v$
20: $j + +$
21: $G \leftarrow GCC$
22: If the number of vertices in GCC is smaller than $k$, place remaining vertices of GCC into $\pi$ in the range $[hub\_start, spoke\_start)$ and return

First, Algorithm 4 identifies the $k$-hubset of $G$: the $k$ highest degree vertices in $G$. We use a descending degree sort to identify the $k$-hubset. For each vertex $v$ that is removed from the graph and placed into the front of $\pi$, we must iterate over $v$’s neighbourhood to decrement the degrees of each of $v$’s neighbours. This ensures that the correct $k$-hubset is selected in the next iteration. Let $G'$ be the subgraph resulting from the removal of the $k$-hubset and its incident edges from $G$. Due to the descending degree sort, the complexity of Step 1 is $O(n \lg n)$.

Next, we find the connected components in $G'$. This can be done by using either a BFS or DFS. We identify the GCC and spokes of $G'$. The spokes of $G'$ are the connected components of $G'$ whose size is smaller than that of the GCC, or the “non-giant” connected components. We sort the spokes of $G'$ by the number of vertices in each spoke. We place the spokes’ vertices into the end of the permutation array ($\pi$) in reverse order, ordered from smallest to largest spoke. Intuitively, we use this ordering to place the spokes’ vertices in $\pi$ because we wish to maximize the “separation” between the lowest degree vertices.
of $G$ (i.e., singleton spokes) from the highest degree vertices $G$ (i.e., the $k$ hubs). We must decrement the degree of all neighbours of $v : v \in G'$. Due to the step that computes the connected components, the complexity of Step 2 is $O(m + n)$.

Finally, we compare the size of the GCC to $k$. If $|GCC| \geq k$, we set $G = GCC$, and recursively repeat the algorithm on $G$. Otherwise, we place the vertices of the GCC in $\pi$. Given the complexities of Steps 1 and 2, the overall complexity of SlashBurn is $O(m + n \lg n)i$, where $i$ is the number of iterations to complete the algorithm.

4.2 Parallel SlashBurn

We parallelize SlashBurn by assembling the most suitable parallel algorithms for each of 1) the degree sort, 2) the connected component identification, and 3) the degree decrement and spoke-sizing operations.

4.2.1 Parallelizing Degree Sort using In-Place Parallel Super Scalar Samplesort ($IPS^4o$)

Since we must sort the hubs of $G$ and spokes of $G'$ at every iteration, we stand to gain substantial speedups by using a performant parallel sort algorithm. By default, the current C++ standard library of the GNU Compiler Collection (GCC) uses a variant of Hoare’s Quicksort [53, 75]. In Quicksort, we select a pivot element that partitions the array such that all elements smaller than the pivot are in the left part of the array and all elements larger than the pivot are in the right part of the array. We solve the left and right subproblems recursively. Samplesort [19, 20] improves on Quicksort by increasing the number of pivots and partitioning the input array into buckets. This makes Samplesort more cache-efficient and better suited for parallelization [8]. Super-scalar samplesort ($s^3$-sort) [93] further improves on Samplesort by avoiding branch mispredictions, where a processor incorrectly predicts the result of element comparisons [8]. Finally, In-Place Parallel Super Scalar Samplesort ($IPS^4o$) is a state-of-the-art improvement of $s^3$-sort that parallelizes the distribution of elements to buckets and modifies the algorithm to be in-place (i.e., using only constant space in addition to the input). Importantly, $IPS^4o$ takes advantage of inputs that contain many identical keys by using separate buckets for elements that are equal to pivots. This is particularly relevant to degree-sorting of power-law graphs, since such graphs contain many vertices with identical small degrees. We parallelize all sorting (degree, spoke) operations of Parallel SlashBurn by using $IPS^4o$.

4.2.2 Parallelizing Connected Components using Afforest

The Connected Components (cc) algorithm is widely used as both a standalone graph analytic task in scientific fields such as biology [55] and image processing [34] and as a preprocessing step for downstream analytics such as graph clustering [39] and partitioning [81]. There is extensive prior work that accelerates cc [41, 83, 88]. We focus on a specific class of cc algorithms that are amenable to parallelization: tree-hooking algorithms. The original tree-hooking parallel algorithm by Shiloach and
Vishkin [83] transforms the graph into trees and iteratively connects the trees in the hooking phase and reduces the trees’ depths in the shortcut phase. Specifically, we begin by assigning all vertices a unique component ID based on their vertex ID. In the hooking phase, we iterate over the edges of the graph. For every edge, we look at the Component IDs of the source and destination vertices. If their component IDs are different, we take the minimum of both IDs, and assign the minimum ID to the vertex with the larger component ID. After the hooking phase, the component ID of some vertices may not point to their root vertex (the vertex with the smallest ID in that component). So, in the shortcut phase, we iterate over the vertices, and repeatedly update the component IDs of vertices until all component IDs point to their corresponding root vertex. The Shiloach-Vishkin (SV) algorithm converges to a valid component ID assignment once the graph is converted to a forest of depth-one trees. The SV algorithm uses parallelism in both phases:

- SV iterates over the edges of the graph in parallel in the hooking phase, and
- SV iterates over the vertices of the graph in parallel in the shortcut phase.

Sutton et al. [90] further optimized SV by using Subgraph Sampling and Large Component Skipping. The Afforest algorithm first performs the hooking phase on a sample of the neighbourhoods of the vertices. The sampling produces an estimate of the largest intermediate component. Next, the hooking phase is performed on the neighbourhoods of vertices that do not belong to the largest intermediate component. By skipping the edges of the largest intermediate component, many edges can be omitted from the hooking computation.

We decided not to explicitly remove vertices from the graphs, since this would require us to rebuild the graph’s CSR in every iteration of SlashBurn. Instead, we indicate the deletion of a vertex by using a bitmap of size $n$ (similar to a tombstone [82]). Specifically, bit $i$ is set if vertex $i$ has been removed from the graph. Otherwise, vertex $i$ is an active vertex. We set a vertex’s bit (i.e., remove it from the graph) atomically using a compare-and-swap operation. This means that we need to modify Afforest slightly so that it can be used for the CC subroutine at each iteration of SlashBurn. We use the bitmap to skip over deleted vertices when traversing the graph to compute the Connected Components. We also use the bitmap to construct a set of active vertices to sample from during Subgraph Sampling.

### 4.2.3 Degree Decrementing and Spoke Sizing Using Spray and MapReductions

The final optimizations we implemented to parallelize SlashBurn were for the degree-decrement and spoke-sizing operations. We perform degree-decrement twice in every iteration of SlashBurn for the removal of Hubs and Spokes, since we must update the degrees of the neighbours of every vertex we remove from the graph. We use spoke-sizing to compute the sizes of all the spokes in the graph after running our modified Afforest. Algorithms 6 and 7 show the single-threaded degree-decrement and spoke-sizing operations, respectively.

Both Degree-Decrement and Spoke-Sizing can be viewed as reductions. A reduction operation uses associative and commutative operators such as summation and multiplication to “reduce” the dimensionality of the input [54]. For example, we can reduce an integer array using the + operator and an
Algorithm 6 Sequential Degree-Decrement

Input $vs$: an array of vertex IDs to remove from the graph
$deg: V \rightarrow [n]$, an array of vertex degrees
$b: V \rightarrow [n]$, a bitmap; $b[v] = 1$ if $v$ has been removed from the graph, 0 otherwise.

1: $n_{vs} \leftarrow \text{length}(vs)$
2: for $i \leftarrow 1$ to $n_{vs}$ do
3: $u \leftarrow vs[i]$
4: for $v \in N(u)$: do
5: if $b[v]$ then
6: continue \triangleright v has been removed from the graph
7: $deg[v] --$

Algorithm 7 Sequential Spoke-Sizing

Input $comp: V \rightarrow [n]$, an array of component ID assignments;
$comp[v]$ is the component ID that vertex $v$ belongs to.
$b: V \rightarrow [n]$, a bitmap; $b[v] = 1$ if $v$ has been removed from the graph, 0 otherwise.
$gcc$: the ID of the Giant Connected Component

Output $ccsize$, a map from component ID to component size

1: $ccsize = \{}$
2: for $i \leftarrow 1$ to $n$ do
3: if $b[i]$ then continue
4: $cid \leftarrow comp[i]$
5: if $cid == gcc$ then continue
6: if $ccsize.contains(cid)$ then $ccsize[cid] ++$
7: else $ccsize[cid] = 1$
8: return $ccsize$

initial value of 0 to compute the sum of its elements.

The intermediate results of reductions inherit the associative and commutative nature of the individual computations. This enables parallelization by splitting the problem into non-overlapping subproblems that are assigned to processors [54]. Each processor performs a local reduction, and the global result is obtained by combining the local results.

Parallel degree-decrement using Spray

Degree-decrement can be computed using a special type of reduction known as an array reduction. Unlike conventional reductions that reduce an array to a scalar variable, an array reduction produces an array as the output. We show the out-degree reduction algorithm in Algorithm 8 and a C++ implementation in Figure 4.1. We compute the out-degrees of a graph by iterating over the edges and using an array reduction. In each iteration, the $out\_degree$ array is updated in the location of the edge’s source vertex by using the associative and commutative $+= \text{ operator}.$
Algorithm 8 Out-degree reduction

Input edges, an ordered list of edges
Output out_degrees, an array of out-degrees; out_degrees[i] stores the out-degree of vertex i

1: out_degrees = [0, ..., 0]  
   ▷ initialize all out-degrees to zero
2: for e in edges do
3:    out_degrees[e.source] ++ 1

Figure 4.1: Sequential out-degree computation using an array reduction.

Figure 4.2 illustrates how we can use OpenMP [79] to parallelize this array reduction. #pragma omp parallel for indicates that the for-loop iterating over the edges should be parallelized. OpenMP parallelizes the for-loop iteration by dividing the iteration space into disjoint chunks that are approximately equal in size and assigning a chunk to each thread. Since the edgelist (edges) can be ordered arbitrarily, special care must be taken when modifying the shared out_degree to avoid race conditions (i.e., multiple threads incrementing the same source vertex’s out-degree at the same time). #pragma omp atomic addresses this by requiring that updates to out_degree be performed atomically.

#pragma omp parallel for
for (int i = 0; i < m; i++) {
    Edge e = edges[i];
    out_degree[e.source] += 1;
}

Figure 4.2: Concurrent out-degree computation using atomic updates.

Since array reductions are so common in high-performance computing [54], the OpenMP standard introduced an array reduction clause in Version 4.5 [78]. Figure 4.3 illustrates how we use an OpenMP array reduction to compute the out-degrees.
```c
#pragma omp parallel for reduction(+:out_degree[0:N])
for (int i = 0; i < m; i++) {
    Edge e = edges[i];
    int u = e.source;
    out_degree[u] += 1;
}
```

**Figure 4.3:** Concurrent out-degree computation using the OpenMP reduction clause.

The `reduction(+:out_degree[0:N])` directive instructs the compiler to perform an array reduction on the `out_degree` array, use the associative and commutative `+` operator, and only write final results to array indices in the range `[0, N)` where `N` is the size of the array.

In OpenMP, array reductions are implemented as follows:

1. Each thread initializes a private copy of the array to which the reduction clause is applied.
2. During parallel iteration, each thread updates its private array instead of the original, shared array to avoid race conditions when concurrently accumulating partial results.
3. At the end of parallel iteration, all private arrays are combined in an implementation-defined order [79], and the reduced values are stored in the original array.

Sparse array reductions, in which not all threads write to every array index, might exhibit poor scalability due to increased memory overhead, as memory grows linearly with the number of threads and the size of the original array, eventually leading to cache thrashing and performance slowdown. For sparse arrays, we could use atomic operations instead of using OpenMP array reductions. But, atomic operations might degrade performance if the sparse array reduction updates specific indices frequently. For example, updating the out-degrees of hub vertices might bottleneck the parallel reduction performance as multiple threads contend to atomically update the out-degrees of the same hubs.

Hückelheim and Doerfert [54] recognized that the two extremes of either: using atomic instructions to update the shared sparse array or maintaining and combining thread-private array copies do not suffice for all applications. Their solution was *Spray*: Sparse Reduction of Arrays in OpenMP. Spray is a header-only library of user-defined sparse array reductions. Spray enables developers to easily switch between sparse reduction strategies depending on their application. Importantly for us, Spray includes a *BlockReduction*. A sparse array *BlockReduction* lazily divides the input array into fixed-size blocks. Whenever a thread tries to update a location in the original array that belongs to a block that the thread has not seen before, the thread initializes a private block to accumulate intermediate values. Once all threads complete their assigned iterations, the reduction is finalized by merging thread-private blocks element by element.

BlockReductions are especially useful in the context of parallel graph processing. All threads likely use private blocks for the locations corresponding to the hub vertices, and blocks containing low-degree vertices will be initialized and used only by threads that access their locations. This eliminates the
expensive memory overhead due to OpenMP array reductions and thread contention due to access to hub vertices’ locations. Figure 4.4 illustrates the differences between OpenMP array reductions and Spray BlockReductions and Figure 4.5 shows the use of BlockReduction to compute the out-degrees of a graph. Algorithm 9 shows how we use BlockReduction to parallelize degree-decrement.

Figure 4.4: A comparison of OpenMP array reduction and Spray BlockReduction. Each box represents a memory location. The boxes for Threads 1 and 2 are private. The locations updated by Threads 1 and 2 are coloured using blue and green, respectively. The Spray BlockReduction avoids the expensive initialization of thread-private arrays. Using BlockReduction, each thread initializes only the memory it requires. Figure adapted from [54].
#include "spray.hpp"

// The BlockReduction object wraps the reduction
// locations out_degree[0:N].
spray::BlockReduction512<uint32_t> out_degree_reduction(out_degree, N);

#pragma omp parallel for
   reduction(+:out_degree_reduction[0:N])
for (int i = 0; i < m; i++) {
   Edge e = edges[i];
   int u = e.source;
   out_degree_reduction[u] += 1;
}

Figure 4.5: Concurrent out-degree computation using a BlockReduction with block size = 512.

Algorithm 9 Parallel Degree-Decrement

Input vs: an array of vertex IDs to remove from the graph
deg : V → [n], an array of vertex degrees
b : V → [n], a bitmap; b[v] = 1 if v has been removed from the graph, 0 otherwise.

1: n_vs ← length(vs)
2: degree_decrement = [0,...,0]  ▶ initialize a decrement array of size n
3: reduction = BlockReduction(degree_decrement, n) ▶ declare a spray block reduction
4: parallel_for i ← 1 to n_vs do
5:   u ← vs[i]
6:   for v ∈ N(u): do
7:      if b[v] then ▶ v has been removed from the graph
8:         continue
9:    reduction[v] +=
10: end parallel_for
11: parallel_for i ← 1 to n_vs do
12:   u ← vs[i]
13:   deg[u] -= degree_decrement[u]
14: end parallel_for

Parallel spoke-sizing using MapReductions

Parallelizing the spoke-sizing operation is simpler than degree-decrement. We adapt the MapReduce programming model [32]. Each thread processes a disjoint region of the component labels array to
compute a reduction of spoke sizes. The intermediate results are stored in thread-private maps mapping spoke IDs to local spoke size. We show the parallel spoke-sizing algorithm in Algorithm 10. In the algorithm, we use the `get_thread_id()` function to get the unique thread ID of the executing thread. We also assume that given $t$ threads, each thread is exclusively assigned an ID between 0 to $t - 1$.

**Algorithm 10 Parallel Spoke-Sizing**

*Input* $\text{comp} : V \rightarrow [n]$, an array of component ID assignments; $\text{comp}[v]$ is the component ID that vertex $v$ belongs to. $b : V \rightarrow [n]$, a bitmap; $b[v] = 1$ if $v$ has been removed from the graph, 0 otherwise. gcc: the ID of the Giant Connected Component

*Output* $\text{ccsize}$, a map from component ID to component size

1: $\text{priv}_\text{ccsize} = \{\{} \ldots \{\}$ \hspace{1em} $\triangleright$ an array of $t$ thread-private maps, where $t$ is the number of threads
2: tid ← `get_thread_id()`
3: parallel for $i ← 1$ to $n$ do \hspace{1em} $\triangleright$ each thread processes a disjoint region of the array
4: if $b[i]$ then continue
5: cid ← $\text{comp}[i]$  
6: if $cid == \text{gcc}$ then continue
7: if $\text{priv}_\text{ccsize}[tid].\text{contains}(cid)$ then $\text{priv}_\text{ccsize}[tid][cid] + +$
8: else $\text{priv}_\text{ccsize}[tid][cid] = 1$
9: end parallel_for
10:
11: $\text{ccsize} ← \{\}$
12: for $i ← 1$ to $t$ do \hspace{1em} $\triangleright$ combine the thread-private maps
13: for cid, component_size in $\text{priv}_\text{ccsize}[i]$ do
14: if $\text{ccsize}.\text{contains}(cid)$ then $\text{ccsize}[cid] += \text{component_size}$
15: else $\text{ccsize}[cid] = \text{component_size}$
16: return $\text{ccsize}$

To summarize, our optimizations parallelize the most critical sequential subroutines of SlashBurn. We use:

1. IPS to sort the hubs by degrees and spokes by size,

2. Afforest to compute the connected components of the graph, ignoring inactive vertices, and sampling only from active vertices,

3. BlockReductions for degree-decrement, and


Since all parallel subroutines (i.e., sorting, connected components, degree-decrement, and spoke-sizing) output the same result as their sequential counterparts, the result of Parallel SlashBurn is equivalent to the sequential implementation. In our evaluation below, we tested the correctness of Parallel SlashBurn by verifying that the output of Parallel SlashBurn is equal to the output of the original Sequential implementation.

43
4.3 Evaluation

Our evaluation answers the following two questions:

1. How much benefit is gained by using our optimizations (Parallel sorting, connected components, degree-decrement, and spoke-sizing)?

2. How does Parallel SlashBurn scale with an increasing number of cores?

We implemented Parallel SlashBurn in C++. We used IPS4o [8] and Spray [54] as submodules and adapted the Afforest implementation from the GAP Benchmark Suite [13]. We used flat_hash_map and flat_hash_set from the Abseil container library [45] to implement MapReduction. We used the Ubuntu Clang++ Compiler, Version 17.0.0. Parallel SlashBurn is publicly available at https://github.com/ubc-systopia/parsb. We evaluated Parallel SlashBurn on an Intel® Xeon® W-2275 CPU @ 3.30GHz with 14 cores (hyperthreading disabled) and 128 GB of RAM. We computed the SlashBurn ordering for a representative subset of graphs from our case study in Chapter 3:

- sx-stackoverflow - a contact network,
- soc-LiveJournal1 - a social network,
- dimacs9-USA - a road network, and
- wikipedia_link_en - a hyperlink network.

To answer the first question, we compared the runtimes of our sequential and parallel implementations of SlashBurn. In the sequential implementation, we used std::sort to sort hubs and spokes and a BFS algorithm to compute connected components CC. We compared the runtime of the sequential and optimized implementations by limiting the number of cores to 1. Table 4.1 summarizes the results of this comparison. The optimizations achieve notable speedups, ranging from 4.33x (dimacs9-USA) - 11.96x (wikipedia_link_en). Even with a single core, the optimized IPS4o and Afforest algorithms outperformed the alternative sequential implementation that used BFS and Quicksort.
Table 4.1: Comparison of execution times for the various implementations of SlashBurn. Run-times are in seconds. 1-Core is a sequential implementation of SlashBurn that uses QuickSort and BFS. The Optimized parallel implementation of SlashBurn uses IPS\textsuperscript{o}, Afforest, BlockReduction and MapReduction. Optimized Speedup is the ratio between 1-Core and 1-Core Optimized execution time. Parallel Speedup is the ratio between 1-Core Optimized to the 14-Core Optimized execution time. Total Speedup is the combined speedup gained by using the Parallel SlashBurn optimizations and the 14 available cores.

To answer the second question, we evaluated the scalability of Parallel SlashBurn by varying the number of cores used (1, 2, 4, 8, or all 14 cores). We present the results of this scalability study in Table 4.1 and Figure 4.6.
Figure 4.6: Scaling of Parallel SlashBurn with an increasing number of cores.

With all 14 available cores, Parallel SlashBurn exhibited scaling factors ranging from 6.78x
(sx-stackoverflow) to 8.19x (soc-LiveJournal1). Our optimized implementation using all available cores significantly reduced the preprocessing cost of the SlashBurn reordering. As a result, Parallel SlashBurn is a more lightweight vertex reordering that justifies the benefits it provides. Previously, the cost of SlashBurn was prohibitively high, requiring 170 repetitions to amortize the cost on the soc-LiveJournal1 graph, considering a 20.1% improvement due to SlashBurn. However, with our optimized implementation, the required number of iterations for amortization has dramatically decreased from 170 to 3, which makes SlashBurn a much more viable preprocessing technique.

Overall, our evaluation demonstrates the significant benefits and improved scalability achieved through our optimizations, making SlashBurn a more efficient and justifiable preprocessing step for graph analysis.
4.3.1 Limitations

We performed our scalability evaluation of SlashBurn on a machine with 14 cores. Unfortunately, we did not reach the multicore performance saturation point, as seen by the parallel speedup slopes in the plots in Figure 4.6. In the future, we intend to test the scalability of Parallel SlashBurn using a larger number of cores.

To more closely inspect the performance of each subroutine in our implementation, we plotted the execution time and number of active vertices per iteration in Figure 4.7. While Parallel SlashBurn has significantly improved performance, we recognize two limitations in our approach that could be...
addressed in future work to further enhance the speed of Parallel SlashBurn.

The first limitation is an increase in runtime per iteration towards the later stages of the algorithm. Ideally, as more vertices become inactive, we would expect the runtime per iteration to decrease. However, this is not the case due to the increasing time required to sort the spokes of the graph. As the graph becomes more fragmented, the number of small spokes grows, and the time we take to sort and distribute these spokes increases, indicated by the widening brown regions in Figure 4.7. Since we process the component label array in parallel, vertices belonging to the same spoke may be assigned to different threads, resulting in a scenario where the vertices of small spokes are distributed among multiple (if not all) threads. To address this, Parallel SlashBurn assigns each thread non-overlapping subregions to write to in the final permutation array. Nevertheless, the substantial number of vertices potentially split across multiple threads may introduce additional work in later iterations, leading to the increased runtime.

The second limitation relates to the degree-sorting step, which involves sorting the entire vertex array in each iteration. Particularly in later iterations where most vertices are inactive with a degree of 0, this can lead to performance degradation. Figure 4.7 highlights this with a nearly constant width associated with the orange Degree Sort regions. Despite the decreasing number of active vertices, we perform the same expensive (albeit optimized) sort operation in every iteration. An alternative approach could use a priority queue data structure to maintain vertices in descending order of their degrees. This data structure would need to either support the DecreaseKey operation (preferably by concurrent threads) or allow concurrent modification of the data structure using concurrent delete/add operations that could be achieved by using a Concurrent Skip List [51]. Notably, previous graph processing work has employed different execution modes based on the algorithm’s stage. For instance, Beamer et al. [12] proposed a Direction-Optimizing BFS with push traversals for sparse iterations (when a small number of vertices are active) and pull traversals for dense iterations. A similar approach could be useful for Parallel SlashBurn by maintaining a priority queue and using IPS to sort the vertices during dense iterations, while utilizing the priority queue during sparse iterations. Addressing these limitations in future research could further optimize SlashBurn, allowing for more efficient graph analysis.
Chapter 5

RHuBarb: Speeding up Edge-Centric Processing using Recursive Hilbert Blocking

In Chapter 3, we showed that using the Hilbert Curve to order the edges of a graph improved the performance of PageRank for various datasets and vertex orderings. However, our case study used a single-threaded PageRank implementation. This raises the question:

<table>
<thead>
<tr>
<th>Research Question 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can we parallelize edge-centric algorithms that use the Hilbert curve to order edges and still benefit from the improved cache locality that is gained by a single-threaded traversal using the Hilbert Curve order?</td>
</tr>
</tbody>
</table>

RHuBarb is our answer to this question. Rhubarb is an edge reordering and blocking technique that uses Recursive Hilbert Blocking and is designed for parallel computation of edge-centric algorithms. Parallelizing edge-centric algorithms that use the Hilbert Curve presents two main challenges: synchronization and load-balancing. Rhubarb uses Spray Reductions [54] (See Section 4.2.3) to synchronize writes to shared vertex data from multiple threads and Recursive Hilbert Blocking to evenly distribute work to threads. Section 5.1 explains why we need to synchronize threads when using the Hilbert Curve. We discuss an alternative approach for synchronization used by Cagra [97] and show how RHuBarb uses Block Reductions to implement synchronization. Section 5.2 introduces the problem of imbalanced distribution of edges in adjacency matrices of real-world graphs and how Recursive Hilbert Blocking ensures an even distribution of edges in our blocking approach. We describe the Recursive Hilbert Blocking algorithm and how we parallelize it using OpenMP tasks. By using OpenMP dynamic scheduling and chunking, Recursive Hilbert Blocking maintains the locality preservation property of the Hilbert Curve (See Section 2.4.2).
5.1 Thread Synchronization using Spray Reductions

A simple way to parallelize edge-centric computation using the Hilbert Curve is to split the graph’s adjacency matrix into equally sized quadrants. Then, assign each thread a different quadrant to work on in parallel. Each thread iterates over the edges of its assigned quadrant, accesses the data of the source and destination vertices of that edge, and performs some computation based on that data. However, this simple parallelization requires synchronization: if multiple threads concurrently update the same vertex’s data, this can cause a race condition. We illustrate this issue in Figure 5.1: if Thread 1 processes the edge \((0, 1)\) at the same time that Thread 2 processes the edge \((2, 1)\), unsynchronized updates to destination vertex 1 can be lost. Note that synchronization is not required in pull-direction parallel processing (See Section 2.3.2), since each thread writes exclusively to a disjoint region of the shared vertex data array.

![Figure 5.1: Potential race condition due to parallel processing of edges using the Hilbert curve. (Figure adapted from [43]).](image)

In the foundational paper describing Propagation Blocking, an optimization used to improve the spatial locality of PageRank, Beamer et al. [14] mention that the “temporal transformations” associated with the Hilbert Space Filling Curve “can greatly complicate parallelization”. Previous work tried the following approach for synchronization.

Cagra [97], an in-memory GPS, uses CSR Segmenting to split the vertex set into segments that fit in the LLC and partitions the graph based on those segments. In their evaluation, Zhang et al. [97] tested two approaches to synchronize parallel Hilbert-ordered updates. The first, HAtomic, uses atomic
compare-and-swap updates. The second, HMerge, creates per-thread private arrays to store updates and merges them at the end. Note that HMerge is equivalent to the default OpenMP array reduction described in Section 4.2.3. Zhang et al. [97] report that HMerge does not scale. HMerge achieves, at best, a 4x speedup using 24 cores to compute the PageRank of the twitter graph, with only 5% percent of the runtime spent merging the private arrays. They attribute the poor scalability of HMerge to Cache Contention. In HMerge, each thread works on a disjoint region of the Hilbert-ordered edgelist. Each thread accesses regions of the shared vertex data arrays that may or may not overlap. More importantly, the sizes of these regions can vary. While Hilbert ordering improves the locality for each thread, threads that are working concurrently on regions that do not temporally/spatially overlap will competitively share the LLC, which results in the poor scalability observed by Zhang et al. [97]. We illustrate this issue on a sample graph in Figure 5.2.
Figure 5.2: Locality of access to shared vertex data by HMerge [97]. Subfigure (a) shows the adjacency matrix of a social network. Each pixel in the \((i, j)^{th}\) coordinate corresponds to the edge \((i, j)\) in the graph. Subfigure (b) shows the division of the Hilbert-ordered edges between four threads. Edges are coloured according to their assigned threads. Although each thread processes roughly the same number of edges, the area of the adjacency matrix covered by each thread varies significantly due to the uneven distribution of edges in the matrix. For example, a clustering of edges in the upper-left quadrant of the matrix corresponds to the small area covered by Thread 1. The left and top rectangular axes of Subfigures (b-c) highlight the regions of the vertex data array from which each thread will read and to which each thread will write, respectively. Subfigure (c) overlays the Hilbert curve (starting from the top-left) over Subfigure (b) to show the order of edge traversal for each thread. The start and endpoints of each thread’s edgelist segment are annotated. While there is spatial overlap in the vertex data regions accessed by each thread, there is no guarantee that the threads will access the overlapping vertices in the shared data array at the same time (i.e., temporal overlap), which can lead to competitive sharing of the cache.
Rhubarb achieves synchronization by using Block Reductions (Section 4.2.3). We treat Edge-centric algorithms as sparse array reductions on the vertex data array of the destination vertices. All threads iterate over their assigned Hilbert-ordered edges in parallel. Each thread uses Block Reductions to accumulate updates to the destination vertices it encounters. Once all threads complete their assigned work, we reduce the thread-local updates, and store the final result in the global vertex data array. This reduction is equivalent to the “merge” operation performed by HMerge. Block Reductions avoid the allocation of large thread-private arrays and enable scalability.

5.2 Load Balancing and Locality Preservation using Recursive Hilbert Blocking and OpenMP Scheduling

Splitting up the adjacency matrix into quadrants with equal sidelength leads to an uneven distribution of edges between threads. Since quadrants act as the base unit of work that is assigned to threads, the varying numbers of edges contained in these quadrants leads to imbalanced workloads. Consequently, some threads could finish their work earlier than others (i.e., there could be stragglers). Figure 5.3 illustrates this issue.
(a) Adjacency Matrix of the CiaoDVD Social Network graph [60]. A marked blue pixel in the \((i,j)^{th}\) coordinates corresponds to the existence of an edge \((i,j)\) in the graph. The graph’s edges have been partitioned into quadrants of size \(1024 \times 1024\).

(b) Number of edges per quadrant in Figure 5.3a.

Figure 5.3: Uneven distribution of edges due to quadrant partitioning.

Cagra’s approach with HMerge [97] (Section 5.1) divides the Hilbert ordered edgelist into equal sizes, ensuring an even distribution of edges among threads. But, this method cannot scale due to cache contention.

Recursive Hilbert Blocking is an edge reordering and blocking technique that ensures a balanced
distribution of edges between adjacency matrix blocks. We describe the Recursive Hilbert Blocking algorithm and provide pseudocode for each of its procedures. Then, we discuss the use of OpenMP scheduling and chunking to dynamically assign groups of consecutive blocks to threads.

5.2.1 Parallel Divide-and-Conquer using OpenMP tasks

Instead of requiring that all quadrants have equal size, we divide the matrix into blocks whose sidelength can be an arbitrary power of 2. We say an adjacency matrix block \( b \) is anchored at the point \((i, j)\) if \((i, j)\) is the top-leftmost point of \( b \), i.e., the smallest source and destination vertices incident on edges in \( b \) are \( i \) and \( j \), respectively. If we fix \( b \) with a sidelength \( s \in \{2^k \mid k \in \mathbb{Z}, 1 \leq k \leq 32\} \), then \( b \) will contain all edges \( \{(u,v) \mid i \leq u < i+s, j \leq v < j+s\} \). We use \( b_{i,j}^\top \) to denote the block anchored at \((i,j)\) with a sidelength \( s \).

Given a graph \( G \) stored using a CSR, \( C \), we can compute the number of edges contained in the block \( b_{i,j}^\top \) by using the property that, for each \( u \in G \), the out-neighbours of \( u \) will be stored contiguously in a sorted order in the subarray of \( C\text{.neighbours} \) defined by the interval: \([C\text{.index}[u],C\text{.index}[u+1])\). We iterate over the source vertices located in \( b \): \([i,i+s)\). For each source vertex, \( u \), we binary search the range \([C\text{.index}[u],C\text{.index}[u+1])\) of \( C\text{.neighbours} \) to compute the indices of the first and last destination vertices that are in \( b \). Specifically, we are searching for the first out-neighbour of \( u \) whose ID is \( \geq j \) and the last out-neighbour of \( u \) whose ID is \( < (j+s) \). If \( \text{low} \) and \( \text{high} \) are the indices of the first and last out-neighbours of \( u \) in \( b \), respectively, then the difference between the indices \( (\text{high} - \text{low}) \) is the number of out-neighbours of \( u \) that are contained in \( b \). We show the pseudocode for this procedure in Algorithm 11. In Algorithms 11 and 12, we use the lower_bound(arr, start, end, val) function that, given an array \( arr \), returns the index of the first element in the subarray of \( arr \) defined by the range \([\text{start},\text{end})\) which is greater than or equal to \( \text{val} \).

Algorithm 11 Procedure to return the number of edges in an adjacency matrix block.

\begin{verbatim}
Input C: Graph stored in CSR format
u_{start},u_{end}: start and end source vertices
v_{start},v_{end}: start and end destination vertices

Output The number of edges in Graph C that lie within the block [u_{start} : u_{end}, v_{start} : v_{end}]

1: function \text{N\_EDGES\_IN\_BLOCK}(C,u_{start},u_{end},v_{start},v_{end})
2:   \text{nnz} = 0
3:   \textbf{for} \ u \ \text{in} \ \text{range}(u_{start},u_{end}): \ \textbf{do}
4:     \text{neigh\_start} = C\text{.index}[u]
5:     \text{neigh\_end} = C\text{.index}[u+1]
6:         \text{\triangleright} \text{Start, Endpoint of } u \text{'s neighbourhood}
7:     \text{\textbf{end}}
8:     \text{\triangleright} \text{Binary search to find the index of the first and last out-neighbour}
9:     \text{low} = \text{lower\_bound} (C\text{.neighbours}, \text{neigh\_start}, \text{neigh\_end}, v_{start})
10:    \text{high} = \text{lower\_bound}(C\text{.neighbours}, \text{low}, \text{neigh\_end}, v_{end})
11:    \text{nnz} += \text{high} - \text{low}
12: \textbf{return} \text{nnz}
\end{verbatim}

\footnotesize
1We limit blocks to a sidelength of at most \(2^{32}\) since most real-world graph datasets contain fewer than \(2^{32}\) vertices. Currently, the largest publicly available real-world graph dataset is \text{Hyperlink2012} [72] with \(3,563,602,789\) vertices.

56
We perform two binary searches for each source vertex to find the first and last destination vertices contained in $b$. If $s$ is the sidelength of $b$ (i.e., $s = u_{end} - u_{start} + 1 = v_{end} - v_{start} + 1$), there are at most $s$ source and destination vertices in $b$. So, the time complexity of Algorithm 11 is $O(s \lg s)$. We show a similar procedure to aggregate the edges of $b$ into an array in Algorithm 12.

**Algorithm 12** Procedure to return the edges in an adjacency matrix block.

**Input**
- $C$: Graph stored in CSR format
- $u_{start}, u_{end}$: start and end source vertices
- $v_{start}, v_{end}$: start and end destination vertices

**Output**
A list of edges in Graph $C$ that lie within the block $[u_{start}, u_{end}, v_{start}, v_{end}]$

1: function EDGES_IN_BLOCK($C, u_{start}, u_{end}, v_{start}, v_{end}$)
2:     edges = []
3:     for $u$ in range($u_{start}, u_{end}$):
4:         neigh_start = $C$.index[$u$]
5:         neigh_end = $C$.index[$u + 1$]  # Start, Endpoint of $u$’s neighbourhood
6: # Binary search to find the index of the first and last out-neighbour
7: # of $u$ that lies in this block
8:     low = lower_bound($C$.neighbours, neigh_start, neigh_end, $v_{start}$)
9:     high = lower_bound($C$.neighbours, low, neigh_end, $v_{end}$)
10:    for offset in range(low, high):
11:        $v = C$.neighbours[$offset$]
12:        edges.append((u, v))
13:    return edges

Algorithms 11 and 12 are the main subroutines of the **Recursive Hilbert Blocking** algorithm. To ensure that every block contains a similar number of edges, we define $\mu$ as the maximum number of edges a block can contain. We recursively split the adjacency matrix until the number of edges within a block is less than or equal to $\mu$. If the number of edges in a block is greater than $\mu$, then we split that block into 4 equally sized subproblems and recurse. Otherwise, we create a new block, sort its edges by ascending Hilbert Index, and compute the block’s Hilbert Index using the block’s anchor point. The Hilbert Index of a block $h_{i,j}$ is $h(i, j)$. Finally, we sort all blocks by ascending Hilbert Index. We describe Recursive Hilbert Blocking in Algorithm 13.

We can use the Master Theorem [17] for divide-and-conquer recurrences to calculate the computational complexity of Recursive Hilbert Blocking. At each recursive step, we use Algorithm 11 to compute the number of edges in the current block. Then, we either:

- divide the current block with sidelength $n$ into 4 blocks with sidelength $n/2$, and recurse. Or,

- construct a new block with the edgelist of this block, and return.

Therefore,

$$T(n) = 4T(n/2) + \Theta(n \lg n)$$

$$T \in \Theta(n^2)$$

A quadratic complexity is prohibitive for most graph algorithms since most real-world graphs contain
Algorithm 13 Recursive Hilbert Blocking

**Input** $C$: Graph stored in CSR format
- $u_{\text{start}}, u_{\text{end}}$: start and end source vertices
- $v_{\text{start}}, v_{\text{end}}$: start and end destination vertices
- $\mu$: the maximum number of edges in a Recursive Block
- $\text{blocks}$: an accumulator list of Recursive Blocks

**Output** $\text{blocks}$: The final list of Recursive Blocks

1: function $\text{RHB}(C, u_{\text{start}}, u_{\text{end}}, v_{\text{start}}, v_{\text{end}})$
2: $\text{nnz} = \text{n_edges_in_block}(C, u_{\text{start}}, u_{\text{end}}, v_{\text{start}}, v_{\text{end}})$
3: if $\text{nnz} \leq \mu$ then
4:     $\text{edges} = \text{edges_in_block}(C, u_{\text{start}}, u_{\text{end}}, v_{\text{start}}, v_{\text{end}})$
5:     sort $\text{edges}$ by ascending Hilbert Index $(h(u, v))$
6:     $b = \text{new RecursiveBlock}()$
7:     $b.\text{edges} = \text{edges}$
8:     $b.\text{hilbert_index} = h(u_{\text{start}}, v_{\text{start}})$
9:     $\text{blocks}.\text{append}(b)$
10: else
11:     $u_{\text{mid}} = \frac{u_{\text{start}} + u_{\text{end}}}{2}$
12:     $v_{\text{mid}} = \frac{v_{\text{start}} + v_{\text{end}}}{2}$
13:     $\text{RHB}(C, u_{\text{start}}, u_{\text{mid}}, v_{\text{start}}, v_{\text{mid}})$
14:     $\text{RHB}(C, u_{\text{mid}}, u_{\text{end}}, v_{\text{start}}, v_{\text{mid}})$
15:     $\text{RHB}(C, u_{\text{start}}, u_{\text{mid}}, v_{\text{mid}}, v_{\text{end}})$
16:     $\text{RHB}(C, u_{\text{mid}}, u_{\text{end}}, v_{\text{mid}}, v_{\text{end}})$

millions, if not billions, of vertices. However, if we use vertex orderings such as Parallel Slashburn to preprocess the graph, we can guarantee that most of the adjacency matrix will be empty. Once a recursive call encounters an empty block, it does not recurse further. This property allows us to prune large subtrees of the recursive call tree and is why we implement Recursive Hilbert Blocking using a top-down, rather than a bottom-up, approach.

To further speedup Recursive Hilbert Blocking, we parallelize the algorithm using OpenMP tasks [79]. The task directive is used to parallelize recursive algorithms. It specifies a unit of work that can be done in parallel by any thread. OpenMP tasks are useful when the number of iterations or recursions is not known at the beginning of the computation [46]. We illustrate the parallelization of Recursive Hilbert Blocking using tasks in Figure 5.4. First, in Line 5 of Figure 5.4, we initialize a parallel region. This creates a team of threads that will execute the region. Then, in Line 7, we instruct a single thread to perform the first recursive call on the entire adjacency matrix. Below, we show how Recursive Hilbert Blocking allows recursive calls to be allocated to other threads in the pool.
void recursive_hilbert_block(CSR c, int mu)  
{
    int n = c.num_vertices;
    int nn = pow(2, ceil(log2(n)));
    #pragma omp parallel
    {
        #pragma omp single
        {
            ...
            RHB(c, 0, nn, 0, nn, blocks, mu);
        }
        #pragma omp taskwait
    }
}

**Figure 5.4:** Parallel Recursive Hilbert Blocking using OpenMP tasks.

Figure 5.5 shows the parallel recursive function. To ensure that multiple threads do not concurrently modify a global array of blocks, each thread maintains a thread-private array of blocks. If the number of edges in the block corresponding to this recursive call is less than or equal to $\mu$, then the thread instantiates the block and adds it to its thread-private array. Otherwise, the thread creates four recursive tasks (Lines 18-24). When a thread encounters a task construct, it may execute the task immediately or place the task in a pool of tasks to be executed.
```c
void RHB(CSR c,
        int u_start, int u_end,
        int v_start, int v_end,
        vector<Block> blocks, int mu)
{
    int nnz = n_edges_in_block(c,
        int u_start, int u_end,
        int v_start, int v_end);
    if (nnz <= mu)
        // create new block, compute its hilbert index,
        // and append it to blocks
        ...
    }
else
{
    int u_mid = (u_start + u_end) / 2;
    int v_mid = (v_start + v_end) / 2;
    #pragma omp task // top left
    RHB(c, u_start, u_mid, v_start, v_mid, blocks, m);
    #pragma omp task // top right
    RHB(c, u_start, u_mid, v_mid, v_end, blocks, m);
    #pragma omp task // bottom left
    RHB(c, u_mid, u_end, v_start, v_mid, blocks, m);
    #pragma omp task // bottom right
    RHB(c, u_mid, u_end, v_mid, v_end, blocks, m);
}
```

**Figure 5.5:** The recursive divide-and-conquer function of Recursive Hilbert Blocking, parallelized using OpenMP tasks.

Once all recursive calls are finished, we synchronize the threads in the parallel region using a `taskwait` directive (Line 12 in Figure 5.4).

### 5.2.2 OpenMP Dynamic Scheduling

A thread-scheduling policy decides which threads process which blocks at what time. Two common policies available in OpenMP are: Static and Dynamic. OpenMP Static scheduling divides the iterations of a loop equally among the available threads. Each thread is assigned a predefined chunk of iterations to execute. If the user provides a `chunksize` parameter, the loop iterations will be split into chunks of that size. By default, the iteration space is split evenly into chunks that are approximately equal in
size, and at most one chunk is distributed to each thread. Static scheduling is deterministic and does not incur any runtime overhead for workload balancing. However, if the amount of work per iteration varies, static scheduling may result in load imbalance.

OpenMP Dynamic scheduling assigns loop iterations dynamically to the available threads at runtime. By default, each chunk is one iteration. Similarly to static scheduling, a user can specify a chunksize parameter that determines the number of iterations per chunk. Once a thread completes its assigned chunk, it requests another chunk from a central work queue. Dynamic allocation of iterations helps achieve load balancing among threads, as each thread can request new work when it becomes available. Dynamic scheduling handles load imbalance better than static scheduling, especially when the workload of each iteration is not known in advance. However, the dynamic nature of the scheduling introduces additional runtime overhead due to the need for thread synchronization and workload distribution. We show the differences between static and dynamic schedules in Figure 5.6.
OpenMP static scheduling with static and dynamic input workloads and different chunk-sizes. Using a static schedule without specifying a chunksize is the same as splitting the iteration space evenly between threads and assigning each thread at most one chunk. If the work per iteration is constant (i.e., a static workload), then a static schedule divides the work evenly between the available threads. If a workload is dynamic or the amount of work per iteration is unknown at compile time, a static schedule can produce an unbalanced work assignment to threads, as is seen by the Static, 8 schedule for the Dynamic Input Workload.

OpenMP dynamic scheduling with dynamic input workloads and different chunksizes. Using a dynamic schedule without specifying a chunksize is the same as specifying a chunksize of 1. Dynamic schedules are designed for dynamic workloads where the amount of work per iteration varies. Note that the most balanced work division happens when using a Dynamic schedule with a chunksize of 1. As we increase the chunksize, the granularity of chunk assignment decreases. This reduces the overhead due to dynamic thread scheduling during parallel execution but can cause workload imbalance, as is seen by the variable total work done per thread for the Dynamic, 8 schedule for the Dynamic Input Workload.

Figure 5.6: OpenMP static vs. dynamic schedules for static and dynamic workloads and different chunksizes. On the left of Plots (a) and (b), we show static and dynamic input workloads, split into 32 iterations, where each block represents an iteration. The amount of work per block is proportional to the darker filled area of that block. Each thread assignment corresponds to a schedule type and chunksize. For example, “Static, 4” uses a Static schedule with a chunksize of 4. Not specifying a chunksize is equivalent to using the OpenMP default. On the right of each plot, we show the cumulative amount of work per thread for an input workload, schedule, and chunksize.
The scale-free nature of real-world graphs will undoubtedly produce some variations in the densities of blocks. Even though we minimize this variation through the combined use of Recursive Hilbert Blocking and vertex orderings such as Parallel SlashBurn, the amount of work per block will still vary. Therefore, static scheduling is unsuitable for Rhubarb. We use dynamic scheduling and expose the chunksize ($\delta$) as a hyperparameter to the user. This presents the following tradeoff:

- A larger chunksize leads to poorer load balancing, since it may cause the dynamic thread scheduler to assign a disproportionately heavy workload to a single thread by grouping dense blocks together. Conversely, a small chunksize produces better load balancing by dividing the workload more finely among threads, reducing the chance that any thread lags behind.

- However, a larger chunksize also groups together a larger number of contiguous Recursive Hilbert Blocks. The thread processing these blocks can benefit from spatial and temporal locality, since the vertices incident to the edges in the blocks will share the same vertex data regions. Using a small dynamic chunksize may assign disjoint chunks of blocks to threads, detracting from the spatial locality of the Hilbert curve.

We explore the space of RHuBarb’s hyperparameters in our evaluation (Chapter 6) and provide heuristic guidelines to help users choose the appropriate dynamic chunksize for their application and graph.
Chapter 6

Evaluation

In Chapter 3, we presented the results of our single-threaded PageRank case study, where we explored different combinations of vertex and edge orderings and found that the benefits of SlashBurn and Hilbert ordering compound to nearly always produce the best performance of all combinations we evaluated: SlashBurn clusters the edges so that they fit in a minimal number of blocks, and the Hilbert order leverages this spatial locality of edges and enables cache reuse. These results motivated us to parallelize the SlashBurn algorithm (Chapter 4) and develop Recursive Hilbert Blocking and Rhubarb (Chapter 5). Next, we evaluate the performance of Rhubarb. We answer the following research questions:

<table>
<thead>
<tr>
<th>Research Question 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>The performance of Rhubarb depends on three user-defined parameters:</td>
</tr>
<tr>
<td>1. ( \mu ), <em>Maximum Number of Edges per Block</em>: each Hilbert block must contain <em>at most</em> this many edges.</td>
</tr>
<tr>
<td>2. ( \delta ), <em>Dynamic Chunk Size</em>: How many consecutive Hilbert blocks should be dynamically assigned to cores during Edge-Centric traversal?</td>
</tr>
<tr>
<td>3. <em>BlockReduction size</em>: the size of the blocks used for Spray BlockReduction.</td>
</tr>
</tbody>
</table>

How should users choose these values?
Research Question 5

How expensive is Rhubarb’s preprocessing?

5.1: 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?

5.2: How long does Recursive Hilbert Blocking take? Does the speedup justify the cost of preprocessing?

Research Question 6

How does Rhubarb compare to other implementations of edge-centric algorithms used by State-of-the-Art GPS?

Research Question 7

Does Rhubarb maintain the Locality Preservation property of the Hilbert Curve?

We begin by describing the graph datasets and algorithms we use in our evaluation and the GPS against which we compare. Sections 6.1-6.3 list the graph datasets, algorithms, and processing systems we used in our evaluation, respectively. Section 6.4 describes our experimental setup. Section 6.5 answers Research Question 4, Section 6.6.1 answers Research Question 5, Sections 6.6.2 and 6.6.3 answer Research Question 6, and Section 6.7 answers Research Question 7.

6.1 Graph Datasets

We list the datasets we use in our evaluation in Table 6.1. sx-stackoverflow (sx) [60] is a user-interaction network from Stack Overflow [1]. soc-LiveJournal1 (lj), twitter (tw), and friendster (fr) [60] are social networks of LiveJournal [4], Twitter [5], and Friendster [42], respectively. dimacs9-USA (usa) is the directed road network of the USA. wikipedia_link_en (en) is a hyperlink network of Wikipedia articles in the English language. sd-arc (sd) is a domain level hyperlink network [73] extracted from Common Crawl [29]. yahoo-song (ys) [60] is a bipartite person-song rating network. movielens-1b_rating (ml) [15, 48] is a bipartite person-movie rating network. movielens-1b_rating is a synthetic dataset that is expanded from 20 million real-world ratings. We include the synthetic dataset in our evaluation since it is the largest, publicly available, bipartite rating network known to us. Our graph dataset contains networks that differ in both size and type, facilitating a thorough evaluation of Rhubarb’s performance (and its competitors) in a different set of scenarios. For each unipartite graph, we compute the following vertex orders: Descending Degree Sort, HubSort [97], HubCluster [40], Rabbit Order [7], and SlashBurn [66]. We include this representative set of vertex orderings, as opposed to only using SlashBurn, to evaluate the effect these orderings
have on the performance of the different systems for the algorithms in our evaluation. While it is possible to order the vertices of bipartite graphs based on their degree (i.e., sort $V_1$ and $V_2$ by descending degree separately and order the disjoint vertex subsets in separate regions of the vertex data array), the SlashBurn and Rabbit-order algorithms were designed to order unipartite graphs, so we use only the original vertex ordering for the bipartite graphs. We leave a comprehensive evaluation of the effects of vertex ordering on bipartite graph algorithms for future work. We convert the original vertex labels of all graphs (given in the original edgelist text file) to a compressed representation so that vertex IDs lie in the contiguous range of $[0, n)$. 
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>$n$</th>
<th>$m$</th>
<th>$d^+$</th>
<th>$d^-$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sx</td>
<td>Stack Overflow Interactions</td>
<td>2,584,164</td>
<td>34,875,684</td>
<td>22,257</td>
<td>38,147</td>
</tr>
<tr>
<td>lj</td>
<td>Social Network</td>
<td>4,846,609</td>
<td>68,475,391</td>
<td>13,905</td>
<td>20,292</td>
</tr>
<tr>
<td>us</td>
<td>USA Road</td>
<td>23,947,347</td>
<td>57,708,624</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>ys</td>
<td>Song Ratings</td>
<td>1,625,951</td>
<td>256,804,235</td>
<td>468,366</td>
<td>307,205</td>
</tr>
<tr>
<td>en</td>
<td>.en wikilinks</td>
<td>13,593,032</td>
<td>437,167,958</td>
<td>1,052,128</td>
<td>9534</td>
</tr>
<tr>
<td>ml</td>
<td>Synthetic Movie Ratings</td>
<td>3,052,948</td>
<td>1,223,962,043</td>
<td>69,898</td>
<td>9925</td>
</tr>
<tr>
<td>tw</td>
<td>Twitter Follower</td>
<td>41,652,230</td>
<td>1,468,365,182</td>
<td>2,997,469</td>
<td>770,155</td>
</tr>
<tr>
<td>sd</td>
<td>Hyperlinks</td>
<td>89,247,739</td>
<td>2,043,203,933</td>
<td>2,335,856</td>
<td>1,317,316</td>
</tr>
<tr>
<td>fr</td>
<td>Social Network</td>
<td>68,349,466</td>
<td>2,586,147,869</td>
<td>3124</td>
<td>5214</td>
</tr>
</tbody>
</table>

Table 6.1: Graph Datasets

6.2 Edge-Centric Graph Algorithms

Rhubarb uses the Hilbert curve to hierarchically order the edges of a graph: we order both the Recursive Hilbert Blocks and the edges within each block using the Hilbert curve. Our reliance on Hilbert edge-ordering translates into a restricted set of graph algorithms that we can use to evaluate the performance of Rhubarb, specifically edge-centric algorithms. To test the performance of Rhubarb, we implemented PageRank (6.2.1), Connected Components (6.2.2), and Collaborative Filtering (6.2.3), as edge-centric algorithms. We use BlockReductions (Section 4.2.3) in our edge-centric implementations to synchronize access to shared vertex data from multiple threads.

6.2.1 PageRank

PageRank is a widely adopted benchmark [14] used to evaluate the performance of GPS. We described the PageRank algorithm in Section 3.1. Algorithm 14 shows how we implement PageRank using Rhubarb. Algorithm 14 is almost identical to Algorithm 3 in Chapter 3, except for the addition of the BlockReduction in Line 9 to accumulate intermediate, thread-private updates and synchronize threads.
Algorithm 14 Parallel Edge-Centric PageRank in Rhubarb

**Input** $B$, an array of Recursive Hilbert Blocks, sorted by ascending Hilbert index

$B[i].edges$ are the Hilbert-ordered edges of the $i^{th}$ Recursive Hilbert Block

$\text{deg} : V \rightarrow [n]$, an array of out-degrees; $\text{deg}[i]$ is the out-degree of vertex $i$

$\gamma = 0.85$, PageRank damping factor

$n_{\text{iters}}$, number of PageRank iterations

**Output** $pr : V \rightarrow [n]$, a PageRank array; $pr[i]$ is the PageRank of vertex $i$

```plaintext
1: $n_{\text{blocks}} \leftarrow \text{len}(B)$
2: $p_{\text{prev}} \leftarrow [0, \ldots, 0]$  \Comment{initialize arrays of size $n$}
3: $p_{\text{curr}} \leftarrow [0, \ldots, 0]$
4: for $\text{iteration} \leftarrow 1$ to $n_{\text{iters}}$: do
5: \hspace{1em} parallel_for $i \leftarrow 0$ to $n - 1$ do
6: \hspace{2em} $p_{\text{prev}}[i] = \gamma \ast p_{\text{curr}}[i] / \text{deg}[i]$
7: \hspace{2em} $p_{\text{curr}}[i] = 1 - \gamma$
8: \hspace{1em} end parallel_for
9: $pr_{\text{reduction}} = \text{BlockReduction}(p_{\text{curr}}, n)$  \Comment{declare a spray block reduction}
10: parallel_for $i \leftarrow 0$ to $n_{\text{blocks}} - 1$: do
11: \hspace{1em} $b \leftarrow B[i]$
12: \hspace{1em} $n_{\text{edges in block}} \leftarrow \text{len}(b.edges)$
13: \hspace{2em} for $j \leftarrow 0$ to $n_{\text{edges in block}} - 1$: do
14: \hspace{3em} $(u, v) \leftarrow b.edges[j]$
15: \hspace{3em} $pr_{\text{reduction}}[v] += p_{\text{prev}}[u]$  \Comment{accumulate PageRank contributions}
16: \hspace{1em} end parallel_for
17: return $p_{\text{curr}}$
```

6.2.2 Connected Components

We implement edge-centric Connected Components based on the Shiloach-Vishkin [13, 83] parallel connectivity algorithm. Algorithm 15 details our implementation.
Algorithm 15 Parallel Edge-Centric Connected Components in Rhubarb

Input \( B \), an array of Recursive Hilbert Blocks, sorted by ascending Hilbert index
\[ B[i].edges \text{ are the Hilbert-ordered edges of the } i^{th} \text{ Recursive Hilbert Block} \]

Output \( \text{comp} : V \to [n] \), a Connected Components assignment array;
\( \text{comp}[i] \) is the ID of vertex \( i \)'s component

1: \( n_{\text{blocks}} \leftarrow \text{len}(B) \)
2: \( \text{comp} \leftarrow [0, \ldots, 0] \)
   \( \triangleright \) initialize component array of size \( n \)
3: \( \text{parallel_for } i \leftarrow 0 \text{ to } n-1 \) \( \triangleright \) initialize each vertex as a singleton component
4: \( \text{comp}[i] = i \)
5: \( \text{end parallel_for} \)
6: \( \text{change} \leftarrow \text{true} \)
7: \( n_{\text{iters}} \leftarrow 0 \)
8: \( \text{while } \text{change} \text{ do} \)
9: \( \text{change} = \text{false} \)
10: \( n_{\text{iters}} += 1 \)
11: \( \text{parallel_for } i \leftarrow 0 \text{ to } n_{\text{blocks}} - 1: \)
12: \( b \leftarrow B[i] \)
13: \( n_{\text{edges}}_{\text{in}}_{\text{block}} \leftarrow \text{len}(b.\text{edges}) \)
14: \( \text{for } j \leftarrow 0 \text{ to } n_{\text{edges}}_{\text{in}}_{\text{block}} - 1: \)
15: \( (u, v) \leftarrow b.\text{edges}[j] \)
16: \( \text{comp}_u \leftarrow \text{comp}[u], \text{comp}_v \leftarrow \text{comp}[v] \)
17: \( \text{if } \text{comp}_u == \text{comp}_v \text{ then continue} \)
18: \( \text{high} \leftarrow 0, \text{low} \leftarrow 0 \)
19: \( \text{if } \text{comp}_u > \text{comp}_v \text{ then high } = \text{comp}_u, \text{low } = \text{comp}_v \)
20: \( \text{else high } = \text{comp}_v, \text{low } = \text{comp}_u \)
21: \( \text{if high } == \text{comp}[\text{high}] \text{ then} \)
22: \( \text{change } = \text{true} \)
23: \( \text{comp}[\text{high}] = \text{low} \)
24: \( \text{end parallel_for} \)
25: \( \text{parallel_for } i \leftarrow 0 \text{ to } n-1 \text{ do} \) \( \triangleright \) path compression
26: \( \text{while } \text{comp}[i] \neq \text{comp}[\text{comp}[i]] \text{ do} \)
27: \( \text{comp}[i] = \text{comp}[\text{comp}[i]] \)
28: \( \text{end parallel_for} \)

return \( \text{comp} \)

We start by assigning a unique component ID to all the vertices in the graph, based on their vertex ID (Line 3). We iteratively merge the connected components of the graph until no further merges are possible. In each iteration, we iterate over the Hilbert-ordered edges - and attempt to merge the components of neighbouring vertices. For each edge \( (u, v) \), we compare the component IDs of its incident vertices,
comp\textsubscript{u} and comp\textsubscript{v}. If the component IDs are different, we determine the higher and lower component IDs (Lines 18-20). To propagate the lower component ID, we perform two operations. We set a flag to indicate that a change in component ID assignment has occurred and update the component ID of the higher component to the lower component ID (Lines 21-23).

After iterating over the Hilbert-ordered edges, some vertices can have component IDs that do not directly point to their root vertex (the lowest vertex ID in the component). We use Path Compression to repeatedly update the component IDs of the graph until all vertices point to the ID of the root vertex (Lines 25-28). The algorithm terminates once we complete an iteration without noting a change in component ID assignment.

6.2.3 Collaborative Filtering

Collaborative Filtering is used in the context of machine learning for recommender systems [80]. The task of Collaborative Filtering is to compress a user-item matrix into a low-dimensional representation in terms of latent factors. A user-item matrix is equivalent to a sparse weighted bipartite graph \( B(U, V, E) \), where \( U \) are users, \( V \) are items, and \( E \) are the ratings users give to items. Specifically, \( B_{i,j} \) is the rating of the \( i \)th user for the \( j \)th item. Collaborative Filtering reduces the sparse bipartite graph into a dense representation [80] that is used to identify items that are nearer to one another in the lower-dimensional space.

Given a bipartite adjacency matrix of ratings, \( B \in \mathbb{R}^{n \times n} \), where \( n \) is the number of users, \( n \) is the number of items, and \( n = n_1 + n_2 \), Collaborative Filtering uses incomplete matrix factorization [59] to compute two dense matrices \( P \in \mathbb{R}^{n_1 \times k} \), \( Q \in \mathbb{R}^{n_2 \times k} \), where \( k \) is a hyperparameter denoting the dimension of the latent factors [89]. Formally, matrix factorization minimizes Equation (6.1) [89],

\[
\min_{P,Q} \sum_{(u,v) \in B} (B_{u,v} - p_u q_v^T) + \lambda \|p_u\|^2 + \lambda \|q_v\|^2
\]  

(6.1)

where \( p_u \) is the latent vector of user \( u \), \( q_v \) is the latent factor of item \( q \), and \( \lambda \) is a regularization constant. We can use Gradient Descent (GD) to compute \( P \) and \( Q \). For each rating \( B_{u,v} \), we calculate the difference between the rating and the inner product of the latent vectors \( p_u \) and \( q_v \).

\[
err_{u,v} = B_{u,v} - p_u q_v^T
\]  

(6.2)

Then, we use the computed error for all \((u,v) \in B\) to perform a gradient descent step, and update \( P \) and \( Q \):

\[
p_u \leftarrow p_u + \alpha (err_{u,v} q_v - \lambda \times p_u)
\]  

(6.3)

\[
q_v \leftarrow q_v + \alpha (err_{u,v} p_u - \lambda \times q_v)
\]  

(6.4)

where \( \alpha \) is the GD step-size. Algorithm 16 shows our edge-centric Collaborative Filtering implementation.
Algorithm 16 Parallel Edge-Centric Collaborative Filtering in Rhubarb

Input $B$, an array of Recursive Hilbert Blocks, sorted by ascending Hilbert index

- $B[i].edges$ are the Hilbert-ordered edges of the $i^{th}$ Recursive Hilbert Block
- $K$, dimension of latent vector
- $\lambda$, regularization constant
- $\alpha$, step size
- $n_{\text{iters}}$, number of Collaborative Filtering iterations

Output $\text{latent}: V \rightarrow [n \times K]$, latent vector array;

the latent vector of vertex $u$ is stored in $\{\text{latent}[u \times K], \ldots, \text{latent}[(u + 1) \times K - 1]\}$

1: $n_{\text{blocks}} \leftarrow \text{len}(B)$
2: $\text{error} \leftarrow [0, \ldots, 0]$ \hfill $\triangleright$ initialize arrays of size $n \times K$
3: $\text{latent} \leftarrow [0, \ldots, 0]$ \hfill $\triangleright$ initialize arrays of size $n \times K$
4: $\parallel$ for $i \leftarrow 0$ to $n \times K - 1$ do $\triangleright$ randomly initialize latent vectors
5: $\text{latent}[i] \leftarrow \text{uniform}(0, 1)$
6: end parallel_for
7: for iteration $\leftarrow 1$ to $n_{\text{iters}}$: do
8: $\text{error\_reduction} = \text{BlockReduction}(\text{error}, n \times K)$ \hfill $\triangleright$ declare a spray block reduction
9: $\parallel$ for $i \leftarrow 0$ to $n_{\text{blocks}} - 1$: do
10: $b \leftarrow B[i]$
11: $n_{\text{edges\_in\_block}} \leftarrow \text{len}(b.edges)$
12: $\parallel$ for $j \leftarrow 0$ to $n_{\text{edges\_in\_block}} - 1$: do
13: $(u, v, w) \leftarrow b.edges[j]$
14: $\text{estimate} = 0$
15: $\parallel$ for $k \leftarrow 0$ to $K - 1$ do $\triangleright$ inner-product of latent vectors for $u, v$
16: $\text{estimate} += \text{latent}[v \times K + k] \times \text{latent}[u \times K + k]$
17: $\text{err} = w - \text{estimate}$
18: $\parallel$ for $k \leftarrow 0$ to $K - 1$ do $\triangleright$ accumulate errors
19: $\text{error\_reduction}[v \times K + k] += \text{err} \times \text{latent}[u \times K + k]$
20: end parallel_for
21: $\parallel$ for $i \leftarrow 0$ to $n - 1$ do $\triangleright$ perform gradient descent step
22: $\parallel$ for $j \leftarrow 0$ to $K - 1$ do
23: $\text{latent}[K \times i + j] += \alpha \times (-\lambda \times \text{latent}[K \times i + j] + \text{error}[K \times i + j])$
24: end parallel_for
return $\text{latent}$

6.3 Graph Processing Systems

We compare Rhubarb to a selection of Graph Processing Systems (GPS) known for enabling efficient parallel graph algorithms. Our evaluation assesses how Rhubarb’s optimizations, specifically Recursive
Hilbert Blocking and dynamic thread scheduling, perform in comparison to modern GPS. We determine whether these optimizations, which offer enhanced locality, result in overall speedup, taking into account preprocessing costs associated with the systems. We specifically consider in-memory GPS, as Rhubarb stores Recursive Hilbert Blocks in memory. Moreover, as Rhubarb is designed to run on a single machine, we excluded distributed GPS from our comparison. While our selected GPS needed to support or enable efficient parallel graph algorithms, we did not restrict our selection to strictly edge-centric implementations. We surveyed the relevant GPS literature, and based on prominent and recurring candidates, we choose three in-memory GPS for our comparative study. By demonstrating Rhubarb’s competitiveness against these systems, we establish the scalability and effectiveness of our parallelization approach using the Hilbert curve.

6.3.1 Ligra

Ligra [84] is a lightweight framework designed for parallel graph processing on multicore machines. It provides two map primitives, VertexMap and EdgeMap, that allow developers to express their application logic to operate on either the vertices or edges of the graph [28]. Ligra uses two different data structures to represent VertexSubsets of active vertices or Vertex Frontiers during graph traversal. Shun and Blelloch [84] observe that the frontier of active vertices can be dense or sparse, depending on the graph algorithm and its stage of execution. A dense frontier implies that most, if not all, vertices are active, while a sparse frontier consists of only a few vertices. Ligra represents sparse frontiers using an unordered set of vertex IDs and dense frontiers using a bit-vector of size $n$, with bit $i$ set to 1 if vertex $i$ is active, and 0 otherwise. Ligra takes advantage of the changing size of the frontier during BFS traversal. It implements a direction-optimizing BFS [12] that changes the direction of edge traversal (push vs. pull) based on the frontier’s size. Although Ligra is relatively dated (2015), the project remains well-maintained with recent contributions and serves as an established baseline for comparison. Ligra uses atomic operations for parallel Connected Components and PageRank and pull-direction traversal for parallel Collaborative Filtering.

6.3.2 GraphMat

GraphMat [89] is a shared-memory GPS designed for large-scale graph analytics. GraphMat uses a vertex-centric programming model that allows developers to express graph algorithms using sparse linear algebra operations. This programming model simplifies algorithm design and provides a high-level of abstraction for graph computations [28]. To achieve high performance, GraphMat uses a Doubly Compressed Sparse Column (DCSC) [25] format to store large sparse graphs efficiently with minimal memory overhead. GraphMat parallelizes graph processing by using row-partitioning and exploiting thread-level and Single Instruction, Multiple Data (SIMD) parallelism. GraphMat implements parallel PageRank and Collaborative Filtering using a generalized Sparse Matrix-Vector Multiplication (SPMV) kernel. We use a Connected Components implementation of the label propagation algorithm from a fork \footnote{https://github.com/altanh/GraphMat} of the original GraphMat implementation. The Label Propagation algorithm starts by assigning ver-
tices to singleton components. Label Propagation merges two neighbouring components by considering the component IDs of the components and assigning the minimum component ID to both components. The algorithm iteratively merges the components of neighbouring vertices until convergence.

6.3.3 GBBS: Graph Based Benchmark Suite

The Graph Based Benchmark Suite (GBBS) [33] provides implementations of parallel algorithms for 20 important graph problems. The optimizations and techniques used by Dhulipala et al. [33] enable their implementation to outperform existing state-of-the-art GPS on the largest real-world graph datasets. GBBS uses Ligra’s EdgeMap and VertexMap abstractions for its parallel implementation and enhances parallel efficiency by using a custom work-stealing thread scheduler [21]. Parallel PageRank is implemented using pull-direction traversal. We implemented Parallel Collaborative Filtering by adapting the Ligra implementation. Parallel Connected Components is implemented using the recursive algorithm from Shun et al. [86]. First, an input graph is decomposed into clusters of smaller diameter. Each cluster is contracted into a single vertex, adding inter-cluster edges and removing duplicate edges and isolated vertices. If the contracted graph consists of isolated “meta”-vertices, then the clusters are the components in the original graph, and the algorithm returns the mapping from vertices to clusters. Otherwise, the algorithm recurses on the clustered graph.

6.4 Experimental Setup

Table 6.2 shows the hardware specification of the machines we use in our experiments. We use AWS for the scaling experiments and Xeon for all other experiments. We run all experiments using Ubuntu 22.04.2 LTS.

<table>
<thead>
<tr>
<th></th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AWS</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel® Xeon® Platinum 8375C CPU @ 2.90GHz</td>
</tr>
<tr>
<td># Cores</td>
<td>32</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>48 KB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1.25 MB</td>
</tr>
<tr>
<td>LLC Cache Size</td>
<td>54 MB</td>
</tr>
<tr>
<td>Main Memory Size</td>
<td>512 GB</td>
</tr>
</tbody>
</table>

Table 6.2: Hardware specification of systems used in evaluation.

As per the authors’ instructions, we compile:

1. Ligra using the OpenCilk-2.0.0 compiler [77],
2. GraphMat using the Intel® icpc (ICC) Compiler Version 2021.9.0 20230302, and
3. GBBS using g++ (Ubuntu 12.1.0-2ubuntu1~22.04) 12.1.0.
For the scaling experiments, we limit the number of available cores to either 1, 2, 4, 8, 16, or all 32 cores on AWS. For all other experiments, we use all 14 cores on Xeon. We instrument Rhubarb using Likwid [91] to measure the cache miss rate during the execution of edge-centric traversals. We implement Rhubarb and all edge-centric applications in C++. We compile Rhubarb using the Intel® icpx Compiler Version 2023.1.0 and parallelize our implementations using OpenMP 5.1 [79] and Spray [54]. Rhubarb and the edge-centric applications are publicly available at https://github.com/ubc-systopia/rhubarb.

### 6.5 Effects of Hyperparameters

We evaluate the performance effect of Rhubarb’s three user-defined parameters.

1. $\delta$, Dynamic Chunk Size: the number of consecutive Hilbert blocks that are dynamically assigned to cores during Edge-Centric traversal.

2. $\mu$, Maximum Number of Edges per Block: each Hilbert block contains at most this many edges.

3. Spray Reduction block size (Section 4.2.3).

We analyze the end-to-end performance of Rhubarb as a function of varying these parameters. We show the results of the analysis for twitter in Figure 6.1. Our complete evaluation includes a representative subset of graphs from our dataset and more hyperparameter configurations. However, for the sake of brevity, we do not include the plots for these results since the results are similar to Figure 6.1. We include the full set of plots in Appendix A.1.
Figure 6.1: Preprocessing and execution times for PageRank on twitter using Xeon with 14 cores as a function of the maximum number of edges per block, $\mu$, dynamic chunk size, $\delta$, and Spray Block Width. All times are rounded to one decimal place. Cells with blue borders denote the fastest execution time, omitting the preprocessing time (shown on the left). The cell with the red border denotes the fastest overall execution time.
Figure 6.1 shows that as we increase $\mu$, the preprocessing times decrease correspondingly. A larger $\mu$ means fewer recursive calls required to reach the base case reducing the overall time required for Recursive Hilbert Blocking. The figure shows that there is some leniency in choosing the optimal values for $\delta, \mu$ and the Spray reduction block size. That is, the best performing hyperparameter configuration is only slightly better than its neighbouring configurations in either dimension.

The region in the hyperparameter space that corresponds to the best PageRank runtime is when the product of the block width and the dynamic chunk size ($\delta$) aligns with the L2 cache size. Specifically, we used a machine with a 1MB L2 Cache for the hyperparameter configuration experiment, and a float datatype to represent PageRank values. We see that the best performing results were around the configuration with a block width of 65,536 and $\delta = 4$, since a thread that processes a group of four consecutive Recursive Hilbert Blocks would write to at most a range of size $65,536 \times 4 \times 4 = 1,048,576$ bytes, or 1MB. Based on the results of this experiment, we advise users to use the size of the private L2 cache as a heuristic in choosing these values, and to ensure that the overall working set size that is assigned to each thread is not several orders of magnitude larger than the L2 cache size. But, we acknowledge two main issues with this heuristic:

1. using the L2 cache and information about vertex data size inherently ties the performance of Rhubarb to a specific algorithm and architecture, and
2. block sidelengths are not constant. In fact, vertex orderings such as SlashBurn and Descending Degree Sort will inevitably produce blocks with variable sidelengths depending on the location of the block in the adjacency matrix.

We elaborate further on this issue in the discussion in the following chapter, where we propose possible modifications and improvements to Rhubarb.

6.6 Comparison to Other Edge-Centric Implementations

We compare Rhubarb to the state-of-the-art, main-memory GPS described in Section 6.3. We report the overhead due to vertex ordering and preprocessing and compare the runtime of Rhubarb to state-of-the-art GPS using the graph datasets and algorithms described in Section 6.1 and Section 6.2, respectively. We evaluate the scalability of the systems by measuring PageRank runtime as a function of an increasing number of cores and measure the cache performance of Rhubarb.

6.6.1 Preprocessing

We divide the preprocessing overhead in our evaluation into two components: vertex ordering and graph preprocessing. For vertex ordering, we use parallel implementations of Descending Degree Sort, HubCluster, HubSort from the DBG repository [40], the original implementation of Rabbit Order [7], and Parallel SlashBurn (Chapter 4). Table 6.3 lists the execution times of these parallel vertex orderings.
Table 6.3: Runtimes of Parallel Vertex Orderings using Xeon with 14 cores. Given 128GB of RAM, Rabbit order was unable to reorder sd-arc and friendster due to insufficient memory. Although Parallel SlashBurn is consistently slower than Rabbit order (the other comparably “midweight” reordering algorithm), its memory consumption is smaller, allowing it to reorder the larger sd-arc and friendster graphs.

We assume a baseline representation of CSR for all systems. We define graph preprocessing overhead as any additional computation performed beyond CSR construction. Ligra and GBBS do not perform any additional preprocessing, resulting in zero overhead. However, Rhubarb’s Recursive Hilbert Blocking and GraphMat’s Doubly Compressed Sparse Column (DCSC) construction contribute to the overall execution time of a graph analytic task. Table 6.4 shows these preprocessing overheads.

Table 6.4: Preprocessing times for GraphMat (GM) and Rhubarb (RHB) using Xeon with 14 cores. Times are in seconds. With 128 GB of RAM, we were unable to preprocess sd and fr using GraphMat. On average, Rhubarb’s preprocessing is $3.07 \times$ faster than the time required to construct the doubly-compressed sparse column (DCSC) representation used by GraphMat.
6.6.2 Runtime

Unipartite Graphs

Tables 6.6 and 6.7 show the performance of the systems.

For all systems, we run the PageRank and Connected Components algorithms until convergence. We adopt the convergence criteria of Langville and Meyer [62] for PageRank and define $\epsilon$, the sum of the differences between the PageRank values of successive iterations, to be $10^{-9}$. Table 6.5 lists the number of PageRank iterations required to reach the convergence criteria for the graphs in our dataset.

<table>
<thead>
<tr>
<th>Graph Code</th>
<th>Number of Iterations to Converge</th>
</tr>
</thead>
<tbody>
<tr>
<td>sx</td>
<td>77</td>
</tr>
<tr>
<td>lj</td>
<td>80</td>
</tr>
<tr>
<td>usa</td>
<td>91</td>
</tr>
<tr>
<td>en</td>
<td>69</td>
</tr>
<tr>
<td>tw</td>
<td>83</td>
</tr>
<tr>
<td>sd</td>
<td>96</td>
</tr>
<tr>
<td>fr</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 6.5: The number of PageRank iterations to converge for $\epsilon = 10^{-9}$. 
<table>
<thead>
<tr>
<th>Graph Name</th>
<th>Vertex Order System</th>
<th>Original</th>
<th>Rabbit</th>
<th>HubCluster</th>
<th>HubSort</th>
<th>Sort</th>
<th>SlashBurn</th>
</tr>
</thead>
<tbody>
<tr>
<td>sx-stackoverflow</td>
<td>GBBS</td>
<td>1.25</td>
<td>1.24</td>
<td>1.21</td>
<td>1.16</td>
<td>1.11</td>
<td>1.03</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>1.79</td>
<td>1.78</td>
<td>1.78</td>
<td>1.73</td>
<td>1.73</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>2.18</td>
<td>2.03</td>
<td>1.88</td>
<td>2.19</td>
<td>2.14</td>
<td>1.74</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>1.03</td>
<td>0.99</td>
<td>0.90</td>
<td>0.90</td>
<td>0.85</td>
<td>0.61</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>GBBS</td>
<td>2.41</td>
<td>2.22</td>
<td>2.44</td>
<td>2.61</td>
<td>2.64</td>
<td>2.56</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>3.97</td>
<td>3.85</td>
<td>3.98</td>
<td>4.15</td>
<td>3.98</td>
<td>3.91</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>4.52</td>
<td>3.65</td>
<td>4.82</td>
<td>5.46</td>
<td>5.63</td>
<td>5.39</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>2.42</td>
<td>2.19</td>
<td>2.27</td>
<td>2.32</td>
<td>2.18</td>
<td>1.63</td>
</tr>
<tr>
<td>dimacs9-USA</td>
<td>GBBS</td>
<td>7.40</td>
<td>7.36</td>
<td>8.76</td>
<td>9.17</td>
<td>7.32</td>
<td>9.65</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>7.69</td>
<td>7.10</td>
<td>8.06</td>
<td>8.09</td>
<td>8.13</td>
<td>8.24</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>3.75</td>
<td>3.45</td>
<td>5.78</td>
<td>6.64</td>
<td>5.24</td>
<td>6.87</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>5.05</td>
<td>6.77</td>
<td>8.38</td>
<td>9.41</td>
<td>7.59</td>
<td>5.42</td>
</tr>
<tr>
<td>wikipedia.link.en</td>
<td>GBBS</td>
<td>9.05</td>
<td>8.88</td>
<td>9.42</td>
<td>10.61</td>
<td>10.08</td>
<td>10.20</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>22.32</td>
<td>22.33</td>
<td>22.67</td>
<td>22.00</td>
<td>21.57</td>
<td>21.93</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>21.54</td>
<td>19.64</td>
<td>22.24</td>
<td>38.20</td>
<td>37.36</td>
<td>40.70</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>10.00</td>
<td>9.20</td>
<td>9.16</td>
<td>9.23</td>
<td>8.08</td>
<td>7.09</td>
</tr>
<tr>
<td>twitter</td>
<td>GBBS</td>
<td>66.49</td>
<td>72.93</td>
<td>61.81</td>
<td>60.43</td>
<td>59.50</td>
<td>52.80</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>93.43</td>
<td>95.82</td>
<td>95.68</td>
<td>89.47</td>
<td>87.25</td>
<td>85.30</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>176.20</td>
<td>193.00</td>
<td>164.00</td>
<td>343.60</td>
<td>321.00</td>
<td>351.40</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>58.76</td>
<td>64.15</td>
<td>50.98</td>
<td>49.61</td>
<td>47.58</td>
<td>29.32</td>
</tr>
<tr>
<td>sd-arc</td>
<td>GBBS</td>
<td>146.66</td>
<td>OOM</td>
<td>140.86</td>
<td>132.78</td>
<td>129.10</td>
<td>100.20</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>337.00</td>
<td>OOM</td>
<td>315.00</td>
<td>318.80</td>
<td>316.80</td>
<td>228.20</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>179.03</td>
<td>OOM</td>
<td>131.69</td>
<td>115.95</td>
<td>106.98</td>
<td>71.52</td>
</tr>
<tr>
<td>friendster</td>
<td>GBBS</td>
<td>154.30</td>
<td>OOM</td>
<td>149.22</td>
<td>150.70</td>
<td>150.23</td>
<td>136.20</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>828.00</td>
<td>OOM</td>
<td>793.20</td>
<td>828.60</td>
<td>678.20</td>
<td>287.20</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>156.20</td>
<td>OOM</td>
<td>108.73</td>
<td>138.96</td>
<td>125.60</td>
<td>114.08</td>
</tr>
</tbody>
</table>

**Table 6.6:** PageRank runtimes, excluding preprocessing, using Xeon with 14 cores. The fastest runtime per graph is highlighted using bold, red text. Cells labeled OOM indicate that the system ran out of memory. For almost all scale-free networks (sx-stackoverflow, soc-LiveJournal1, wikipedia.link.en, twitter, and sd-arc), using Rhubarb and SlashBurn results in the fastest runtime to compute the PageRank of the graph. For sx-stackoverflow, soc-LiveJournal1, wikipedia.link.en, and twitter, the next best performing combination is Rhubarb and Descending Degree Sort. This is due to the similarity between Descending Degree Sort and SlashBurn as well as Rhubarb’s ability to efficiently traverse dense colocated adjacency matrix blocks. We use Figure 6.2 to illustrate the similarity between the sparsity patterns of the adjacency matrices of SlashBurn and Descending Degree Sort.
Figure 6.2: Recursive Hilbert Blocks of wikipedia_link_en with SlashBurn and Descending Degree Sort vertex ID assignments, color-coded by the number of edges within each block (maximum 131,072 edges per block). Both Rhubarb and Descending Degree Sort cluster the majority of the edges in the graph in top-left quadrant of the adjacency matrix. Rhubarb leverages this clustering by dividing the dense region into smaller blocks than those in the sparser regions of the matrix. This facilitates parallel traversal using the available cores with a finer granularity of core-to-block assignment compared to the sparser regions of the graph. Specifically, since Rhubarb uses a constant block chunksize for thread assignment, concurrent threads that process the blocks in the dense upper-left quadrant share the read and write working set for the vertex data of the hub vertices. The sparser regions in the adjacency matrix, represented by lighter and larger blocks, are assigned to separate threads through individual, non-overlapping BlockReduction regions. This reduces the memory overhead required per iteration to aggregate thread-private contributions from all threads.
<table>
<thead>
<tr>
<th>Graph Name</th>
<th>Vertex Order System</th>
<th>Original</th>
<th>Rabbit</th>
<th>HubCluster</th>
<th>HubSort</th>
<th>Sort</th>
<th>SlashBurn</th>
</tr>
</thead>
<tbody>
<tr>
<td>sx-stackoverflow</td>
<td>GBBS</td>
<td>0.05</td>
<td>0.07</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>0.34</td>
<td>0.34</td>
<td>0.33</td>
<td>0.28</td>
<td>0.28</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>0.05</td>
<td>0.05</td>
<td>0.04</td>
<td>0.04</td>
<td><strong>0.04</strong></td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>0.05</td>
<td>0.10</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.04</td>
</tr>
<tr>
<td>soc-LiveJournal</td>
<td>GBBS</td>
<td>0.09</td>
<td>0.10</td>
<td>0.08</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>1.04</td>
<td>1.07</td>
<td>1.02</td>
<td>1.14</td>
<td>1.10</td>
<td>1.10</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>0.09</td>
<td>0.08</td>
<td>0.09</td>
<td>0.11</td>
<td>0.11</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>0.10</td>
<td>0.36</td>
<td>0.13</td>
<td>0.13</td>
<td>0.27</td>
<td>0.18</td>
</tr>
<tr>
<td>dimacs9-USA</td>
<td>GBBS</td>
<td>1.23</td>
<td>1.18</td>
<td>0.80</td>
<td>0.52</td>
<td>0.46</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>OOT</td>
<td>OOT</td>
<td>OOT</td>
<td>OOT</td>
<td>OOT</td>
<td>OOT</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>21.90</td>
<td>26.68</td>
<td>26.82</td>
<td>5.71</td>
<td>5.16</td>
<td>5.80</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>0.19</td>
<td>0.19</td>
<td>0.24</td>
<td>0.29</td>
<td>0.27</td>
<td>0.28</td>
</tr>
<tr>
<td>wikipedia_link_en</td>
<td>GBBS</td>
<td>0.23</td>
<td>0.30</td>
<td><strong>0.22</strong></td>
<td>0.24</td>
<td>0.23</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>4.45</td>
<td>5.01</td>
<td>4.55</td>
<td>3.78</td>
<td>3.68</td>
<td>3.79</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>0.41</td>
<td>0.38</td>
<td>0.44</td>
<td>0.70</td>
<td>0.78</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>0.30</td>
<td>2.94</td>
<td>0.55</td>
<td>0.57</td>
<td>0.40</td>
<td>0.45</td>
</tr>
<tr>
<td>twitter</td>
<td>GBBS</td>
<td>1.49</td>
<td>1.55</td>
<td>1.32</td>
<td>1.28</td>
<td>1.33</td>
<td>1.34</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>30.16</td>
<td>30.48</td>
<td>30.04</td>
<td>27.96</td>
<td>27.31</td>
<td>27.08</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>2.68</td>
<td>3.15</td>
<td>2.78</td>
<td>5.20</td>
<td>4.53</td>
<td>5.58</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>1.51</td>
<td>35.36</td>
<td>1.33</td>
<td><strong>1.22</strong></td>
<td>1.81</td>
<td>1.55</td>
</tr>
<tr>
<td>sd-arc</td>
<td>GBBS</td>
<td>3.89</td>
<td>OOM</td>
<td>3.22</td>
<td>3.11</td>
<td>3.04</td>
<td><strong>2.63</strong></td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>7.65</td>
<td>OOM</td>
<td>6.88</td>
<td>6.90</td>
<td>6.57</td>
<td>3.95</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>138.97</td>
<td>OOM</td>
<td>113.29</td>
<td>21.91</td>
<td>11.46</td>
<td>28.87</td>
</tr>
<tr>
<td>friendster</td>
<td>GBBS</td>
<td>4.88</td>
<td>OOM</td>
<td>4.11</td>
<td><strong>3.96</strong></td>
<td>3.91</td>
<td>5.73</td>
</tr>
<tr>
<td></td>
<td>GraphMat</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td></td>
<td>Ligra</td>
<td>15.33</td>
<td>OOM</td>
<td>15.32</td>
<td>18.66</td>
<td>18.58</td>
<td>12.52</td>
</tr>
<tr>
<td></td>
<td>Rhubarb</td>
<td>11.28</td>
<td>OOM</td>
<td>30.76</td>
<td>14.35</td>
<td>18.36</td>
<td>17.60</td>
</tr>
</tbody>
</table>

Table 6.7: Connected Components Runtimes, excluding preprocessing, using Xeon with 14 cores. The fastest runtime per graph is highlighted using bold, red text. Cells containing OOM indicate that the system ran out of memory, while cells containing OOT indicate that the tests ran longer than five minutes.

We highlight the following datapoints from Table 6.7.

1. In the case of GraphMat, the implementation of Connected Components using label propagation took over 5 minutes for the high-diameter usa graph for all vertex orders. This result is not surprising, since it is well established that using Label Propagation to compute the Connected Components of high-diameter graphs can perform much worse than standard, edge-centric approaches [84]. However, GraphMat uses a vertex-centric programming model to express graph algorithms as sparse-matrix algebraic operations, and since the authors did not include a refer-
ence implementation of Connected Components, we used this variant of the algorithm that is easily expressed using GraphMat’s programming model.

2. **Rhubarb outperforms GBBS on CC for the usa road network, due to the unusually large diameter of the road network and the nature of the GBBS algorithm.** During each iteration of CC, GBBS decomposes the graph into clusters with diameter \((\lg n)/\beta\), with \(\beta = 0.2\) by default. Given that usa has approximately 20M vertices, the intermediate cluster diameter after the first contraction will be approximately 120. Since the road network has an uncharacteristically large diameter of 8440, this requires more iterative contractions to compute the connectivity of the road network, resulting in reduced performance compared to Rhubarb for this specific test.

3. **Degree-based orderings improve the performance of CC on Ligra on large diameter graphs.** Ligra’s CC implementation iteratively updates each vertex component ID to be the minimum ID entry of its neighbors. For high-diameter graphs, this algorithm performs worse than standard edge-centric algorithms [84]. While sorting vertices by degree disrupts the inherent geographical locality of the dataset, it also reduces the number of iterations needed for convergence. “High” degree vertices are assigned the lowest IDs and are no longer spatially colocated with their geographical neighbours. These lower ID vertices consistently propagate their minimal ID values to their neighbors, resulting in fewer total iterations to converge on global component ID assignments.

4. **Rabbit order is suboptimal for CC on Rhubarb for power-law graphs.** The original vertex ID assignment of wikipedia_link_en and sd-arc uses the hierarchical nature of hyperlink networks to assign IDs to vertices. Rabbit-order instead attempts to detect communities of hyperlinks, which has the undesired effect of degrading the performance of the iterative Shiloach-Vishkin implementation we use in Rhubarb. Figure 6.3 illustrates and compares the sparsity patterns of the adjacency matrices of the Original and Rabbit Order isomorphisms and elaborates on this effect.
Figure 6.3: Recursive Hilbert Blocks of wikipedia_link_en with Original and Rabbit-order vertex ID assignments, color-coded by the number of edges within each block (maximum 131,072 edges per block). The figure shows edge clustering in the upper-left quadrant of the Original adjacency matrix. Rabbit-order uses the detected communities to reorder vertices, shown in the clustered blocks along the diagonal. Using Rabbit-order for CC in Rhubarb proves suboptimal since it increases the number of iterations required for convergence. In the original order, the lower ID vertices account for the majority of the edges in the graph. Rabbit-order relabels these hub vertices and assigns them IDs that are closer to the IDs of the vertices in their communities. Since many of these hierarchical communities are themselves connected to the other communities in the graph, Rabbit-order causes more frequent minimum Component ID relabellings as the Shiloach-Vishkin algorithm progresses, which directly translates to an increased number of iterations to converge (58 with Rabbit vs. 3 with Original), and a longer runtime.

Figures 6.4-6.7 compare the total runtime (including preprocessing) of the systems.
Figure 6.4: PageRank (PR) and Connected Components (CC) total runtimes for sx-stackoverflow and soc-LiveJournal1 using Xeon with 14 cores. Preprocessing and execution times are shown as darker and lighter bars, respectively.
Figure 6.5: PageRank (PR) and Connected Components (CC) total runtimes for dimacs9-USA and wikipedia_link_en using Xeon with 14 cores. Preprocessing and execution times are shown as darker and lighter bars, respectively. GraphMat’s Connected Components runtime on dimacs9-USA exceeded 5 minutes, so we omit it from the plot.
Figure 6.6: PageRank (PR) and Connected Components (CC) total runtimes for twitter and sd-arc using Xeon with 14 cores. Preprocessing and execution times are shown as darker and lighter bars, respectively. Both GraphMat and Rabbit Order ran out of memory trying to preprocess the larger sd-arc graph on our machine with 128 GB of RAM.
Figure 6.7: PageRank (PR) and Connected Components (CC) total runtimes for friendster using Xeon with 14 cores. Preprocessing and execution times are shown as darker and lighter bars, respectively. Both GraphMat and Rabbit Order ran out of memory trying to preprocess the larger friendster graph on our machine with 128 GB of RAM.

Although Rhubarb and SlashBurn consistently deliver the fastest PageRank runtimes among the systems we compared to, the benefits gained are overshadowed by the costs of preprocessing. Despite our parallelization efforts described in Chapters 4 and 5, the “price of admission” for Rhubarb remains prohibitively high for edge-centric algorithms such as PageRank and, especially, Connected Components. Nonetheless, users can amortize the preprocessing cost of Recursive Hilbert Blocking across different algorithm invocations. In the next section, we conduct a similar performance comparison of Rhubarb...
with its competitors on Collaborative Filtering.

**Using Rhubarb for Edge-Centric Algorithms with Large Vertex Data**

Collaborative filtering is a unique graph analytic task in that it presents a tradeoff between complexity and accuracy, expressed in the dimensionality of the latent vector. Also, Collaborative Filtering maintains larger per-vertex state compared to PageRank and Connected Components, resulting in a considerably larger working set size. By using Rhubarb to traverse the edges of the bipartite graph in a locality preserving manner, we can speedup Collaborative Filtering by improving the locality of access to the significantly larger vertex state array.

Collaborative filtering uses gradient descent to minimize the difference between the dot products of latent factors and the weights of edges (i.e., ratings). After completing $i$ iterations of Collaborative Filtering, the sum of squared errors quantifies the approximation accuracy of the incomplete matrix factorization: a lower sum of squared error corresponds to a more accurate approximation of the original user-item bipartite matrix, $B$. Importantly, as a rule-of-thumb, a higher dimensionality ($k$) will result in a lower final error but a longer running time. If the latent space is too low-dimensional, there is a greater chance of information loss during the matrix factorization process, and we may struggle to capture the full complexity of the user-item interactions, leading to higher final error. At the same time, a larger latent vector size requires more floating-point additions and multiplications (seen in the loops on Lines 15, 18 of Algorithm 16), which will translate to a longer execution time. The interaction between accuracy and time of gradient descent is also dependent on the stepsize, $\alpha$, and regularization constant, $\lambda$, but we reserve a thorough exploration of this space for future work. We illustrate the accuracy-time tradeoff for GBBS in Figures 6.8 and 6.9.
(a) The final sum of squared errors and completion time for 500 iterations of Collaborative Filtering on the user-song ratings dataset, yahoo-song [60], as a function of $k$, using Xeon with 14 cores. We set the stepsize, $\alpha$, to $10^{-9}$, and the regularization constant, $\lambda$, to 0.0001.

(b) The final sum of squared errors and completion times for 100 iterations of Collaborative Filtering on the user-movie ratings dataset, movielens-1b_rating [15, 48], as a function of $k$, using Xeon with 14 cores. We set the stepsize, $\alpha$, to $3.5 \times 10^{-7}$, and the regularization constant, $\lambda$, to 0.0001.

Figure 6.8: Tradeoffs between runtime, accuracy, and latent factor dimension ($k$). Although we use GBBS for this evaluation, we verify that given the same hyperparameter configuration (e.g., dimension of latent vector, stepsize, and regularization constant), all systems exhibit the same error curves. Figures 6.8a and 6.8b show that, for both bipartite graphs $ys$ and $ml$, as $k$ grows, the time required for Collaborative Filtering grows linearly, while the final error decreases exponentially.
Figure 6.9: Tradeoffs between runtime, accuracy, and latent factor dimension ($k$). Figures 6.9a and 6.9b present the normalized mean squared error per iteration of Collaborative Filtering for the same graphs and values of $k$. A larger $k$ allows the latent factors to better represent the user-item interactions in the ratings datasets, resulting in a lower initial error. Although larger $k$ values correspond to longer execution times per iteration, increasing the dimensionality of the latent factors enables faster training. For example, the error curve for $k = 48$ in Figure 6.9a takes approximately 400 iterations to achieve the minimal error for this hyperparameter configuration. In Figure 6.9b, although the error curve for $k = 8$ has a higher initial error than $k = \in \{5, 6, 7\}$, it eventually achieves the lowest error among all values of $k$. For both graphs, achieving the same accuracy using fewer dimensions is either impossible or would require a significantly greater number of Gradient Descent iterations.

To see how the different systems handle the complexity-accuracy tradeoff of Collaborative Filtering, we evaluate the performance of the systems on $y_s$ and $m_1$ as a function of $k$. We show the results in Figure 6.10.
Time to complete 500 iterations of Collaborative Filtering on the user-song ratings dataset, yahoo-song [60], as a function of $k$.

Time to complete 100 iterations of Collaborative Filtering on the user-movie ratings dataset, movielens-1b_rating [15, 48], as a function of $k$.

Figure 6.10: Collaborative filtering runtimes using Xeon with 14 cores.

Though the vertices of the bipartite graphs have not been reordered, Rhubarb is able to outperform all systems for most values of $k$. After accounting for the cost of preprocessing (Table 6.4), Rhubarb provides a net-gain in performance improvement for both graphs over all systems and values of $k$. Recursive Hilbert Blocking shines for this type of workload: the relatively large per-vertex data is accessed in a locality preserving manner due to the usage of the Hilbert curve, which allows cores to better utilize the private and shared caches. The Collaborative Filtering results highlight candidate graph workloads with the following characteristics that are ideal for Rhubarb:

1. relatively large per-vertex data (e.g., vectors of 8-byte double as opposed to the traditional 4-byte uint32_t or float).

2. require repeated iterations over the edges of the graph to refine and improve a target such as accuracy or convergence criteria.

3. allow edge iteration in an arbitrary order (e.g., using the Hilbert Curve).

6.6.3 Scaling

We evaluate the scalability of the GPS by measuring the time to complete 20 iterations of PageRank on twitter as we increase the number of cores. We also reorder twitter using Descending Degree Sort, Parallel SlashBurn, and Rabbit-order to see how vertex ordering affects scalability.
Figure 6.11: Runtime and Parallel Scaling of 20 iterations of PageRank on twitter using the Original and Descending Degree Sort Vertex Orderings for \{1, 2, 4, 8, 16, 32\} cores on AWS.
Figure 6.12: Runtime and Parallel Scaling for 20 iterations of PageRank on twitter using the SlashBurn and Rabbit Vertex Orderings for \{1, 2, 4, 8, 16, 32\} cores on AWS.

Figures 6.11 and 6.12 illustrate the following take-aways regarding the scalability of Rhubarb with respect to the vertex ordering used to represent the graph.

1. For all vertex orderings, Rhubarb achieves the fastest single-thread runtimes due to the improved locality achieved through the hierarchical use of the Hilbert curve to order blocks and edges.

2. Ligra is unable to scale past 8 cores for Degree-based orderings such as Descending Degree Sort and SlashBurn. Ligra’s use of atomic operations to synchronize threads results in contention within the contiguous vertex data region containing high-degree vertices, leading to degraded performance.

3. Rhubarb’s scalability is a function of the density and compression of the adjacency matrix. A compressed adjacency matrix, characterized by a minimal number of dense blocks, facilitates better performance. Using the SlashBurn representation, Rhubarb achieves an impressive parallel speedup of 25.82x with 32 cores. However, as the representation becomes less compressed, the parallel speedup decreases to 18.89x, 17.01x, and 16.44x for Descending Degree Sort, Rabbit-order, and the original vertex ordering, respectively. We illustrate this interaction in Figures 6.13-
4. GBBS achieves excellent scalability regardless of the vertex ordering. We attribute this to GBBS’s efficient pull-direction implementation, which avoids atomic operations, and its even distribution of work to threads through a custom, work-stealing scheduler.

Figures 6.13-6.16 show the densities of Recursive Hilbert Blocks and the distributions of block sidelengths for the twitter graph for the Original, Rabbit, Descending Degree Sort, and SlashBurn vertex orderings. In each plot, the top-row shows the Recursive Hilbert Blocks of each ordering. Blocks are color-coded based on the number of edges they contain (maximum 131,072 edges per block). Each plot’s title displays the name of the Vertex Ordering and the corresponding parallel speedup achieved (as seen in Figures 6.11 and 6.12). The bottom-row shows the distribution of Recursive Hilbert Blocks sidelengths by anchor x-coordinate. The anchor x-coordinate of a Recursive Hilbert Block is the smallest source vertex ID incident on the edges in that block.
Figure 6.13: Recursive Hilbert Blocks and distribution of block sidelengths for Twitter using the Original Vertex Ordering.
Figure 6.14: Recursive Hilbert Blocks and distribution of block sidelengths for twitter using the Rabbit Vertex Ordering.
Figure 6.15: Recursive Hilbert Blocks and distribution of block sidelengths for twitter using the Descending Degree Sort Vertex Ordering.
Figure 6.16: Recursive Hilbert Blocks and distribution of block sidelengths for twitter using the Parallel SlashBurn Vertex Ordering.
Figures 6.13-6.16 show that Rhubarb’s scalability is tied to the compression of the adjacency matrix, measured in the number and location of denser blocks - a more compressed representation leads to better scalability for Rhubarb. Descending Degree Sort and SlashBurn place the dense, small blocks in a contiguous region within the adjacency matrix. This reduces the working set of block reductions by limiting the range of writes that will be assigned to each core, further improving Rhubarb’s scalability.

6.7 Locality Preservation of Rhubarb

The Hilbert curve’s locality preservation property ensures that edges that are close to each other in the 2D adjacency matrix will remain close to each other in the 1D Hilbert edge ordering. In our single-threaded case study (Chapter 3), we attributed the faster traversals of Hilbert-ordered edges to better spatial and temporal locality when compared to push/pull edge orders. We also claimed that Rhubarb, through the use of Recursive Hilbert Blocking and OpenMP dynamic scheduling, maintains the locality preservation property of the Hilbert Curve. We substantiate these two claims by comparing the L2 and LLC miss rate for pull-direction vs. Hilbert-ordered PageRank using 1 and 14 cores. We illustrate the results in Figure 6.17.
Figure 6.17: Cache miss rates for 20 iterations of PageRank on twitter using Pull-Direction and Hilbert Edge Orderings, with 1 and 14 cores on Xeon. Each bar represents an edge-ordering (Pull vs. Hilbert) and number of cores. “Hilbert-14” refers to using Rhubarb with 14 cores. When using a single core, we see that the Hilbert curve’s locality preservation property leads to significantly fewer L2 and LLC cache misses, resulting in faster runtimes in our single-threaded case study. For the multicore experiments, we measure the miss rate for each core separately and report the mean and standard deviation. Even after accounting for the standard deviation, using the Hilbert curve and all cores results in a decrease in L2 cache miss rate. This improvement is due to the partitioning of Hilbert-ordered edges and the grouping of contiguous Hilbert-ordered blocks using Rhubarb’s Recursive Hilbert Blocking and dynamic scheduling, which reduces the range of reads and writes assigned to individual cores. This trend does not extend to the LLC. While the multicore Hilbert LLC miss rate is substantially lower than the multicore pull miss rate, it is nearly double than the single-core Hilbert LLC miss rate. While Rhubarb mitigates the competition for the LLC by assigning consecutive groups of blocks to cores, the concurrent access of multiple cores to shared vertex data fills up the shared cache, resulting in a slightly higher number of misses.

6.8 Takeaways

Our evaluation demonstrates that Rhubarb is well-suited for graph analytic workloads that meet the following criteria:

- The vertex data (state) is large (> 4-8 bytes).
- It involves iterative edge-centric computation that refines and improves a target, such as accuracy.

We found that Rhubarb is an ideal candidate for graph analytic tasks such as Collaborative Filtering. Our experiments on the yahoo-song and movielens datasets show that due to the relatively large vertex state and large number of iterations required to reach a minimal approximation error, Rhubarb delivered an end-to-end speedup of $1.22 - 2.33\times$ over GBBS, $1.05 - 2.64\times$ over GraphMat, and $1.89 - 2.61\times$ over Ligra.
While using Parallel SlashBurn and Rhubarb resulted in the fastest runtimes on PageRank for the scale-free networks in our dataset, and, interestingly, the fastest Connected Components runtime on the usa road network, these speedups did not offset the cost of Parallel SlashBurn and Recursive Hilbert Blocking. Rhubarb benefited most when processing graphs reordered using SlashBurn, since the Slash-Burn ordering yields a compressed adjacency matrix. The next-best performing data point was often using the Descending Degree Sort isomorphism. The similarity in speedup was due to the clustering of the hub vertices for both SlashBurn and Descending Degree Sort. This is particularly important since graph algorithms such as $k$-Core [69], Triangle Counting [85], and Graph Coloring [49] either benefit from or rely on degree sorting as a subroutine or preprocessing step. Since users might already store the graph isomorphism based on a degree-based vertex ordering, this increases the viability of Rhubarb as a potential solution for users to perform additional edge-centric analytics.
Chapter 7

Discussion

We place Rhubarb in the context of similar graph processing systems that use the Hilbert curve to speed up graph processing. While Rhubarb is primarily designed for graph processing on a single node, previous work used the Hilbert curve to accelerate graph processing in both distributed and single-node systems. We describe the approach taken by these systems and compare Rhubarb to these implementations. We then acknowledge some limitations of our approach and suggest ways in which future work could improve Rhubarb and Parallel SlashBurn.

7.1 Other Uses of Hilbert Ordering

Mosaic [67] is a GPS that achieved trillion-edge scale graph processing using an out-of-core architecture with fast storage media (e.g. NVMe SSDs) and massively parallel coprocessors (e.g., Xeon Phi). A contribution of Mosaic that is relevant to our work is their use of Hilbert-ordered tiles. A tile is a disjoint set of edges. Mosaic orders tiles using the Hilbert curve and sorts edges within a tile by destination ID (i.e., pull-direction). Mosaic parallelizes computation across, and not within, tiles. However, Mosaic partitions the edges into tiles by balancing the number of unique vertices that are incident to a tile, whereas Rhubarb partitions edges into blocks by balancing the number of edges in a block. Specifically, inside a tile, the number of unique vertices is at most $2^{16}$. Using a smaller range of tile-local vertex IDs allows Mosaic to represent IDs using less memory. Each tile maintains metadata to map local IDs to their global counterparts. While Mosaic specifically targets out-of-core and/or massively parallel architectures, and Rhubarb targets traditional main-memory systems, we believe that limiting the number of unique vertices that lie within a block can further improve the performance of Rhubarb and address the concern we raised in Section 6.5 about the variability of the sidelengths of blocks. We elaborate on this below.

Photon [43] is a distributed GPS that uses the Hilbert Curve to order the edges of the graph for edge-centric algorithms. Photon uses Graph Striping to split the Adjacency Matrix of a graph into stripes of equal length. Each stripe is assigned to a machine. The data of the destination vertices that belong to a stripe are updated only by the machine that owns that stripe. Each machine further splits its assigned stripe into a column of Computation Squares. Photon uses the Hilbert Curve to iterate over the edges
in each computation square. Graph striping is equivalent to the division of work done in parallel pull-direction computation, but, since all stripes have the same width, it is likely that the number of edges per stripe varies greatly, which translates into uneven workloads between machines.

Rhubarb combines techniques from both Photon and Mosaic by ordering blocks and edges within blocks using the Hilbert Curve. Also, Hilbert balances work by ensuring that blocks contain at most \( \mu \) edges, and uses dynamic scheduling to ensure a well-balanced distribution of work to cores.

### 7.2 Limitations and Future Work

We outline several limitations of our work and ways in which it can be improved.

#### 7.2.1 Sparse vs. Dense Blocks and Differentiated Modes of Computation for Rhubarb

Some recent work \([27, 61, 97, 98]\) uses a form of \emph{CSR Segmenting} to accelerate graph analytics. CSR Segmenting partitions the vertices into cacheable subsets that fit in the L2 or LLC. Each core is exclusively assigned a vertex subset, which limits the range of reads and writes for each core and improves cache locality. Several systems \([27, 61, 98]\) combined CSR Segmenting with other optimizations: blocking and ordering of updates to ensure sequential memory access \([61, 98]\) and combining cross-partition updates to reduce memory traffic using a technique known as Propagation Blocking \([14]\).

These optimizations can be integrated into Rhubarb to further enhance its performance. This would involve addressing two interrelated issues with Rhubarb that arise from the variable sidelengths of Recursive Hilbert Blocks. Even when using vertex orderings such as SlashBurn and Degree Sort, the scale-free nature of real-world graphs leads to varying sparsity patterns in different regions of the graph’s adjacency matrix. Orderings such as SlashBurn and Descending Degree Sort benefit Rhubarb precisely because they cluster the edges within a smaller, denser region of the adjacency matrix. Nevertheless, regardless of the graph isomorphism, certain regions of the adjacency matrix will be sparse. In Rhubarb, this phenomenon is evident in the variability of Recursive Hilbert Blocks’ sidelength. As a consequence, some blocks have large sidelengths but contain at most \( \mu \) edges, resulting in sparsity. The existence of large sparse blocks has two negative implications:

1. Variability in blocks’ sidelength prevents us from using a smaller datatype (e.g., \texttt{uint16.t}) to represent vertex IDs within a block, similar to Mosaic’s approach. Limiting the maximum sidelength to \( 2^{16} \) could improve Rhubarb’s performance by representing block edges using less memory.

2. In our single-threaded case study (Chapter 3), we demonstrated that Hilbert ordering performs better on denser subgraphs due to the smaller Average Read-Write Distance of a dense graph. The difference in performance is highlighted in Figure 3.2 when comparing the peak Millions Edges per Second (MEPS) for uniformly random and Kronecker graphs.

Rhubarb could be further improved by distinguishing between blocks and choosing an appropriate mode of computation based on sidelength and density. This could be done by defining a \emph{dense} block as
any block whose sidelength is \( \leq 2^{16} \) and whose density is above a certain threshold, \( \rho \). Rhubarb could then differentially process blocks based on their category:

- Use the Hilbert curve to iterate over the edges of dense blocks (as is currently done by Rhubarb).
- For all other sparse blocks, use a technique like CSR Segmenting and/or Propagation Blocking and iterate over the edges in pull-direction.

We illustrate the variability of blocks’ sidelengths in Figure 7.1.
Figure 7.1: The top plots show the Recursive Hilbert Blocks of wikipedia_link_en with the SlashBurn vertex ID assignment. Blocks whose density is $\geq \rho = 10^{-6}$ and whose sidelength is $\leq 65,536$ are color-coded by their density. All other sparse blocks are blank. The top-left plot shows the entire adjacency matrix, and the top-right plot zooms in on the region containing the majority of the densest blocks. The bottom plot shows the distribution of Recursive Hilbert Blocks sidelengths by anchor $x$-coordinate. Bars corresponding to block sidelengths that are greater than 65,536 are hatched.

Figure 7.1 shows that a majority of the blocks have sidelengths that are $\leq 65,536$. Due to the scale-free nature of the hyperlink en network and the clustering properties of SlashBurn, 79.7% (!) of the edges are in regions covered by dense blocks. This raises an important observation and question: A large proportion of the edges of power-law graphs that have been reordered using SlashBurn lie in a
relatively small area of the adjacency matrix. Can we leverage this property of scale-free networks to compute “approximate” properties of the original graph using this tightly colocated majority of edges? For example, we can use Rhubarb to compute properties (e.g., PageRank, latent factors) of only the vertices that are incident to edges that lie in dense blocks. We know that Rhubarb will perform this task quickly, since the subgraph induced by the dense blocks is both smaller and denser. But, this begs the question: how close are the computed values of the subgraph induced by the dense subset of edges to the values we would have obtained if we included the entire graph in the computation? If the approximation is “close-enough” to the original values, but can be computed in a fraction of the time, exploring this tradeoff between execution time and accuracy may be an extension of Rhubarb that could be worth pursuing.

7.2.2 Improving Parallel SlashBurn

The authors of SlashBurn implement two optimizations to SlashBurn that improve the compression of the adjacency matrix. These optimizations modify two main subroutines of SlashBurn, namely: hub selection and spoke ordering.

First, the basic SlashBurn algorithm (and the one we used in our parallel implementation) selects \( k \) hub vertices to remove from the graph in every iteration in order of descending degree. The authors note that, for \( k > 1 \), after removing \( 1 \leq i < k \) hubs, it may be the case that the \( i + 1 \)th hub has become more isolated, and is longer a high-degree vertex. To address this, they propose to select hubs greedily: whenever we remove a hub, decrement the degrees of all of its neighbours, and use the updated degrees to select the next hubs. Figure 7.2 depicts the difference between Basic and Greedy hub selection.

![Figure 7.2: Basic vs. Greedy k-hub selection. Using Basic k-hub selection and \( k = 6 \), the vertices labelled with 1 – 5 are removed. Vertex 6 is then selected as the next hub due to its high degree in the original graph, even though its degree is 0. Greedy k-hub selection addresses this “incorrect” removal by updating degrees sequentially. (Figure adapted from [66]).](image)

Lim et al. [66] show that using Greedy Hub Selection consistently produced isomorphisms that require a smaller number of blocks. Unfortunately, it is not clear how to parallelize Greedy Hub Selection, since, prior to the removal of the \( i \)th hub, we must remove and update the degrees of all the neighbours of the 1 to \((i - 1)\) hubs. The cost of using a sequential Greedy Hub Selection should be weighed against the performance improvement it imparts on the downstream analytic task that Parallel SlashBurn aims to improve.

The second optimization of SlashBurn deals with an alternative way of ordering spokes. Lim et al. [66] note that as we increase \( k \) (the number of hubs selected in each iteration for removal), the thickness
of the “wings” of the adjacency matrix increases, which directly translates to a larger number of blocks, and worse compression. They attribute this to the ordering of spokes - all connected components of the graph in the current iteration of SlashBurn except the Giant Connected Component GCC. The basic SlashBurn algorithm places the vertices of the spokes into the final vertex ordering based on the descending size of spokes. They improve spoke-ordering by instead ordering spokes according the largest hub vertex ID to which a spoke is connected. To do this, the authors use a modified BFS that they name Blocked-BFS($V'$), where $V' \subset V$. A Blocked-BFS($V'$) is a BFS performed on the graph obtained by removing the vertices in $V'$ and all their adjacent edges (the BFS is “blocked” once it reaches any $v \in V'$). They implement Hub-ordering by performing a Blocked-BFS from the neighbours of the Hub vertices. As opposed to Greedy Hub Selection, Hub-ordering can be parallelized by performing the Blocked-BFS traversals rooted at the neighbours of the $k$ hubs in parallel. Since the authors report improvement to the compression quality of SlashBurn using Hub-ordering of spokes, it is worth investigating the improvement it might provide to Parallel SlashBurn, and by extension, Rhubarb.
Chapter 8

Conclusion

We began this dissertation by asking whether the benefits of vertex and edge ordering compound. We found that for scale-free graphs, using the SlashBurn vertex order and Hilbert edge order produced notable performance improvements over other vertex-and-edge orderings due to the combined improvement to spatial locality by the two techniques: the compressed clustering of edges relabelled using SlashBurn and the locality preservation of edges on the Hilbert curve. This motivated our two novel contributions: a parallel implementation of the SlashBurn algorithm and the Rhubarb edge ordering and blocking technique.

Parallel SlashBurn produces notable speedups, ranging from 4.33× - 11.96×, over the existing sequential implementation. Our optimized parallel version is scalable, achieving a speedup of 6.78× - 8.19× with 14 cores. This optimized implementation significantly reduces the preprocessing cost, making SlashBurn a more viable and lightweight vertex reordering for graph analysis.

Our evaluation of Rhubarb demonstrates its suitability for graph analytic workloads with large vertex data and iterative edge-centric computation that refines a target, such as accuracy. Collaborative Filtering emerges as an ideal candidate for Rhubarb, delivering end-to-end speedups of 1.22× - 2.33× over GBBS, 1.05× - 2.64× over GraphMat, and 1.89× - 2.61× over Ligra.

To the best of our knowledge, Rhubarb enables the first scalable implementation of parallel edge-centric algorithms using the Hilbert curve for main-memory GPS. These promising results provide a foundation for future research exploring the use of Rhubarb for distributed or GPU-accelerated graph processing. Future work could investigate the performance of a distributed graph processing system that uses edge partitioning and leverages Rhubarb for processing the edges on each machine, or assigning Recursive Hilbert Blocks to separate GPU threads to explore further gains in performance and scalability.

Although Parallel SlashBurn and Rhubarb produce the fastest runtimes for certain graph processing tasks, the speedups do not offset the associated preprocessing costs in some cases. We find that Rhubarb’s best performance is achieved when iterating over graphs reordered using SlashBurn or Descending Degree Sort due to their clustering of hub vertices.

Our research sheds light on the complex interactions between vertex and edge orderings in graph processing and highlights the importance of developing efficient preprocessing techniques to enhance...
graph analytic performance. Parallel SlashBurn and Rhubarb pave the way for more effective and scalable approaches to handle tera-scale graphs, opening new possibilities for future advancements in the field of graph processing.
Bibliography


[10] V. Balaji and B. Lucia. When is graph reordering an optimization? In IEEE International Symposium on Workload Characterization (IISWC), 2018. → pages xi, 2, 9, 10, 11, 12, 13, 19


112


113


[71] 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 1, 2, 3, 11, 14, 16, 27


Appendix A

Supporting Materials

A.1 End-to-end performance of Rhubarb as a function of its hyperparameters

We evaluate the performance effect of Rhubarb’s three user-defined parameters.

1. $\delta$, Dynamic Group Size: the number of consecutive Hilbert blocks that are dynamically assigned to cores during Edge-Centric traversal.

2. $\mu$, Maximum Number of Edges per Block: each Hilbert block contains at most this many edges.


We evaluate a representative dataset of graphs

1. sx-stackoverflow: a small contact network,
2. dimacs9-USA: the USA road network,
3. wikipedia_link_en: the English wikipedia hyperlink network, and
4. twitter: a social network.

The hyperparameters we evaluate are:

1. $\delta$, Dynamic Group Size: \( \{2^k \mid k \in \mathbb{Z}, 1 \leq k \leq 8\} \) (1 – 256),

2. $\mu$, Maximum Number of Edges per Block: \( \{2^k \mid k \in \mathbb{Z}, 14 \leq k \leq 19\} \) (16,384 – 524,288), and

3. Spray Reduction block size: \( \{2^k \mid k \in \mathbb{Z}, 12 \leq k \leq 17\} \) (4096 – 131,072).

We show the results in Figures A.1-A.8.
Figure A.1: Preprocessing and execution times for 77 iterations of PageRank on `sx-stackoverflow` as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.
Figure A.2: Preprocessing and execution times for 77 iterations of PageRank on sx-stackoverflow as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.
Figure A.3: Preprocessing and execution times for 91 iterations of PageRank on dimacs9-USA as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.
**Figure A.4:** Preprocessing and execution times for 91 iterations of PageRank on dimacs9-USA as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.
Figure A.5: Preprocessing and execution times for 69 iterations of PageRank on wikipedia_link_en as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.
Figure A.6: Preprocessing and execution times for 69 iterations of PageRank on wikipedia\_link\_en. as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.
Figure A.7: Preprocessing and execution times for 83 iterations of PageRank on twitter as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.
**Figure A.8:** Preprocessing and execution times for 83 iterations of PageRank on twitter as a function of the minimum number of edges per block, $\mu$, dynamic group size, $\delta$, and Spray Reduction Block Width. Cells with blue borders denote the best execution time per row. Cells with red borders denote the best overall execution time.