UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title
Optimizing Sparse Graph and Tensor Algorithms

Permalink
https://escholarship.org/uc/item/68k7f9jp

Author
Lonkar, Amogh

Publication Date
2024

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

OPTIMIZING SPARSE GRAPH AND TENSOR ALGORITHMS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Amogh Lonkar

December 2024

The Dissertation of Amogh Lonkar
is approved:

Asst. Prof. Scott Beamer, Chair

Prof. Seshadhri Comandur

Asst. Prof. Tyler Sorensen

Peter Biehl
Vice Provost and Dean of Graduate Studies

Copyright © by
Amogh Lonkar
2024

Table of Contents

List of Figures **vi**

List of Tables **xv**

Abstract **xvii**

Dedication **xix**

Acknowledgments **xx**

1 Introduction **1**
1.1 Thesis Overview 5

2 Workload Analysis of a Sparse Tensor Decomposition **7**
2.1 Introduction 7
2.2 Background 9
2.2.1 Tensor Data Structures 9
2.2.2 Matricized Tensor Times Khatri Rao Product (MTTKRP) 10
2.3 Experimental Setup 11
2.3.1 Sparse Tensor Frameworks 12
2.3.2 Tensor Dataset 12
2.4 Performance Analysis 13
2.5 Improving Performance 16
2.5.1 Comparison Against Leading Frameworks 19
2.6 Related Works 20
2.7 Conclusion 22

3 Improving Scalability of Pivoting-Based Clique Counting **23**
3.1 Introduction 23
3.2 Background 27
3.2.1 Preliminaries 27
3.2.2 Enumeration-Based K-Clique Counting 29

3.2.3 Pivoting-Based K-Clique Counting 29
3.3 Parallelizing the Ordering Phase 34
3.3.1 Parallel Core-Approximation 34
3.3.2 Centrality-Based Ordering 38
3.4 Maximum Neighbor Influence 40
3.4.1 Work-Locality Tradeoff 41
3.5 Improving Counting Phase Scalability 44
3.5.1 Reducing Memory Consumption 45
3.6 Evaluation 47
3.6.1 Experimental Setup 48
3.6.2 Accelerating the Ordering Phase 49
3.6.3 Work-Locality Tradeoff and Maximum Neighbor Influence 56
3.6.4 Reduction in Memory Usage 60
3.6.5 Parallel Scaling of k-Clique Counting 63
3.6.6 Total Execution Time Comparison 65
3.6.7 Comparison Against GPU 68
3.7 Related Work 73
3.8 Conclusion 74

4 Actor-Based Distributed Breadth-First Search 76
4.1 Introduction 76
4.2 Background 78
4.2.1 Conventional Top-Down BFS 78
4.2.2 Bottom-Up BFS 79
4.2.3 The Actor Model 81
4.2.4 HCLib-Actor Framework 82
4.2.5 The Actor Graph Library (AGL) 83
4.3 Actor-Based Direction-Optimizing BFS 84
4.3.1 Parallel Top-Down BFS 84
4.3.2 Parallel Bottom-Up BFS 86
4.3.3 Parallel Direction-Optimizing BFS 90
4.4 Evaluation 91
4.4.1 Experimental Setup 91
4.4.2 Bottom-Up Comparison 92
4.4.3 Direction-Optimizing BFS 100
4.4.4 Comparison Against Prior Work 101
4.5 Related Work 104
4.6 Conclusion 105

5 Conclusion 107
5.1 Summary of Contributions 108
5.2 Future Work 109

Bibliography **111**

List of Figures

1.1 The sparse graph is stored efficiently in the compressed sparse row (CSR) format. We can access any edge in the graph using the offset (red) and neighbor (yellow) arrays. Typical graph algorithms involve iterating over a vertex's neighbors and performing some computation. In this example, we want to update the score of a vertex by summing the scores of its neighbors. This results in irregular memory accesses since the scores of the neighbors may not be near each other. 2

1.2 Latency (cycles) for random memory accesses to a fixed-size array using a pointer-chase microbenchmark measured on an Intel Xeon Platinum 8260 system. When the array is small enough to fit into the caches (<72MB), the latency is not as high. However, once the array is too big and requests regularly go to DRAM, the latency increases significantly. 3

2.1 A 2-D Tensor stored in the Coordinate (COO) format. The x and y-coordinates of each non-zero are stored in separate arrays with the associated non-zero value. 10

2.2 IPC vs MPKI for MTTKRP in different frameworks run on various tensors in the FROSTT dataset. Higher MPKI results in low IPCs, suggesting that LLC misses are a problem. Low MPKI numbers, coupled with low observed bandwidth utilization suggests that only a few LLC misses are quite problematic. 14

2.3 Fraction of cycles without any new instructions issued vs MPKI for MTTKRP in different frameworks run on various tensors in the FROSTT dataset. Higher MPKI results in more stalls as the processor has to wait for data to arrive from main memory before it can start processing it. We observe that even for a very low MPKI of 5, the processor spends 80% of cycles without issuing any instructions. This means that the performance of current out-of-order processors is highly sensitive to even a few LLC misses. . . . 15

2.4 Poor load miss density of the baseline algorithm. Since generating the address requires additional computation, loads are spaced far apart and the resulting number of memory requests is much smaller than the memory system allows. There is potential to improve performance by amortizing the request overhead by getting more misses in flight. 16

2.5 Speedup vs MPKI for SIMD using manual intrinsics for various tensors. Initially, when the tensor is small enough to fit within cache (low MPKI), SIMD speedup is high due to better resource utilization. As the tensors get bigger, the processor is stalled for significant cycles waiting on data from main memory and the speedup starts to plateau. However, for larger tensors, the speedup is still roughly 1.5× over the baseline. 18

2.6 Speedup vs improvement in bandwidth utilization over the baseline for SIMD, Software Prefetching and combining both. Higher is better for both axes. Increasing bandwidth utilization leads to better performance (higher speedups) for MTTKRP. 20

2.7 Speedups over our baseline for existing frameworks and our SIMD and Software prefetching implementations. Tensors are ordered according from smallest runtime (fastest) to largest runtime (slowest) for the baseline implementation. We present only kernel runtimes, however, both SPLATT and ALTO require additional preprocessing, which if considered, result in runtimes longer than our own implementations. SPLATT is unable to decompose certain larger dimensional tensors and is generally the slowest framework. 21

3.1 Frequency distribution of k-cliques in different graphs. Enumeration-based algorithms are exponentional with respect to clique size (k), and increasing frequency of moderately sized cliques in real-world graphs further exposes this increasing complexity. 25

3.2 Converting an undirected input graph (left) to directed acyclic graph (right) by a degree-based ordering. Furthermore, the highlighted (red) portion on the right indicates the subgraph induced by vertex 0. 28

3.3 Array for sets X, P and R and a dense lookup for the location of vertices within the set while processing vertex 0 in the example DAG (Figure 3.2). Storing the sets in a single array allows for efficiently transferring vertices between X, P and R in between recursion levels. 31

3.4 Differences in the function call heirarchy between enumeration-based algorithms (left) and Pivoter (right) for counting 4-cliques associated with vertex 0 in the example DAG from Figure 3.2. The number in the box denotes the vertex for which the induced subgraph is being built in that level. Vertices are processed sequentially in a depth-first manner. In the heirarchy on the left, the sequence $1 \rightarrow 4$ is processed multiple times. In contrast, the stack for Pivoter is much smaller and avoids redundant computation in the counting phase. 33

3.5 Converting the undirected input graph (left) to a directed acyclic graph (right) by a core ordering. Even though vertex *0* has the highest degree, most of its neighbors are low-degree vertices. Hence it has a lower degree than vertex 1 after peeling vertices 4, 5, and 6, and the edge is directed from $0 \rightarrow 1$. 39
3.6 Differences in degree distribution after producing a DAG using core (left) and degree ordering (right) on the Skitter graph. 41
3.7 Dense structure for storing the neighbor lists of the first-level induced subgraph. 45
3.8 Sparse structure in ComSpark for storing neighbor lists in the first-level subgraph. Instead of a dense array to point to the locations of the neighbor lists, the sparse structure uses a hash map with vertex identifiers as keys and pointers to the neighbor lists as values. . . 46
3.9 Optimized structure with remapping in ComSpark for storing neighbor lists in the first-level subgraph. Only the $d(\vec{v})$-sized remapped array (orange) and the neighbor lists are stored in memory. 47
3.10 Maximum out-degrees produced by changing the error parameter in the parallel approximation algorithms. The maximum out-degrees for core, degree and centrality-based orderings are also included as a reference. As ϵ increases, and more vertices are removed in each round, the maximum out-degree produced by the ordering increases. 51
3.11 Comparison between time to produce various orderings. On larger graphs, our approximation is significantly faster. In addition to being fast, our approximation with $\epsilon = -0.5$ produces the same maximum out-degree as core ordering. This results in the counting phase time between both to be comparable. Degree ordering is always the fastest ordering, but it does not always result in the best counting times. 53

3.12 Comparison between the time to count 8-cliques for each ordering. The core ordering and our parallel approximation generally result in the best counting times due to its algorithmic efficiency. Graphs like DBLP, Baidu and Friendster benefit more from a degree ordering. 54

3.13 Comparison between the total execution times for counting 8-cliques using each ordering. In graphs where core ordering results in the fastest counting, our parallel approximation with $\epsilon = -0.5$ results in much faster overall time due to the fast, parallel ordering. . . . 55

3.14 Differences in the number of instructions executed during the counting phase while counting 8-cliques using both orderings. Lower is better. The instruction count is normalized to that of core ordering. The number of million instructions per-vertex for core ordering is denoted at the top of each bar. Even though degree ordering results in marginally faster counting on some graphs, core ordering is always more algorithmically efficient as fewer instructions are always executed. 56

3.15 Ratio of the number of function calls per-vertex and MPKI between core and degree ordering in the counting phase while counting cliques using our dense structure. Since the maximum out-degree of degree ordering is typically higher, degree ordering results in more instructions than core ordering while counting cliques. More function calls lead to more cache locality since the first-level subgraph is reused in subsequent recursive calls. Degree ordering has a lower MPKI than core ordering for every graph. 57

3.16 Total execution time for counting varied clique sizes using only our core approximation, only the degree ordering, or the ordering selected by our heuristic. Our heuristic always selects the correct ordering for these graphs and it does not add significant overhead. 60

3.17 Comparing the total process memory usage between ComSpark's dense, sparse and remapped structures for counting 8-cliques in the input graphs. Lower is better. Memory usage is normalized to ComSpark's dense structure memory consumption. Dense structure memory consumption (GB) is denoted above each bar. The same ordering is used for all runs. On average, the sparse and remapped structures result in 3.31× less memory consumption. . . 61
3.18 Comparing the reduction in memory usage and MPKI between ComSpark's dense and remapped structures for counting 8-cliques in the input graphs. Lower is better. Both metrics are normalized to ComSpark's dense structure. The same ordering is used for all runs. Compaction in the remapped structure allows a larger fraction of the induced subgraph to reside in cache, reducing the miss rate. 62
3.19 Comparing the performance of different ComSpark memory structures while counting 8-cliques on 64 threads. Higher is better. The same ordering is used for all runs. The remapped structure provides the fast access of the dense structure and the memory compression of the subgraph structure, resulting in good overall performance. . 63
3.20 Comparing parallel scaling of between different subgraph structures in ComSpark for the entire process of counting 6, 12-cliques in all input graphs. The time for each run includes the time to compute the heuristic, and both ordering and counting phases. Both ComSpark structures scale linearly, resulting in better overall performance. For Baidu and Friendster, memory becomes a bottleneck for our dense implementation at 32 threads. Our more compact sparse and remapped structures avoid this and scales linearly even beyond 32 threads. 64

3.21 Total execution time (on a log scale) required for counting cliques of different sizes on the input graphs for each of the CPU algorithms (Pivoter [56], Arb-Count [104], ComSpark) and GPU-Pivot running on an NVIDIA Volta V100 GPU. Lower is better. GPU-Pivot does not report times for $k > 11$. We observe that the lone enumeration-based algorithm (Arb-Count) takes longer for higher values of k. In contrast, the pivoting-based approaches typically do not get slower for higher k. Due to improved parallel scaling, ComSpark is much faster than Pivoter, despite requiring constant time for various k. This allows the inflection point at which pivoting starts to win to decrease from $k = 10$ to $k = 8$ on larger graphs. Due to better scaling, ComSpark outperforms GPU-Pivot for all k on DBLP and larger k on two out of the four common graphs (As-Skitter and Orkut). 66
3.22 Number of function calls required in the counting phase while counting cliques of different sizes. The number of calls are normalized to that of counting 4-cliques in each graph. Counting larger cliques in denser graphs with many cliques like As-Skitter and Orkut result in more work until it plateaus. This is due to an optimization in the original Pivoter code that allows early termination in the counting process. 69
3.23 Self-normalized execution times for GPU-Pivot and ComSpark for As-Skitter and Orkut. We normalize the execution times for various values of k by dividing by the execution time of $k = 4$ for that specific implementation. We observe that the execution time for GPU-Pivot increases with k. In contrast, ComSpark's execution time does not increase significantly. Notably, in the case of As-Skitter, ComSpark has almost constant execution time for all values of k. This allows ComSpark to outcompete GPU-Pivot for larger k. 69

3.24 Self-normalized execution times for GPU-Pivot and ComSpark for LiveJournal. We normalize the execution times for various values of k by dividing by the execution time of $k = 4$ for that specific implementation. While the execution times for both algorithms increase with k, ComSpark shows a slower rate of increase. 70

3.25 Total execution times for counting various k-cliques in LiveJournal using ComSpark and GPU-Pivot on a log-scale. Since GPU-Pivot only reports times up to $k = 8$, we project the performance for counting $k >= 9$ for GPU-Pivot using a regression. We use the equation $y = 0.021 * e^{1.6037x}$ to model the performance of GPU-Pivot. The R-squared value for our regression is 0.99. 71

4.1 Software stack for implementing Actor-based BFS. We implement BFS in the Actor Graph Library (AGL), which is built on top of the HCLib-Actor framework (yellow). The HCLib-Actor framework leverages the Conveyor library (green) for message aggregation. . 84

4.2 Edges checked in each level by different bottom-up implementations for a SCALE 26 R-MAT graph. 93

4.3 Edges checked in each level by different bottom-up implementations for the Twitter graph. 93

4.4 Total number of messages sent during different bottom-up implementations for a SCALE 26 R-MAT graph and the Twitter graph. 94

4.5 Time for each level for different bottom-up implementations for a SCALE 26 R-MAT graph and the Twitter graph. *Sequential* takes 198.61s for the first level on R-MAT and 580.09s and 517.28s for the first two levels on Twitter respectively. 95

4.6 Comparison between weak scaling for various bottom-up implementations. 95

4.7 Impact of different batch parameters on performance. We vary the number of request-response queries for *Threshold* and the number of messages in a batch for *Batch* (right). We measure the time taken for levels 2 and beyond for the R-MAT graph, and levels 3 and beyond for Twitter. 97
4.8 Execution time (left) and messages sent (right) for levels 2-7 of bottom-up BFS on an R-MAT SCALE 26 graph. *Parallel* takes 3.50s for level 2. 98
4.9 Execution time (left) and messages sent (right) for levels 2-7 of bottom-up BFS on the Twitter graph. *Parallel* takes 1.04s for level 3. 98
4.10 Total number of messages sent during the relevant bottom-up levels using different messaging schemes for a SCALE 26 R-MAT graph and the Twitter graph. 99
4.11 Effect of switching heuristic on hybrid BFS performance. 100
4.12 Time for each level for the top-down, bottom-up and hybrid implementations for a SCALE 26 R-MAT graph and the Twitter graph. 101
4.13 Comparison between weak scaling for various implementations. . . 102
4.14 Normalized search rate (MTEPS/Core) for various implementations.102

List of Tables

2.1 Summary of notation used. 10
2.2 Summary of the properties of input tensors taken from FROSTT [105]. 13

3.1 Summary of the properties of input graphs used in the Evaluation. All graphs are unweighted and symmetrized prior to analysis. These graphs are taken from a variety of sources [33, 67, 101]. We only use Soc-Pokec for evaluating our heuristic, and LiveJournal for comparison against the GPU. 48
3.2 Comparison between time taken to convert the input graph into a DAG using the sequential core and degree orderings and the associated counting times for counting 8-cliques. The fastest overall times are bolded. The core ordering is guaranteed to produce the lowest maximum out-degree, which typically reduces the work in the counting phase. 50
3.3 The number of rounds required to order the graph for different values of ϵ. Setting ϵ to -0.5 results in the same maximum out-degree as the core ordering. The number of rounds in this case is still significantly less than the |V| rounds (on the order of millions) required by the sequential core ordering algorithm. Setting ϵ to 50000 effectively results in a degree-based ordering since only one round is required. $\epsilon = 0.1$ is a good compromise between parallelism (number of rounds) and ordering quality (Figure 3.10). 52

3.4 Hardware performance counters for the counting phase of degree ordering normalized to core ordering. Degree ordering always executes more instructions, but executes them faster due to fewer cache misses (MPKI). 58
3.5 Order-selecting heuristic inputs, measurements, and decisions for counting 8-cliques. Our heuristic selects our core approximation $a\delta > 0.15$, or if there are more than 0.10 common neighbors. We select a degree ordering otherwise, or if the graph is very small ($|V| < 1M$). Our heuristic always selects the correct ordering for these graphs. The time to compute the heuristic is tiny. 59
3.6 Summary of total execution time for counting cliques using Pivoter [56], Arb-Count [104], GPU-Pivot [4] and ComSpark. We use the times reported by GPU-Pivot in their paper. Every other algorithm is executed using 64 threads on the same machine (CPU) under the same conditions. The execution times reported include any preprocessing, including graph ordering, but ignore graph reading times. The best CPU execution time is denoted in bold and if the GPU execution time is the fastest, it is denoted in green. 67
3.7 Comparison of CPU and GPU specifications. These values were obtained from [89, 112]. 72
3.8 Performance of the GPU normalized to that of the CPU. We use the power (TDP), transistor and area (mm^2) for each system as reported in Table 3.7. 72

4.1 Various messaging schemes for parallel bottom-up BFS 90
4.2 Difference in amount of communications and synchronization between our Actor-based hybrid BFS and CombBLAS. We use mpiP [116] to profile CombBLAS. 103

Abstract

Optimizing Sparse Graph and Tensor Algorithms

by

Amogh Lonkar

Most big data processing tasks today rely heavily on graph and tensor algorithms to uncover useful information within real-world data. Graph algorithms are used to model relationships between entities, such as social networks. On the other hand, tensor algorithms are essential for processing multi-dimensional data, such as those encountered in machine learning, image recognition, and natural language processing.

Real-world data is sparse and irregular, making these data processing tasks difficult for modern processors. These algorithms require a significant amount of communication between compute and memory, and sometimes between different compute nodes for very large problem sizes. In this dissertation, we analyze graph and tensor workloads to understand data access patterns. We then present optimizations for improving data communication efficiency within the system to improve performance. We consider multiple problems, such as clique counting in graphs (computationally intensive), sparse tensor decomposition (effects of higher dimensionality), and distributed breadth-first search (synchronization and communication across a network).

In this dissertation, we present ComSpark, a clique counting algorithm which scales linearly on CPUs and outperforms GPUs in some cases. To improve the performance of sparse tensor decompositions, we present two software optimizations that improve memory bandwidth utilization. Finally, we reduce the amount of communication required in distributed breadth-first search by using asynchronous

actor messages. Our techniques allow challenging sparse workloads to run faster and enable us to process larger graphs and tensors on current hardware.

To my family

Acknowledgments

I would like to thank many people who have been instrumental during my PhD journey at UC Santa Cruz.

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Scott Beamer, for his guidance and support over these five years. I joined UCSC very green as a researcher, and with minimal experience in computer science. Scott took a chance on me as a student, and for that I will be forever grateful. He has been a true role model as a researcher and person, and his dedication to teaching others is admirable. I could not have had a better mentor to introduce me to graphs, and throughout the years I have learned many things about architecture, parallel computing, and algorithms from our conversations. He struck a great balance between allowing me the freedom to drive projects and providing help, advice and ideas when needed. Scott inspires me to push myself and strive for excellence, and I am better for it.

I would also like to thank the rest of my dissertation committee members, Prof. Seshadhri Comandur and Prof. Tyler Sorensen, for their valuable feedback. Sesh provided useful information on clique counting. Tyler provided several useful suggestions, not just for this dissertation, but also for my Master's Thesis. His feedback undoubtedly made both works better.

I would like to thank all of the members of the VAMA lab for our enjoyable and insightful interactions over the years. Tanuj Gupta, Nishant Khanorkar and Vincent Titterton have been a pleasure to work with on Actors. Sharing an office with Jason Vranek, Haoyuan Wang, Yuanpeng Liao, Priyanka Dutta, Alex Lee, and Jessica Dagostini has been a great experience. Jessica always provided extremely valuable feedback for my research talks, and was a familiar face at Supercomputing. I also thank Sabyasachi Basu and Daniel Paul-Peña for our

conversations on subgraph counting and graph theory. I was lucky to have a rich grad student community at UC Santa Cruz who made my time during the PhD memorable, and interacting with all of my friends has been one of the best parts about this journey. There are too many people to mention by name, but I sincerely thank each and every one.

I am grateful to the members of the FORZA team for their generous support over the last few years. I am especially appreciative of Akihiro Hayashi and Youssef Elmougy for all of their assistance with HCLib. Without their efforts, the BFS chapter in this thesis would not be possible. I also enjoyed getting to know, and interacting with Rich Vuduc, Jeff Young, and Souvi Hati during the site visits.

Last, but certainly not least, I would like to thank my family. Our parents always encouraged me and Omkar to prioritize our education. They made tremendous sacrifices that allowed us to focus on our studies without having to worry about anything. Their endless love and support does not go unnoticed. This journey would not have been possible without them by my side, and this achievement is as much theirs as it is mine. Working and living together with Omkar has been the best part of the last year.

Chapter 1

Introduction

Structures like graphs and tensors represent relations between various entities, such as friends in a social network, interacting proteins in biological processes, vector spaces in quantum mechanics, etc [5, 31, 64, 82, 110]. Graph and tensor algorithms iterate over these structures to uncover useful patterns or relationships between the data. Typically, this intermediate data is then fed as input into machine learning models to accomplish tasks such as classification and recommendation [41, 47, 50, 62, 91, 92, 102].

Data in the real world tends to be sparse. Classic examples include a relatively small number of accounts with a very large number of followers on Twitter, and the frequency of common words like 'the' and 'of' in a text corpus [29, 64]. Additionally, the volume of real-world data available to us keeps increasing. Both of these factors necessitate compressed data structures which avoid storing non-useful information, i.e. zeros. These include Compressed Sparse Row (CSR) for graphs and Coordinate (COO), Hierarchical Coordinate (HiCOO), Compressed Sparse Fiber (CSF) etc. for tensors [69, 107]. While these structures tend to be space-efficient, they result in irregular control flow and memory access patterns, both of which are challenging for performance on current processors (Figure 1.1).

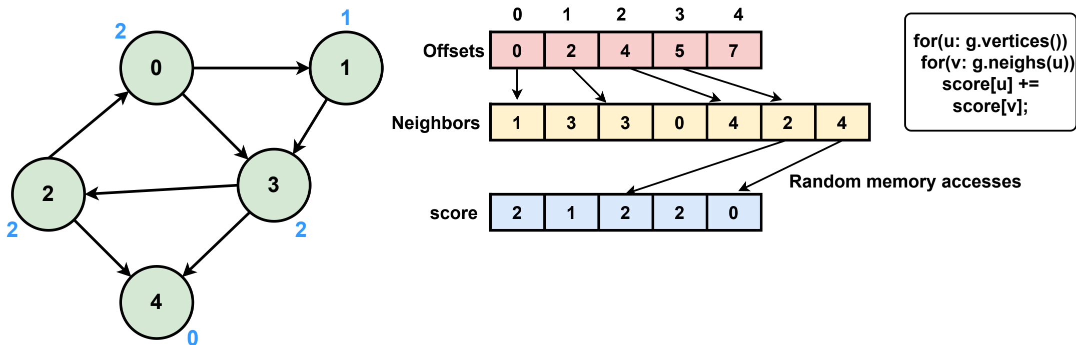

Figure 1.1: The sparse graph is stored efficiently in the compressed sparse row (CSR) format. We can access any edge in the graph using the offset (red) and neighbor (yellow) arrays. Typical graph algorithms involve iterating over a vertex's neighbors and performing some computation. In this example, we want to update the score of a vertex by summing the scores of its neighbors. This results in irregular memory accesses since the scores of the neighbors may not be near each other.

The gap between processor speeds and data transfer rates in DRAM, also known as the memory wall, keeps on increasing [119]. Modern processors employ caches and various speculative mechanisms such as hardware prefetchers to improve performance. Sparse real-world data results in algorithms with irregular memory access patterns, which are not predictable by existing prefetchers. Furthermore, data is too large to fit in fast on-chip caches, leading to a large number of requests to DRAM. For reference, the L1 cache is typically over 100× faster than DRAM, and accesses to DRAM typically stall the processor for hundreds of cycles waiting for the requested data (Figure 1.2). This is especially problematic for sparse graph and tensor algorithms due to their communication-centric nature. We find this inhibits their parallelism, and ultimately hinders performance.

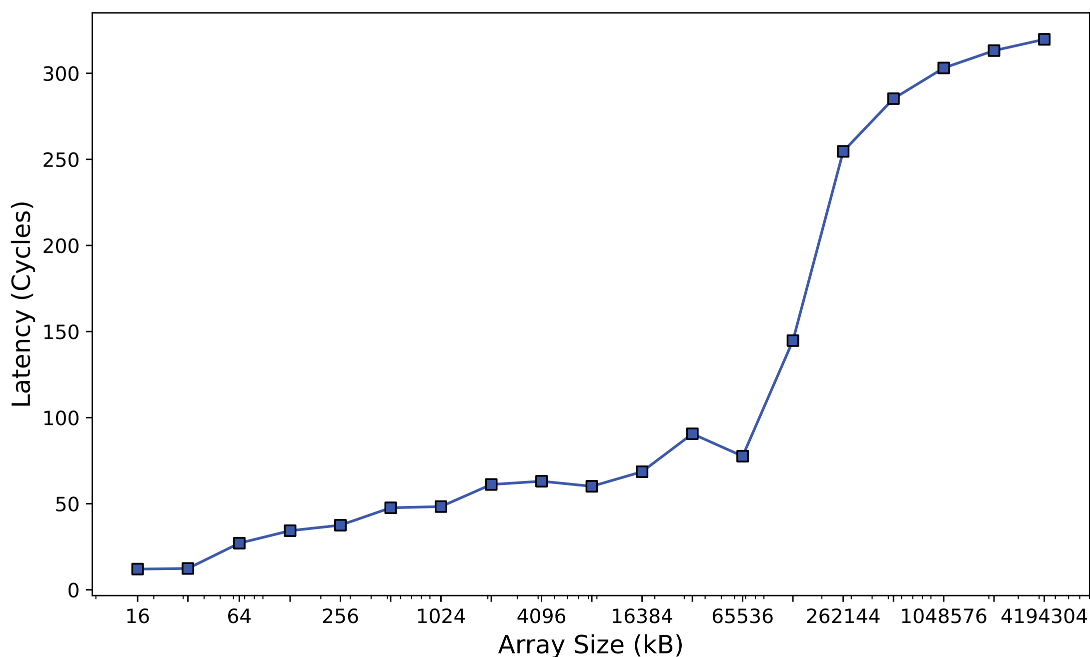

Figure 1.2: Latency (cycles) for random memory accesses to a fixed-size array using a pointer-chase microbenchmark measured on an Intel Xeon Platinum 8260 system. When the array is small enough to fit into the caches (<72MB), the latency is not as high. However, once the array is too big and requests regularly go to DRAM, the latency increases significantly.

Many graph algorithms require large amounts of communication, so they have low arithmetic intensity [8]. Real-world graphs tend to only have a few high-degree vertices, leading to reduced locality in data accesses. As a result, graph algorithms do not fully utilize the compute or memory bandwidth capabilities of modern processors. Sparse tensors face similar performance challenges. Aside from large storage demands due to the sheer size of datasets, non-zeros are spread out far apart and reuse is limited.

The increasing scale of modern data and its irregular nature make sparse graph and tensor algorithms memory-bound. On our dual-socket Intel Xeon system, we measure an 11.48% degredation in the performance of BFS, a communication-intensive workload, for every 10ns increase in memory latency. The performance

of BFS drops significantly (1.67x slowdown) when we access memory on the remote socket. We also measure low instruction per cycle (IPC) rates (< 0.3) for these memory-bound applications on our system. In contrast, we measure IPCs of over 2 for compute-intensive applications like matrix-multiply. This necessitates optimizing data movement between compute and memory to efficiently use resources in modern processors. Data communication efficiency can be improved by reducing the total amount of data moved by doing less algorithmic work, or by increasing throughput via improved memory-level parallelism. In practice, these can be achieved via cache-friendly data structure layouts or general communication efficiency improvements. Prior works improve performance for sparse graph and tensor workloads by reducing cache miss rates [7,12,30,65,85,95,108,118,123], or increasing memory bandwidth utilization [1, 52, 84].

In this dissertation, we explore the effects of parallelism, throughput and latency on performance as given by Little's Law [73] (parallelism = throughput $\times$ latency). We attempt to understand the fundamental issues in data flow across the memory heirarchy, and even the network, in various types of systems. We analyze the workload characteristics for different sparse graph and tensor algorithms using hardware performance counters and performance monitoring tools like Intel VTune. We also closely examine the software for potential inefficiencies. We focus on memory allocation for intermediate data structures, memory access patterns, and the amount of algorithmic work leading to extraneous communication. We also consider how load balance between different threads impacts parallel performance. Based on our analysis, we target specific optimization strategies to improve communication efficiency within the system.

As the future moves towards heterogeneous computing with multiple domain-specific accelerators, efficient data movement between different types of cores be-

comes key to improving performance and maintaining energy efficiency. We hope that the techniques we describe in this dissertation can be applied to those systems.

1.1 Thesis Overview

To introduce the performance traits of a communication-bound algorithm, we present a workload characterization of sparse tensor decompositions in Chapter 2. Tensor decompositions can expose underlying hidden relationships within a dataset. Many applications in data mining, signal processing, image classification, and outlier detection are powered by tensor decompositions. Tensors provide us an opportunity to study the effect of higher dimensionality of data on algorithm performance. In practice, we find that tensor algorithms perform much better on modern processors than graph algorithms. We also find that a small number of last-level cache (LLC) misses are very problematic and cause the processor to stall for a significant amount of cycles. We investigate the effects of different approaches to improve performance by increasing bandwidth utilization such as SIMD and Software Prefetching.

In Chapter 3, we present a heuristic to select the best parallel ordering, and cache-friendly data structures to improve the performance of clique counting. Clique counting is a fundamental task in network analysis, and is used to identify dense clusters in a graph. Clique counting is especially challenging because of the combinatorial explosion in complexity and the high storage requirements of intermediate data structures. Specifically, we consider pivoting-based clique counting which efficiently counts large cliques in sparse graphs. Through our innovations, we are able to achieve linear parallel scaling up to 64 cores, resulting in the fastest parallel CPU implementation to the best of our knowledge.

Finally, we consider the impact of communication across a network in a distributed system for breadth-first search (BFS) in Chapter 4. We consider BFS because it is a fundamental algorithm for traversing through a graph. By leveraging the Actor model, we reduce the amount of high latency network messages and synchronization overhead to improve performance of distributed BFS.

In Chapter 5, we summarize our contributions and suggest potential future work.

Chapter 2

Workload Analysis of a Sparse Tensor Decomposition

In this chapter, we lay out a methodology for understanding sparse algorithm performance by considering Matricized Tensor Times Khatri-Rao Product (MTTKRP), a sparse tensor decomposition. We analyze the MTTKRP algorithm, and use hardware performance counters to identify the system bottleneck, i.e. memory latency. We then present two targeted software optimizations to improve performance by increasing memory bandwidth utilization.

2.1 Introduction

Tensors are multi-dimensional arrays which can flexibly represent data relations. Decomposing a tensor into a function of low-rank components can identify interesting and useful information. As such, the use of tensors has become popular in machine learning, social network analysis, data mining, computer vision, and quantum computing [2, 31, 42, 62, 63, 93, 94, 103]. In recent years, there has been a big push to design efficient algorithms and accelerators to improve sparse tensor

algebra performance to enable these applications [44, 51, 52, 61, 69, 107–109].

Sparse tensors present significant performance challenges for current general-purpose systems [70]. Since most of the values within a tensor are zero, sparsity is naturally present in the data. Additionally, high dimensionality leads to large storage overheads, requiring more efficient data structures to represent sparse tensors. These two factors combined cause algorithms to have poor spatial and temporal locality, as useful data is spread far apart with limited opportunity for reuse. Furthermore, the sheer size of the data necessitates practically efficient algorithms to compute tensor decompositions. Finally, the algorithms may need to be specialized for different tensor representations such as Coordinate (COO), Hierarchical Coordinate (HiCOO) [69] and Compressed Sparse Fiber (CSF) [107]. Some optimizations are often specific for a given dimensionality and not may not be generally applicable.

The most exciting tensor applications make use of decompositions. A sparse tensor can be represented as a sum of rank-1 tensors (vectors) using Canonical Polyadic Decomposition (CPD) [54]. It is analogous to Singular Value Decomposition (SVD) for matrices. The most computationally intensive step in the Alternating Least Squares (ALS) algorithm for CPD is Matricized Tensor Times Khatri Rao Product (MTTKRP), which flattens the tensor along each dimension and take a column-wise Kronecker Product. CPD's application in machine learning for finding trends in datasets over a wide range of domains drives the need for improving the performance of MTTKRP.

In this work, we present a detailed performance analysis of the MTTKRP kernel to aid future development and contribute a deeper understanding of the limiting factors of recent innovations. More generally, our work provides a convenient case study exploring processor performance for a sparse workload. We

characterize leading sparse tensor frameworks as well as our own code with hardware performance counters. Interestingly, we observe that sparse tensors often perform well on general-purpose CPUs, and can often benefit from using SIMD instructions. Their performance (in terms of instruction throughput) is typically better than comparable sparse graph workloads, which often perform poorly on current CPUs, and consequently have even more room for improvement [11].

Despite the perceived limitations of current hardware for sparse tensor workloads, our analysis finds that a surprisingly few number of last-level cache (LLC) misses are the primary performance bottleneck remaining. These misses are costly in modern out-of-order processors, and can greatly impair instruction throughput. To confirm our observations and scout possible solutions, we evaluate different techniques to improve performance by utilizing more memory bandwith. We consider single-instruction multiple data (SIMD) extensions and software prefetching. Our analysis can motivate the development of novel algorithms and accelerators for sparse tensor workloads.

2.2 Background

2.2.1 Tensor Data Structures

The coordinate (COO) data structure is the simplest way of storing a sparse tensor [62]. In this format, the coordinates of every non-zero and the associated non-zero value are stored in separate arrays (Figure 2.1). The benefit of this structure is that it can easily store an arbitrary-dimensional tensor without requiring a complex lookup to find the required data. Since it stores the indices of all of the non-zeros in the tensors, this structure may consume a large amount of space. Additional optimized data structures like Heirarchical Coordinate (HiCOO), Com-

pressed Sparse Column (CSC), and Compressed Sparse Fiber (CSF) compress this structure even further.

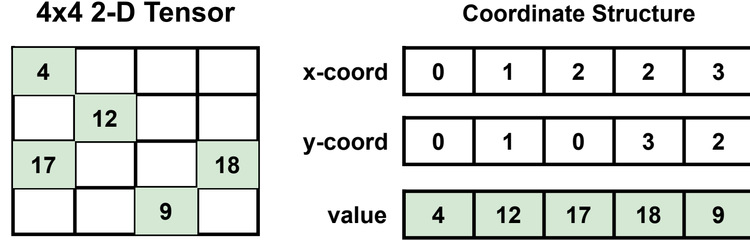

Figure 2.1: A 2-D Tensor stored in the Coordinate (COO) format. The x and y-coordinates of each non-zero are stored in separate arrays with the associated non-zero value.

2.2.2 Matricized Tensor Times Khatri Rao Product (MTTKRP)

MTTKRP is the main computational kernel in tensor decompositions. In this section we present a simple MTTKRP implementation and explore potential performance bottlenecks (Algorithm 1).

Notation	Description
$\mathcal{X}$	Tensor
M	Modes (dimensions)
$\mathcal{X}.nz$	Set of non-zeros in $\mathcal{X}$
R	Number of columns in factor matrices
n_i	i^{th} coordinate of non-zero $n \in \mathcal{X}.nz$

Table 2.1: Summary of notation used.

Consider a 3D tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and factor matrices $B \in \mathbb{R}^{J \times R}$ and $C \in \mathbb{R}^{K \times R}$. Let the output of MTTKRP be the updated factor matrix $A \in \mathbb{R}^{I \times R}$, such that $A = \mathcal{X} \times B \bigodot C$. In practice, factor matrices are stored in a row-major format.

Algorithm 1 MTTKRP Algorithm based on PASTA [70]

1: **function** MTTKRP($\mathcal{X}$, B, C)
2:     **for all** $n \in \mathcal{X}.nz$ **do**
3:         $x \leftarrow n_x, y \leftarrow n_y, z \leftarrow n_z$
4:         **for all** $r = 0...R-1$ **do**
5:             $A[x \times R + r] += n \times B[y \times R + r] \times C[z \times R + r]$
6:     **return** A

We analyze Algorithm 1 line-by-line considering potential memory access locality. Since the coordinate representation stores the coordinates for each non-zero in a separate array, Line 3 results in consecutive accesses that enjoy great spatial locality. These coordinates are then used to index specific rows in each factor matrix to perform the multiplication.

Various compression techniques used to avoid storing zeros in sparse tensors may result in poor spatial and temporal locality while accessing different rows of the factor matrices via indirection (Line 5 in Algorithm 1). The similarity of values of $\{x, y, z\} \times R + r$ among subsequent accesses depends on the density of non-zeros along those dimensions. There is potential for tensors to have vastly differing densities of non-zero values along each dimension. This also raises concerns about load imbalance affecting parallelizability, as each dimension's work is done independently. Thus, we expect Line 5 in Algorithm 1 to be a performance bottleneck. This step is especially inefficient since we compute each of the corresponding row indices right before accessing memory.

2.3 Experimental Setup

We perform our experiments on a dual-socket Intel Xeon Platinum 8260 with 768 GB of RAM. Each socket has 24 physical cores running at 2.40 GHz and a 35.75 MB shared L3 cache. For this analysis, we perform experiments using

only a single thread to better focus on the performance implications of the core microarchitecture. We collect hardware performance counter results using `perf`, and we report DRAM traffic measured at the memory controller, as memory bandwidth is a common bottleneck for data-intensive workloads.

2.3.1 Sparse Tensor Frameworks

For our analysis, we select three of the fastest tensor algebra frameworks for comparison. PASTA is a parallel benchmark suite for sparse tensor algorithms with reference implementations for different tensor operations in different representation formats [70]. SPLATT is an optimized library for sparse tensor applications, which implements a novel sparse tensor data structure and other optimizations like re-ordering transformations and cache tiling [106]. ALTO uses highly tuned compiler intrinsics to utilize ISA extensions to improve performance [52].

As a baseline, we implement our own version of PASTA's MTTKRP algorithm in C++ with small modifications such as replacing C-style arrays with STL vectors and fixing the traversal order along a single mode. Our baseline performs similarly and in some cases, better than PASTA (Figure 2.7). Furthermore, when we optimize, writing our own implementation ensures that any performance differences are due solely to intentional algorithmic interventions as opposed to implementation details.

2.3.2 Tensor Dataset

We use a variety of tensors to capture various workload properties. We use all of the input tensors from the FROSTT dataset [105] except Patents and Reddit-2015 (segfaults for multiple frameworks) and Matrix Multiplications (small size). To help generalize the results across a number of tensors, we broadly characterize

each tensor based on the number of non-zeros as small (< 6 M), medium (6 M – 80 M) or large (> 80 M). This allows us to observe the effect of tensor size on performance.

Tensor	Description	Non-zeros (M)	Dimensions
Amazon Reviews	Product reviews on Amazon	1741.81	3
Chicago Crime	Crime reports in Chicago	5.33	4
Delicious	Webpage tags by user	140.13	4
Enron Emails	List of words sent in emails	54.20	4
Flickr	Image tags by user	112.89	4
LBNL-Network	Anonymized network traffic	1.70	5
NELL-1	Language learner database	143.60	3
NELL-2	Language learner database	76.88	3
NIPS Publications	List of published papers	3.10	4
Uber Pickups	Location data of Uber requests	3.31	4
VAST 2015 Mini-Challenge 1	Synthetic dataset	26.02	5

Table 2.2: Summary of the properties of input tensors taken from FROSTT [105].

2.4 Performance Analysis

Modern general-purpose CPUs are generally thought to be a poor fit for sparse, irregular computations such as those required for tensor decompositions. However, we routinely observe instruction per cycle (IPC) rates above 1 for the tensors in FROSTT, and in most cases, between 1.5 and 3 (Figure 2.2). Radical IPC improvements are unlikely on a 4-wide machine like the one we use. In this section, we discern that only a few LLC misses are responsible for the lowest IPCs. In the next section, we describe software solutions to improve performance by increasing memory bandwidth utilization.

We find that optimized tensor codes across all frameworks perform decently

well on our modern CPU. Some of the smallest tensors achieve IPCs greater than 2. For some of the largest tensors (worst performing), we observe an IPC around 1 and 2.5–10 Misses Per Kilo Instruction (MPKI) (Figure 2.2). As the LLC MPKI increases (higher cache miss rate), the IPC lowers, and this is apparent across all of the frameworks. The worst case LLC MPKI is roughly 20, which is not as frequent as one might expect for a large sparse workload. These executions are not limited by memory bandwidth, as we observe generally low bandwidth utilization ($\approx$ 20% on average). Thus, memory latency must be the problem. Retrieving data from memory stalls the pipeline for a significant amount of cycles, worsening performance. Unsurprisingly, we find that larger tensors result in higher MPKI and worse performance across all the frameworks (Figure 2.2). The SPLATT framework has a substantially lower IPC, because it uses a more sophisticated algorithm that is more irregular.

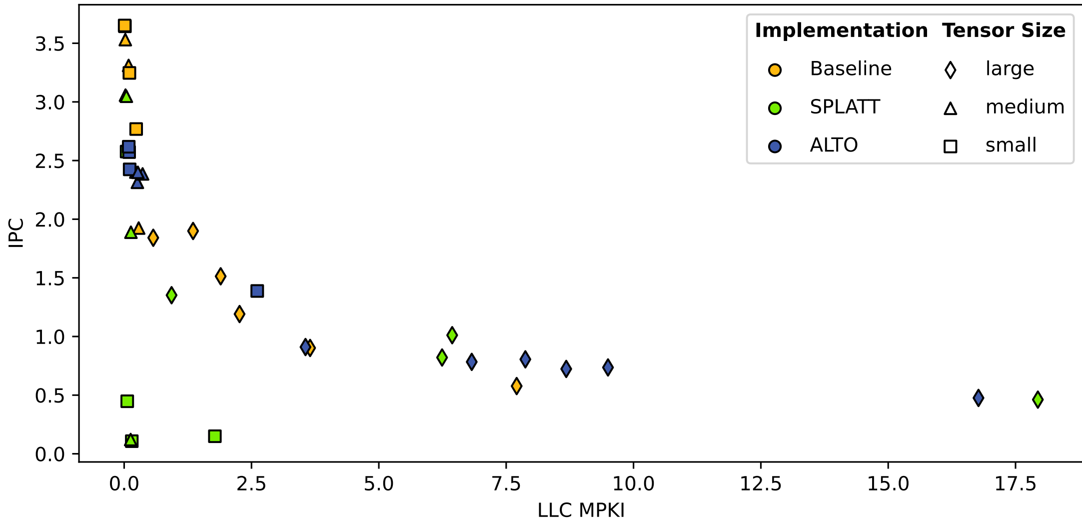

Figure 2.2: IPC vs MPKI for MTTKRP in different frameworks run on various tensors in the FROSTT dataset. Higher MPKI results in low IPCs, suggesting that LLC misses are a problem. Low MPKI numbers, coupled with low observed bandwidth utilization suggests that only a few LLC misses are quite problematic.

High sparsity leads to memory efficiency challenges. While the coordinate for-

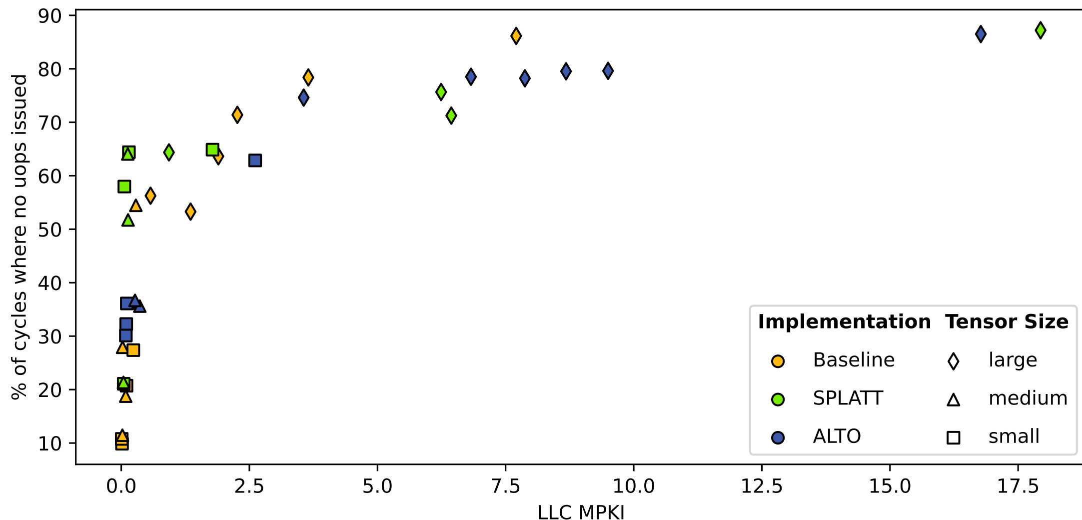

Figure 2.3: Fraction of cycles without any new instructions issued vs MPKI for MTTKRP in different frameworks run on various tensors in the FROSTT dataset. Higher MPKI results in more stalls as the processor has to wait for data to arrive from main memory before it can start processing it. We observe that even for a very low MPKI of 5, the processor spends 80% of cycles without issuing any instructions. This means that the performance of current out-of-order processors is highly sensitive to even a few LLC misses.

mat alleviates some of the locality concerns by compactly storing the coordinates in an array and accessing them linearly, the algorithm still requires M random array accesses for each non-zero value to retrieve the elements for the factor matrix product. Computing the row indices one by one before accessing the array spaces out these loads, further exacerbating the problem (Figure 2.4). A single LLC miss can stall the pipeline for a large number of cycles. From our measurements, we observe that an LLC MPKI of 5 corresponds to an average of roughly 80% of cycles in which no micro-ops are issued (Figure 2.3).

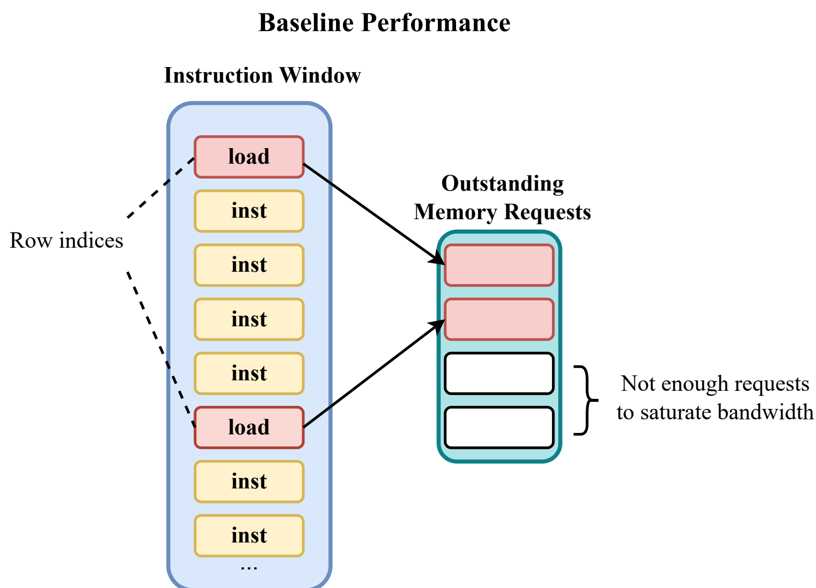

Figure 2.4: Poor load miss density of the baseline algorithm. Since generating the address requires additional computation, loads are spaced far apart and the resulting number of memory requests is much smaller than the memory system allows. There is potential to improve performance by amortizing the request overhead by getting more misses in flight.

2.5 Improving Performance

As memory latency seems to be the biggest bottleneck for sparse tensor workloads, there are two options to make these algorithms run more efficiently: reduce the number of cache misses or do useful work during a miss. The miss rate depends on the layout of the tensor and the traversal order (we desire a higher density of non-zeros to achieve more hits in the factor matrices). Prior work has reduced memory traffic (and LLC misses) by various optimizations like tiling and reordering computations to "densify" each dimension [108] and reducing the number of array accesses through compiler intrinsics [52]. Another potential innovation is to create a heuristic to determine the best traversal order for a specific input tensor, but that may require a large amount of preprocessing.

Getting More Misses in Flight

Per Little's Law, *Parallelism* = *Throughput* × *Latency.* Assuming latency (average memory access time) is constant, we can increase throughput by improving parallelism. This queueing theory result can be put into practice by increasing the number of outstanding memory requests. Currently, the memory bandwidth is underutilized as there are too few instructions that cause a cache miss in the instruction window (Figure 2.4). However, the few instructions that do cause cache misses cause lengthy stalls. Thus, the processor is executing instructions at a slow rate (due to stalls) while also greatly underutilizing memory bandwidth. SIMD instructions and software prefetching are two common ways to get more memory misses outstanding and thus improve bandwidth utilization.

SIMD

A SIMD memory instruction operates on more addresses at a time, increasing the potential amount of cache misses in the instruction window. MTTKRP is amenable to vectorization since it performs multiplication on columns of data for each non-zero. We find that GCC autovectorizes code when using the flags `-O3 -march=native -mtune=native`. This transformation is fairly performant and we observe speedups between 1.21× - 2.11× (geomean: 1.44×) over our baseline. We also use OpenMP's SIMD directive, resulting in speedups between 1.03× - 2.48× (geomean: 1.51×) over the baseline. Lastly, we achieve the highest performance gains over the baseline from packed SIMD by manually using Intel's intrinsics (1.37× - 2.81×, geomean: 1.88×).

SIMD provides the biggest benefit when the tensor is sufficiently small to fit within the cache because the vector units in the processor are not waiting on data from main memory and can be fully saturated (Figures 2.5 & 4.1). Even though

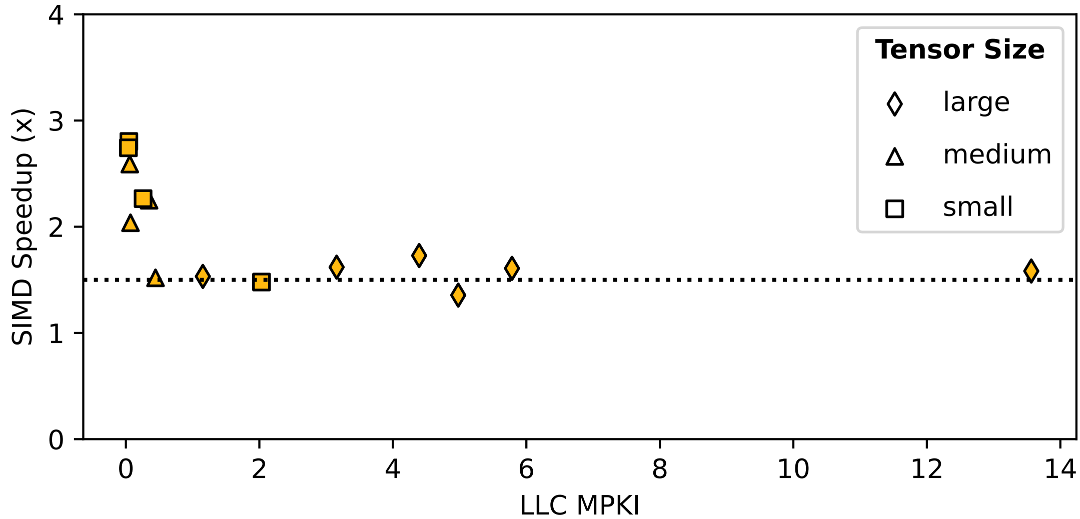

Figure 2.5: Speedup vs MPKI for SIMD using manual intrinsics for various tensors. Initially, when the tensor is small enough to fit within cache (low MPKI), SIMD speedup is high due to better resource utilization. As the tensors get bigger, the processor is stalled for significant cycles waiting on data from main memory and the speedup starts to plateau. However, for larger tensors, the speedup is still roughly 1.5× over the baseline.

SIMD benefits the larger tensors as well, by increasing memory-level parallelism (MLP), we do not observe as big of a performance boost. While bandwidth utilization is increased over the baseline (Figure 4.1), the processor is still stalled for a significant number of cycles waiting on data from main memory. We observe this in Figure 4.1, where we can see that larger tensors with a higher MPKI do not enjoy as big of a boost. However, even in the best case, we observe 54% bandwidth utilization of the single-core limit. We identify this as an opportunity for further improvement via hardware acceleration.

Software Prefetching

MTTKRP has a lot of indirect memory accesses (Line 5 in Algorithm 1), which are difficult for hardware prefetchers. Since this access pattern is easy to compute

in software due to the coordinate data structure, we leverage software prefetching to manually load future useful rows in the factor matrices from memory. We use the compiler intrinsic `__builtin_prefetch` to perform this operation. Memory prefetching bypasses the instruction window entirely and does not stall the pipeline, allowing us to transfer additional useful data during compute by taking advantage of reserve bandwidth (Figure 4.1). We apply software prefetching to our baseline and measure the performance improvement (Figure 4.1). Software prefetching results in speedups of $1.16-2.20\times$ (geomean: $1.53\times$) over the baseline.

Unlike SIMD, we observe the biggest performance improvements over the baseline using software prefetching when the tensors have many non-zeros (Figure 2.7). These tensors are too large to fit in cache, and transferring useful data earlier helps reduce the communication latency. Since both of our optimizations work best on different tensors, we combine both optimizations in a unified implementation which has good overall performance (Figures 4.1 & 2.7).

2.5.1 Comparison Against Leading Frameworks

We compare the performance of our optimizations against leading tensor frameworks (Figure 2.7).

Of the existing frameworks, we find that ALTO has the leading performance due to its very efficient bitmap-based storage structure which allows for fast indexing, fewer memory transactions, and better locality. Additionally, ALTO uses OpenMP's SIMD directive to vectorize their code. Our software optimizations (SIMD, Software Prefetching and SIMD+Software Prefetching) also perform well compared to our baseline and PASTA, and in some cases, even outperform ALTO (Flickr-3D, Amazon). Combining SIMD and Software Prefetching results in the best performance on larger tensors, while SIMD has large speedups on smaller

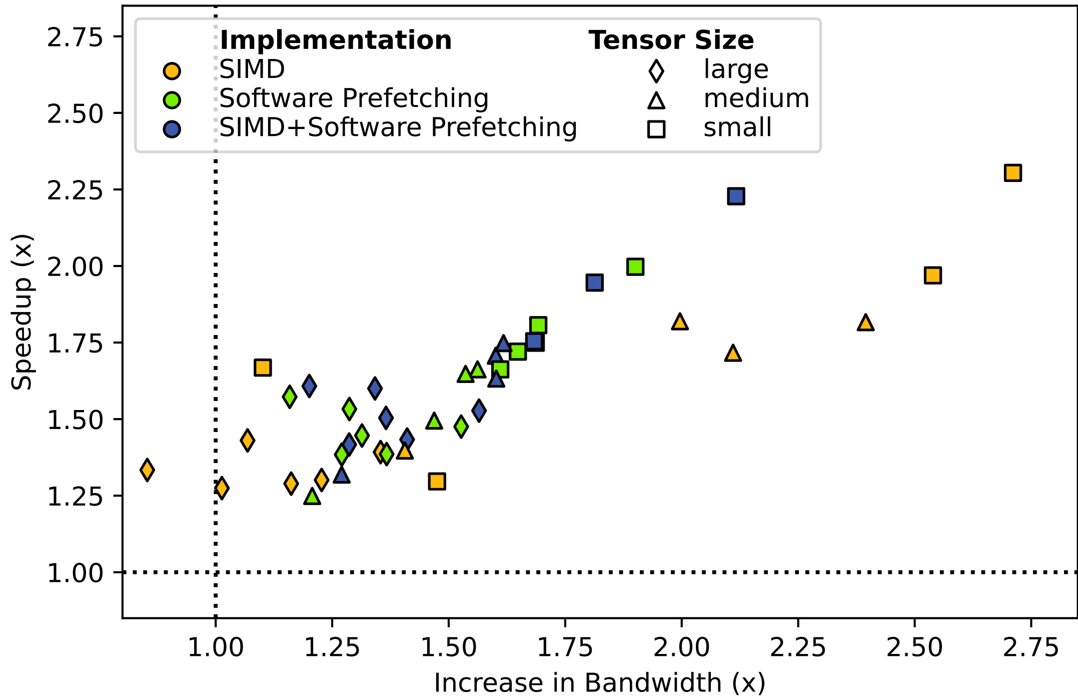

Figure 2.6: Speedup vs improvement in bandwidth utilization over the baseline for SIMD, Software Prefetching and combining both. Higher is better for both axes. Increasing bandwidth utilization leads to better performance (higher speedups) for MTTKRP.

tensors. We believe ALTO would also benefit from using manual intrinsics for SIMD and software prefetching. SPLATT typically has the worst performance between the different implementations.

2.6 Related Works

Due to the rising popularity of data analytics, there have been several recent works modeling and improving the performance of sparse tensor workloads on current hardware. Li et. al. present PASTA [69], a sparse tensor benchmark suite including multiple different tensor kernels to aid application developers in evaluating different systems as well as performance of new optimizations. They

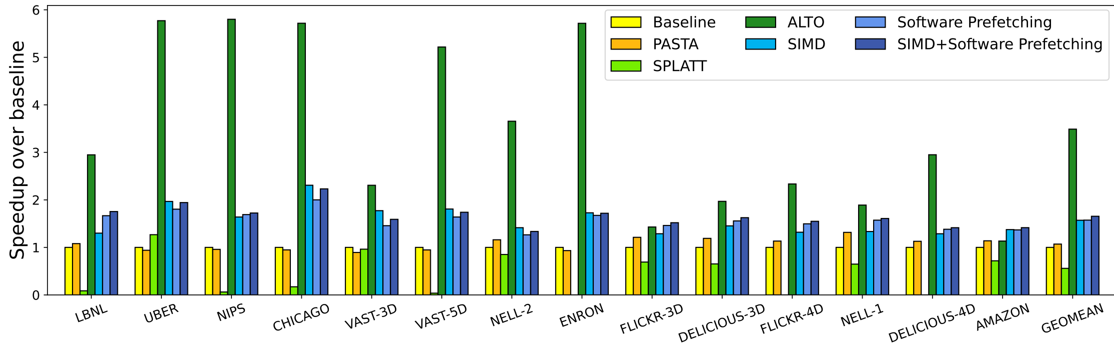

Figure 2.7: Speedups over our baseline for existing frameworks and our SIMD and Software prefetching implementations. Tensors are ordered according from smallest runtime (fastest) to largest runtime (slowest) for the baseline implementation. We present only kernel runtimes, however, both SPLATT and ALTO require additional preprocessing, which if considered, result in runtimes longer than our own implementations. SPLATT is unable to decompose certain larger dimensional tensors and is generally the slowest framework.

also present a novel storage format for sparse tensors that improves data locality while preserving generality with respect to dimensions [71]. Smith et. al. present SPLATT, a fast, parallel implementation for 3-D tensors [108]. Their innovations include cache-friendly reordering of the tensor and cache tiling. Helal et. al. present ALTO, a bit-encoded linearization of a COO tensor using bit-wise scatter and gather operations, resulting in significantly fewer memory accesses for indexing non-zeros [52]. Nguyen et. al. apply a similarized linear format on GPUs. Baskaran et. al. present multiple optimizations for processing tensors at large scale, reducing memory consumption and synchronization [7]. Liu et. al. present F-COO, a novel sparse tensor storage format for GPUs, and a parallel MTTKRP implementation for GPUs. In addition to software optimizations, many recent works have developed novel hardware accelerators for sparse tensors [44, 51, 109].

2.7 Conclusion

We analyze the performance of different frameworks' MTTKRP implementations using performance counters. Analogous to sparse graph workloads, we find that existing algorithms result in a small number of LLC misses, which are extremely problematic on current out-of-order machines [11]. These algorithms are also vulnerable to low miss densities, further accentuating the effect of the aforementioned LLC misses. In this work, we improve performance of this kernel by increasing MLP using SIMD and software prefetching. However, we find that this kernel still suffers from significant LLC miss penalties. Thus, there is an opportunity to improve performance via hardware or software by either further increasing MLP or reducing the number of LLC misses. For example, accelerators with tensor-specific prefetchers and novel data structures which support traversal orders reducing memory access times could improve efficiency. MLP could also be increased by processing sparse tensors on massively parallel architectures like GPUs, which are designed for hiding latency. Our performance improvements using SIMD show promise for this approach.

Chapter 3

Improving Scalability of Pivoting-Based Clique Counting

3.1 Introduction

Dense subgraph counting is an important problem in network analysis with many applications in social network analysis and bioinformatics [66]. An example of a noteworthy dense subgraph is a k-clique, which has exactly k vertices, and every vertex in the subgraph is connected to each other. Counting the number of cliques in a graph is an important step in identifying pockets of density within a graph. Clique finding is a well-researched topic in the graph community and it has many interesting practical applications for community detection [41,47,50,91] and social network analysis [92, 102]. In recent years, the data mining community has incorporated clique finding into deep learning classifiers to enhance recommender systems in social networks [81, 90]. Clique finding is also used prominently in bioinformatics, and researchers have used graph models to find variants in gene sequences [82], efficiently group related genes in a database [110], and perform

protein structure analysis [5]. The rapid growth in the number of social media users and the large size of genome data has amplified the need for high performance systems capable of analyzing these huge networks to count or enumerate their k-cliques.

Clique finding represents only a single problem in Graph Pattern Mining (GPM). Motifs are more general graph patterns that also interest researchers in this space. Various GPM frameworks have been developed to count repeated instances of given patterns within the search graph. These include Sandslash [25], Pangolin [26], Peregrine [58], Arabesque [113], Fractal [36], and others. The aforementioned frameworks use more generalized algorithms and provide their own APIs for counting instances of arbitrary, user-given patterns. In contrast, there are also algorithms specialized for counting k-cliques, such as kClist [32], Pivoter [56], and Arb-Count [104]. By virtue of being custom designed to solve a specific problem, these algorithms generally perform better than the general-purpose GPM frameworks.

Clique counting is a challenging but rich problem to explore. Searching for cliques involves considering various combinations of vertices which results in a combinatorial explosion in algorithmic work. Naturally, optimizing the algorithmic cost is fruitful, but pragmatic simplifications can often lead to the fastest implementations. Depending on the size of the graph, the size of the clique being counted, and the topological properties of the input graph, different approaches may be the fastest. Thus, a high-performance implementation must carefully balance algorithmic efficiency, parallelizability, and memory time-space tradeoffs.

Leading clique counting algorithms typically contain two main phases: an ordering phase, which converts the undirected input graph into a directed acyclic graph (DAG), and a counting phase, which is dominated by recursively building

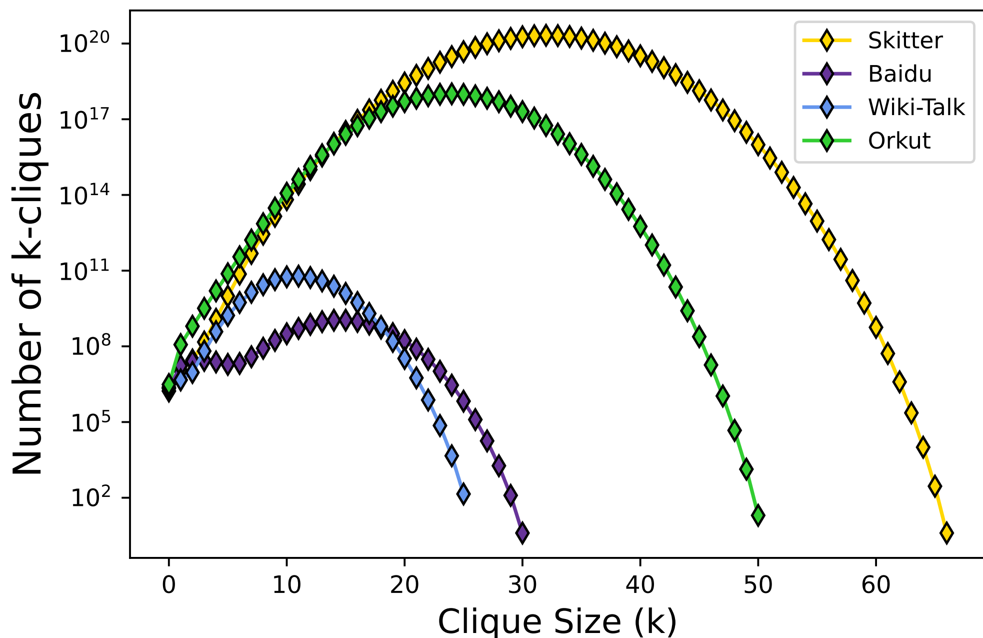

Figure 3.1: Frequency distribution of k-cliques in different graphs. Enumeration-based algorithms are exponentional with respect to clique size (k), and increasing frequency of moderately sized cliques in real-world graphs further exposes this increasing complexity.

induced subgraphs. Different algorithms approach the counting phase differently, and they can be classified as enumeration-based [32, 104] or pivoting-based [56] algorithms. Enumeration-based algorithms are generally faster for counting smaller cliques, while pivoting-based algorithms are faster for larger cliques. The presence of large cliques in large real-world networks (Figure 3.1) necessitates a combination of algorithmic efficiency and practical performance to process them.

As core counts continue to increase in modern machines, improving parallel performance becomes increasingly essential to process graphs at scale. Given that multiple factors affect the performance of clique counting, we take a holistic approach to improve its parallel scalability and overall performance on CPUs. We analyze both the ordering and counting phases of the current state-of-the-art

pivoting-based clique counting algorithm Pivoter [56], to understand how these factors impact performance, and present targeted optimizations. In this work, we present *ComSpark*, a fast and scalable exact clique counting algorithm, and make the following contributions:

- We explore tradeoffs between the quality of the ordering and the time to produce the ordering. We consider a traditional core ordering, a parallel degree ordering, a parallel core ordering approximation, and a novel parallel centrality-based ordering. Our parallel core ordering approximation performs well and typically produces the same maximum out degree as the original core ordering. Our novel centrality-based ordering replicates the characteristics of a core ordering while being easy to compute.

- We observe that different orderings result in better counting times for different graph topologies due to a work-locality tradeoff. We present our analysis of this tradeoff, and provide a heuristic which quickly identifies when a particular ordering will be beneficial and computes that ordering during the ordering phase.

- To increase the parallel scalability of the counting phase, we present two subgraph data structures which greatly reduce the memory usage while still allowing for fast access to neighbor lists.

- Combining our heuristic and the optimizations in the ordering and counting phases, we achieve the first scalable clique counting pivoting-based algorithm on CPUs.

We evaluate ComSpark on a suite of real-world graphs and compare it to Pivoter and other prior work. We demonstrate that ComSpark achieves near-linear parallel scaling up through 64 threads for the entire clique counting process.

We also compare ComSpark against a GPU implementation of pivoting (GPU-Pivot). We show that ComSpark achieves better scalability as clique size (k) increases, allowing it to outcompete the GPU while counting larger cliques.

3.2 Background

3.2.1 Preliminaries

For a given undirected input graph G, we want to count the number of cliques present of a given size k. A *clique* is a completely connected subgraph, i.e. each vertex is directly connected to every other vertex in the subgraph. The input graph G consists of a vertex set, *V(G)*, and an edge set, *E(G)*. For a k-clique C present in the input graph, *V(C)* $\in$ *V(G)*, *E(C)* $\in$ *E(G)* and $|V(C)| = k$. Each vertex u in G has a neighborhood, which is the set of the vertices with which u shares an edge. The neighborhood of u in G is indicated by *N(u)* and its size is the degree $d(u) = |N(u)|$. We treat G as an undirected graph, so $v \in N(u) \implies u \in N(v)$.

To reduce the amount of work done when counting cliques, efficient algorithms transform G into a DAG, which we denote by $\vec{G}$. This step adds work to direct edges to remove cycles in order to avoid counting the same clique multiple times. Consequently, any vertex u in the new DAG can have two types of neighbors: in-neighbors and out-neighbors. In-neighbors are those vertices in *N(u)* for which u is the destination vertex of the shared edge. Conversely, *u's* out-neighbors are those vertices in *N(u)* where u is the source vertex of the edge. While counting cliques, we only consider the out-neighbors of a vertex u in $\vec{G}$, denoted by $\vec{N(u)}$. The number of out-neighbors of a vertex u is called its out-degree and is written as $\vec{d(u)}$.

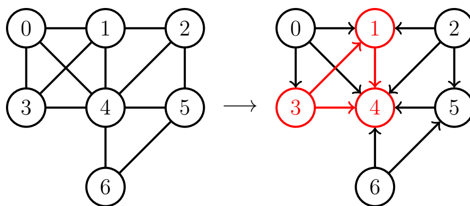

Figure 3.2: Converting an undirected input graph (left) to directed acyclic graph (right) by a degree-based ordering. Furthermore, the highlighted (red) portion on the right indicates the subgraph induced by vertex 0.

Given a total ordering ω, *directionalizing* transforms the graph G to $\vec{G}$ by removing the edge $v \to u$ from $E(G)$ if $\omega(v) \geq \omega(u)$ and keeping only the edge $u \to v$ in $E(\vec{G})$. Edges are thus directed from a lower ω to a higher ω vertex. Pivoter uses a core/degeneracy ordering, which guarantees the lowest maximum out-degree (core value) of a vertex in $\vec{G}$. While this approach reduces the amount of work done while counting cliques, it requires a fair bit of effort to compute, and cannot be parallelized since it requires removing vertices in the graph one by one [83]. More recently, efforts have been made to approximate the core ordering and improve parallelism by removing multiple vertices with degrees under a certain threshold [14]. Alternatively, a degree ordering compares vertices by degree and uses the identifier as a tiebreaker (example in Figure 3.2). Computing a degree ordering is easy to parallelize. We describe the degree ordering (ω) as:

$$\omega(u) > \omega(v) \quad \text{if} \quad (d(\vec{u}) > d(\vec{v})) \vee (d(\vec{u}) = d(\vec{v})) \wedge (u > v)$$

A major step in counting cliques is building a vertex-induced subgraph. This subgraph represents the current scope of the graph being explored, which could potentially be a clique. The induced subgraph contains the vertices in the neighborhood of the target vertex and any edges between them. The induced sub-

graph does not include the target vertex itself. The highlighted portion in Figure 3.2 shows the vertex 0 induced subgraph in the example graph. We denote the subgraph induced by vertex u on $\vec{G}$ to be $\vec{g_u}$ with $V(\vec{g_u}) = N(\vec{u})$ and $E(\vec{g_u}) = \{(v_1, v_2)|(v_1, v_2) \in E(\vec{G}) \wedge v_1 \in N(\vec{u}) \wedge v_2 \in N(\vec{u})\}$. Building an induced subgraph enables efficiently checking connectivity between vertices.

3.2.2 Enumeration-Based K-Clique Counting

Clique counting algorithms like kClist [32] and Arb-Count [104] directionalize the graph and recursively build induced subgraphs to count cliques. Subgraphs are built by intersecting adjacency lists to find common neighbors to include in the subsequent recursion levels. In each recursion level, these algorithms build a subgraph for every vertex in the graph at that level. This strategy potentially results in redundant work (Figure 3.4) and becomes more expensive as the depth of the call stack increases (for larger values of k). Since these algorithms explicitly process every vertex in every level, they are known as enumeration-based algorithms.

3.2.3 Pivoting-Based K-Clique Counting

In recent years, more work has been done to improve the practical efficiency of clique counting algorithms through pruning away extra edges from the search space. Jain and Seshadhri present Pivoter [56], the leading pivoting-based algorithm for k-clique counting. Pivoter uses pivoting to count maximal cliques using the Bron-Kerbosch algorithm [17, 39, 114]. Once a maximal clique of size n has been found, the number of k-cliques present in the maximal clique can be calculated by the formula $\binom{n}{k}$. The work savings come from being able to count cliques by judiciously selecting which subgraphs to build in each level based on a

"pivot" vertex. This strategy results in a deeper call stack to find maximal cliques, albeit with less branching (Figure 3.4). Pivoter is able to count instances of 100-cliques and larger in real-world networks. In contrast, leading enumeration-based algorithms work well on k up to 8 (Table 3.6).

Pivoter first converts the input graph into a DAG using a core ordering which can be generated in linear time. The algorithm for computing the core ordering from Matula and Beck [83] guarantees the smallest maximum out-degree in the graph. This is significant, because the maximum degree determines the execution time of the counting phase and a smaller maximum degree typically means less algorithmic work (i.e. a shallower call stack) and thus a more efficient algorithm.

Algorithm 2 Core Ordering

```
1:  function OrdCore(G)
2:      r ← 0
3:      ω ← [0, 0, ..., 0]                          ▷ Array to hold order of each node
4:      while r ≠ |V(G)| do
5:          Sort nodes by degree
6:          r ← r + 1
7:          minDegVertex ← first element from sorted list
8:          ω[minDegVertex] ← r
9:          V(G) ← V(G) \ minDegVertex
10:         Update degrees of all v ∈ N(minDegVertex)
11:     return ω
12: Build Graph G⃗ using ω                          ▷ Add edge (u,v) if ω[u] < ω[v]
```

Once the ordering of each vertex is computed (Algorithm 2), the graph is directionalized and edges $u \to v$ are only included if and only if $\omega(u) < \omega(v)$. Producing the core ordering is a sequential operation since it requires removing a

single vertex from the graph in each iteration.

After the DAG is generated, Pivoter starts counting the cliques (Algorithm 3). It keeps track of 3 sets: X (vertices to be excluded from recursion), P (candidate vertices for recursion) and R (vertices forming the current clique) (Figure 3.3). Every vertex is always in exactly one set. The three sets are stored together as a single of array of size $|V(G)|$ with pointers denoting the boundaries between the sets. This allows for reuse while efficiently moving vertices within different sets between recursion levels via swaps. In addition to the sets, Pivoter uses a $|V(G)|$-sized array lookup to map vertex identifiers to their index within the set array.

After initializing the sets, Pivoter recursively builds subgraphs, adds the inducing vertices to R and picks the highest degree vertex in P to be the pivot. Instead of building a new subgraph at each recursion level, Pivoter reuses and mutates the current subgraph since the next subgraph is a subset of it. Pivoter only moves the edges present in the next-level subgraph to the front of the neighbor list. To build a maximal clique, the current clique can be extended by adding the pivot vertex or the non-neighbors of the pivot to R. Pivoter avoids recursing over the neighbors of the pivot, resulting in significant work savings (Figure 3.4). Since the original DAG is not being modified any further (only the subgraph is modified), each top-level vertex can be processed in parallel.

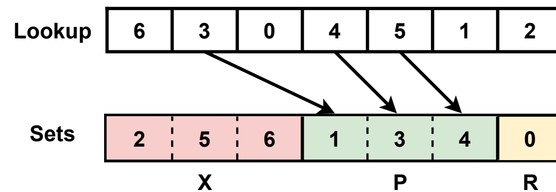

Figure 3.3: Array for sets X, P and R and a dense lookup for the location of vertices within the set while processing vertex 0 in the example DAG (Figure 3.2). Storing the sets in a single array allows for efficiently transferring vertices between X, P and R in between recursion levels.

Algorithm 3 Pivoter Algorithm for Counting k-cliques

```
1: function RecursiveCounting(g⃗_v, P, R, X)
2:     if P = {} then                                   ▷ Discovered maximal clique
3:         return (|R| choose k)
4:     else
5:         pivot ← {p ∈ V(g⃗_v) | d(p) = max d(p)}
6:         for all w ∈ {pivot} ∪ {P \ N(pivot)} do
7:             Move w from P to R
8:             Build subgraph g⃗_w
9:             RecursiveCounting(g⃗_w, P, R, X)
10:            Move w from R to X
11: function Counting(G, k)
12:     count ← 0
13:     G⃗ ← OrdCore(G)                                  ▷ Directionalize input graph
14:     for all v ∈ V(G⃗) in parallel do
15:         P ← N(v)
16:         R ← {v}
17:         X ← {}
18:         Build subgraph g⃗_v
19:         count += RecursiveCounting(g⃗_v, P, R, X)
20:     Return count
```

The worst case execution time for Pivoter is $O(c(G)^2|SCT(\vec{G})|+m+n)$, where n is the number of vertices, m is the number of edges, and *c(G)* is the core value, i.e. the maximum out-degree generated by ordering the vertices and SCT is the search space with complexity $|SCT(\vec{G})| = O(n3^{c(G)/3})$. Since the complexity of the algorithm does not depend on the size of the clique (k), large cliques can

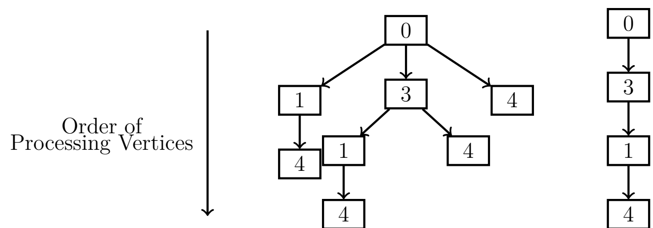

Figure 3.4: Differences in the function call heirarchy between enumeration-based algorithms (left) and Pivoter (right) for counting 4-cliques associated with vertex 0 in the example DAG from Figure 3.2. The number in the box denotes the vertex for which the induced subgraph is being built in that level. Vertices are processed sequentially in a depth-first manner. In the heirarchy on the left, the sequence $1 \rightarrow 4$ is processed multiple times. In contrast, the stack for Pivoter is much smaller and avoids redundant computation in the counting phase.

be counted efficiently using Pivoter. In contrast, the complexity of the leading enumeration-based algorithm Arb-Count is $O(m\alpha^{k-2})$, where α is the arboricity of the graph, making it less efficient for counting larger cliques.

Currently, the only parallel optimized implementation of pivoting-based clique counting algorithms is GPU-Pivot, presented by Almasri et al. [4]. To the best of our knowledge, no literature currently exists for optimizing Pivoter for CPUs. In GPU-Pivot, a vertex or an edge is assigned to a block of threads (warp). However, due to the nature of the pivoting algorithm, opportunities for additional parallelization among the threads in a warp are limited. GPU-Pivot partitions groups of threads into blocks. The threads in these blocks sequentially process the same sequence of vertices during the counting phase. Only the process of finding the pivot vertex is done in parallel within a block. This makes fully utilizing the parallel capabilities of GPU hardware very difficult and impacts scaling on large dense graphs with many cliques.

GPU-Pivot stores the entire adjacency matrix in a binary format to reduce memory consumption and performs intersections of neighbor lists using binary operations. Additionally, GPU-Pivot builds a new subgraph per recursive level.

This results in additional intersection operations to build the subgraph. In contrast, we use a CSR-like structure to store the subgraph. We also avoid building a new subgraph per-level and are able to reuse our first-level induced subgraph by storing the relevant neighbors for a particular level at the beginning of the neighbor list.

3.3 Parallelizing the Ordering Phase

3.3.1 Parallel Core-Approximation

When scaling to larger target clique sizes, limiting the maximum out-degree in the directionalized graph is essential for performance. The counting effort required per vertex is superlinear with respect to degree, so limiting the maximum out-degree during the ordering phase can greatly reduce the amount of algorithmic work in the subsequent counting phase. For this reason, Pivoter employs a core ordering which guarantees the lowest maximum out-degree. Even though computing a core ordering is sequential, for moderate or larger clique sizes, the time it saves in the counting phase typically outweighs the time a fast parallel ordering (to create a different directionalized graph) would save in the ordering phase (Table 3.2).

However, a sequential ordering can still take up a significant fraction of the overall time (Table 3.2). Thus, we attempt to accelerate the ordering phase to improve the parallelizability of the entire algorithm. We take inspiration from an existing parallel approximation presented by Besta et. al. [14] used for graph coloring. While the core ordering algorithm used in Pivoter removes a single vertex (with the least degree) in the graph at a time, the approximation algorithm relaxes the lowest degree condition by removing multiple vertices from the graph in every

round if their degree is less than $(1 + \epsilon) \times \delta$, where ϵ is an error parameter and δ is the average degree of remaining the graph. It then updates the degrees of the neighbors of the removed vertices in parallel, before sorting the remaining vertices in the graph by their new degrees.

Algorithm 4 Parallel Core Ordering Approximation

```
 1: function ApproxCoreOrder(G, ε)
 2:     while r ≠ |V(G)| do
 3:         remove ← {}                          ▷ Vertices to be removed in round
 4:         δ ← |E| / |V|                        ▷ Compute average degree
 5:         for all u ∈ V(G) do
 6:             if d(u) < (1 + ε) × δ then
 7:                 remove.append(u)
 8:                 V(G) ← V(G) − u              ▷ Remove vertices
 9:                 ω[u] ← r                     ▷ Assign rank to vertex
10:                 r ← r + 1
11:         for all u ∈ remove in parallel do
12:             for all v ∈ N(u) do
13:                 d(v) ← d(v) − 1              ▷ Update degrees
14:         Sort nodes by degree
15:     return ω
16: Build Graph G⃗ using ω                        ▷ Add edge (u,v) if ω[u] < ω[v]
```

Since the time spent in the counting phase typically depends on the maximum out-degree produced by the ordering, we use the maximum out-degree to determine the quality of an ordering. A larger ϵ implies that more vertices are removed in the initial rounds, increasing the amount of parallelism but lowering the quality of the ordering. If ϵ is set high enough, the ordering produced by this algorithm

results in a degree-based ordering. In contrast, a smaller ϵ results in fewer vertices removed in each round, reducing the amount of parallelism but producing a higher quality ordering. A small enough ϵ results in an approximation of the core ordering.

An error parameter (ϵ) of *0.1* is considered to be a good compromise as it removes enough vertices to effectively parallelize each round and requires a relatively small number of rounds, while producing a low enough maximum out-degree [14]. We emphasize that the goal of Besta et. al.'s work was to produce an ordering which results in good graph coloring performance. In contrast, we use the ordering to reduce the amount of work for counting cliques. In our experiments, we sweep various values for the error parameter ϵ and find that ordering quality and counting performance is typically best when the maximum out-degree is equal to that produced by the core ordering (Figure 3.10). We are able to achieve this by setting ϵ to -0.5 on the graphs we test. This allows us to remove a large fraction of low-degree vertices in the initial rounds in parallel, and produce the same maximum out-degree as the core ordering. At the other end of the spectrum, setting ϵ to a very large number (50000) effectively results in a degree ordering.

Prior work has been to optimize ordering for improving the counting phase performance. Li. et. al. present various color-based ordering algorithms for reducing the amount of search paths while counting k-cliques [72]. We emphasize that their algorithms are sequential and targeted towards enumeration-based clique counting methods, which suffer from redundant work. Our parallel orderings are targeted towards pivoting-based clique counting, which already prunes redundant search paths by selecting a pivot vertex (Figure 3.4).

Parallel Optimizations

In addition to a parallel core ordering approximation, we present further optimizations to improve performance. In the first round of ordering, we remove a large fraction of vertices in the graph (up to 75%). This creates a large amount of parallel work to update the degrees of the neighbors of the removed vertices. We normally use atomic instructions to ensure correctness while modifying the degrees in parallel. We refer to this as a push-based update (Algorithm 5). High-degree vertices which have many low-degree neighbors being removed can suffer from contention as their degrees need to be decremented multiple times (steps 11-13 in Algorithm 4). To avoid contention when a large number of vertices are being removed, we develop a pull-based algorithm where each vertex in the graph checks how many of its neighbors are being removed before updating its degree locally (Algorithm 6). On the suite of input graphs we test, we find that this optimization results in a 1.04× speedup over only using the pull-based algorithm in each level. This is especially beneficial for extremely large graphs like Friendster, where over 10 million vertices are removed in the first round (0.74s faster, 1.1× speedup). Additionally, we use parallel sort in C++17 to sort the remaining vertices in the graph by their degrees (step 14 in Algorithm 4).

Algorithm 5 Push-based degree update

1: **for all** $u \in remove$ in parallel **do**
2:     **for all** $v \in N(u)$ **do**
3:         atomic_fetch_add($d(v)$, -1)

Algorithm 6 Pull-based degree update

```
1: for all u ∈ |V| in parallel do
2:     if u ∉ remove then
3:         for all v ∈ N(u) do
4:             if v ∈ removed then
5:                 d(u) ← d(u) − 1                    ▷ No atomics required
```

Despite our efforts to parallelize core ordering due to its work savings in the counting phase, we find that degree ordering results in faster total execution times for some graphs (Figure 3.12), and we explore this in Section 3.6.

3.3.2 Centrality-Based Ordering

As a cheaper alternative to the compute-intensive parallel core-approximation, we aim to understand the functionality of the core ordering and replicate its effect with a simpler ordering. We first differentiate the core ordering with the degree-based ordering to better understand its characteristics.

Consider the undirected input graph in Figure 3.5. A degree-based ordering would direct the edge $1 \rightarrow 0$ since *d(0) > d(1)*. In the core ordering, vertices *4*, *5* and *6* would be removed in early rounds since they have the lowest degrees. Vertex *0* is the next vertex removed since it now has the lowest degree in the remaining graph. Since vertex *1* is removed later than vertex *0*, it has a higher rank and the core ordering directs the edge $0 \rightarrow 1$. By peeling the lowest degree vertex at a time, the core ordering process implicitly ranks the vertices by their *importance*. In this context, importance is defined as high-degree vertices which are also connected to other high-degree vertices. Various centrality measures can compute this effect at a fraction of the cost of the parallel core-approximation. Since most of vertex *0*'s neighbors have low degrees, it is not more *important* than

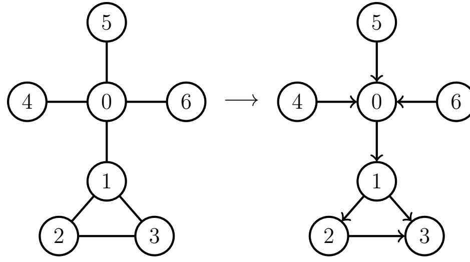

Figure 3.5: Converting the undirected input graph (left) to a directed acyclic graph (right) by a core ordering. Even though vertex *0* has the highest degree, most of its neighbors are low-degree vertices. Hence it has a lower degree than vertex 1 after peeling vertices 4, 5, and 6, and the edge is directed from $0 \rightarrow 1$.

vertex 1, which has higher-degree neighbors. Centrality measures like Eigenvector centrality can quickly identify *important* vertices in the graph [16].

We use the Eigenvector centrality to compute a relative score for each vertex, and direct edges from vertices with lower scores to higher scores. The pseudo-code for our ordering algorithm is as follows:

Algorithm 7 Centrality-based Ordering

1: **function** OrdCentrality(G)
2: $\omega \leftarrow [\frac{1}{|V|}, \frac{1}{|V|}, ..., \frac{1}{|V|}]$ ▷ Rank of each vertex
3: **for all** i until max_iters **do**
4: **for all** $u \in V$ in parallel **do**
5: contributions[u] = $\omega[u]$
6: **for all** $u \in V$ in parallel **do**
7: sum ← 0
8: **for all** $v \in N(u)$ **do**
9: sum ← sum + contributions[v]
10: ω[u] = sum
11: **return** ω
12: Build Graph $\vec{G}$ using ω ▷ Add edge (u,v) if $\omega[u] < \omega[v]$

The centrality-based ordering is very simple to compute compared to the parallel core-approximation, since it only requires summing up the scores of each vertices' neighbor's scores. This can be done very efficiently in parallel. With a small number of iterations (3), the centrality-based ordering produces a maximum out-degree that lies between that of the core ordering and degree-based ordering (Figure 3.10). While the centrality-based ordering never results in the fastest performance, it is always faster than the slower ordering between core and degree.(Figure 3.13). This shows that a successful ordering should not only consider the degrees of the neighbors, but their *importance* as well.

3.4 Maximum Neighbor Influence

In our analysis, we find that different orderings perform better during the counting phase for different types of graph topologies (Table 3.2). The fastest overall execution time (includes ordering and counting time) is typically achieved by the core approximation with $\epsilon = -0.5$ or the degree-based ordering (Figure 3.13). This differs from the previously thought notion that core ordering is always better for pivoting. Although it is conceivable that a faster to compute ordering might result in a lower total time than the core ordering, it is surprising it is faster specifically in the counting phase. Our analysis reveals that counting with the core ordering executes fewer instructions, but counting with the degree ordering executes instructions faster (Section 3.6). In other words, the core ordering does have the expected algorithmic advantage, but the degree ordering has a practical speed advantage. Thus, which ordering is faster depends on whether the core ordering saves a sufficient amount of algorithmic work to overcome the degree ordering's speed advantage.

One factor determining the difference in performance for the two orderings is

the different degree distributions once the graph has been directionalized. High degree vertices connected to higher degree vertices result in a large maximum out-degree in the degree-based ordering. Social networks are assortative, and such a cluster is common [87]. If some of the higher degree vertices have many low-degree neighbors, the core ordering ranks that vertex relatively lower and it will have some directed edges going outward. We find that core ordering results in vertices with lower maximum out-degrees than that of degree ordering (Figure 3.6).

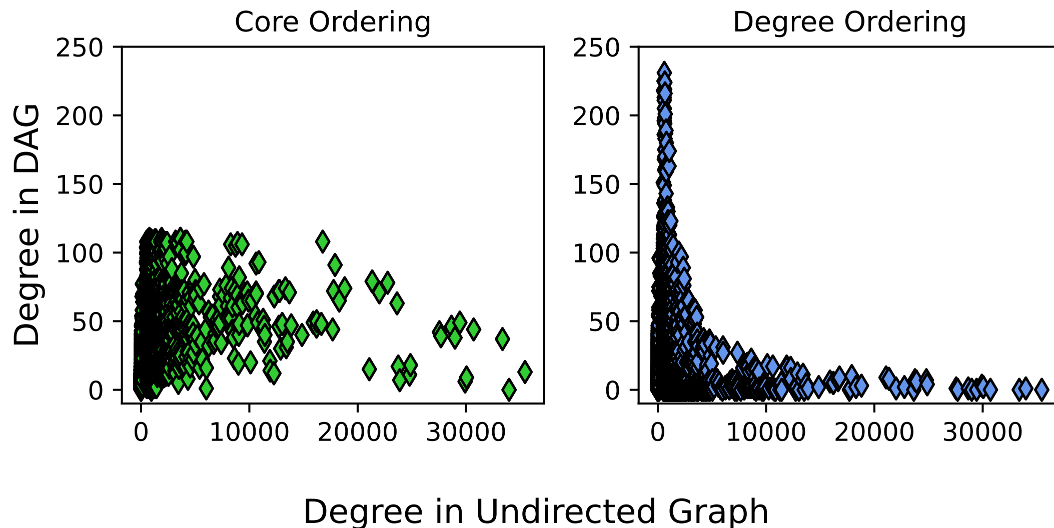

Figure 3.6: Differences in degree distribution after producing a DAG using core (left) and degree ordering (right) on the Skitter graph.

3.4.1 Work-Locality Tradeoff

We next analyze why the degree ordering sometimes counts faster despite both orderings using the same counting implementation. Our analysis reveals that counting with a degree ordering executes instructions faster due to fewer cache misses (Section 3.6). Both orderings start from the same graph, and the resulting DAGs have the same average degree, but their degree distributions can differ (Figure 3.6). In practice, the degree ordering does not achieve as low of a

maximum out degree as the core approximation, but those extra edges reduce the degrees elsewhere.

The amount of algorithmic work required for a particular vertex to count cliques is determined by the size of the first-level subgraph induced by that vertex ($g_{\vec{v}}$ in step 18 of Algorithm 3). The lower out-degrees from the core ordering are attractive because they result in smaller subgraphs, and consequently, fewer recursive function calls (step 9 in Algorithm 3). The work savings are magnified when the graph has a lot of cliques, i.e. the number of recursive calls is high due to the large number of edges in the subgraph. While core ordering always results in less algorithmic work than a degree-based ordering (Figure 3.14), we still find that degree ordering results in marginally faster counting time on certain graphs. When the number of cliques is not high, the amount of work is only slightly more for degree ordering. The larger subgraphs formed from high out-degree vertices from the degree ordering result in better cache locality since the first-level subgraph is reused in subsequent recursion levels, leading to better counting performance (Figure 3.15).

Since the ordering has a significant impact on the overall execution time of the algorithm, it is imperative to select the proper ordering for the best performance. Our analysis shows that either the core-approximation or the degree-based ordering results in the best overall performance (Figure 3.13). The number of cliques in the graph determines whether the reduced algorithmic work from the core ordering is more beneficial for performance than the increased locality from larger subgraphs due to the degree ordering. We present a heuristic to determine which ordering will perform better by predicting whether the graph contains a large number of cliques. On the graphs we analyze, we observe that there are many cliques when the most influential vertex (vertex with the highest degree) in the

graph is connected to another influential vertex. When the highest-degree neighbor of the highest-degree vertex has a sufficiently high degree, there tends to be a large number of common neighbors between the two vertices, forming densely connected pockets in the graph with many cliques. This follows the assortative property of social networks.

We measure the influence of the highest-degree neighbor of the most influential vertex by calculating what fraction of edges in the undirected graph it is a part of. We refer to this fraction as *maximum neighbor influence*. We predict whether the graph will have many cliques or not by multiplying the maximum neighbor influence with the average degree of the graph. This value can be computed quickly in linear time. Based on our analysis of several real-world graphs, we predict that if the value is above *0.15*, there are enough cliques in the graph for the work savings from core ordering to result in better performance. If the value is below *0.15*, we predict that there are not enough cliques in the graph and the locality benefits and the computational efficiency of degree ordering impact total performance more. To add robustness to our heuristic, we also test the assortativity of the graph by measuring the number of common neighbors between the most influential vertex and its highest-degree neighbor. We find that over 10% of the neighbors are common between the two vertices when there are many cliques. We also find that degree ordering is more advantageous for smaller graphs ($|V| < 1M$), where ordering is a significant fraction of the total time (Table 3.5). Based on these values, we select the proper ordering between the core-approximation and the degree-based ordering.

For the example undirected graph from Figure 3.2, the most influential vertex is vertex *4*, since it has the highest degree. It's highest-degree neighbor is vertex *1* with degree 4. Thus, the maximum neighbor influence for this graph is $\frac{d(v)}{|E|}$

$= \frac{4}{12} = 0.33$. The result of multiplying the maximum neighbor influence with the average degree (1.71) is *0.56*. Based on our heuristic, we select the core-approximation ordering in the ordering phase. We also observe that 75% of vertex *1*'s neighbors are common with vertex *4*, increasing the confidence of our heuristic.

3.5 Improving Counting Phase Scalability

While we focus on selecting the best ordering for the counting phase, it is still important to optimize counting phase performance since it is the longest phase of the entire clique counting process. We base our counting implementation on Pivoter since its superior algorithmic complexity enables it to count large cliques in large real-world graphs. Since the counting phase does not modify the DAG (the mutations are to induced subgraphs), recursively processing each vertex can be done in parallel. A vertex-parallel strategy brings up two potential concerns: load imbalance due to the skewed degree distributions of real-world graphs and increased memory requirements for large thread-local structures for storing the induced subgraph.

To analyze the impact of load imbalance, we both attempt to improve load balance as well as measure the amount of time spent doing work by each thread. We sweep various scheduling parameters such as task granularity (chunk sizes) and scheduler types (static, dynamic, cyclic), and we are not able to fully improve parallel scalability. Additonally, we measure the time required for each thread during the entire counting phase while running on 64 threads. The coefficient of variance (the ratio of standard deviation to the mean) for the execution time of each thread across the entire suite of input graphs is 0.03. Through our analysis, we find that load balance is at most a minor factor, and memory usage hinders scalability to a much greater extent. Thus, we optimize the induced subgraph

data structure to reduce memory consumption.

3.5.1 Reducing Memory Consumption

The induced subgraph data structure is critical to counting cliques. While processing a vertex, the first level induced subgraph is built and then modified in subsequent recursion levels to count all the cliques originating from that vertex. Pivoter stores the induced subgraph and associated structures per thread since different threads are working on different root vertices and thus different induced subgraphs. In the original Pivoter code, the subgraph is represented as an adjacency list, with an array of size $|V(G)|$ pointing to inner arrays that store the neighbors of the associated vertices (Figure 3.7). The whole structure is effectively a 2D array. Using an array of size $|V(G)|$ allows for fast access to neighbor lists within the subgraph by simply using the vertex identifier as an index. This subgraph structure is allocated and initialized locally per-thread, including the $|V(G)|$-sized index pointing to inner arrays (for neighbor lists). With minor adjustments to memory allocation, we observe that such a dense structure is sufficient to result in good parallel performance for most applications on contemporary hardware (Figure 3.20). We refer to this approach as *ComSpark (dense)*.

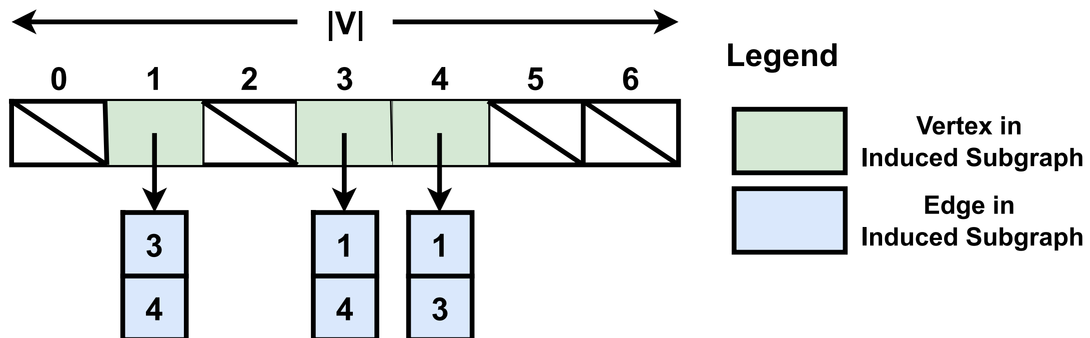

Figure 3.7: Dense structure for storing the neighbor lists of the first-level induced subgraph.

For large graphs, storing a $|V(G)|$-sized index array per thread may not be feasible. Note that if the number of threads is greater than the average degree of the graph, these indices alone will consume more memory than the original graph. To further compress the subgraph and improve locality, we only index the vertices with non-zero degree in the subgraph (Figure 3.8). Since the number of non-zero degree vertices in the subgraph is at worst $c(G)$ (which is on the order of 100s, as opposed to $|V(G)|$, which is on the order of millions), this may even allow the entire subgraph to fit in cache. We use a hash map with vertex identifiers as keys and a pointer to its neighbor list in the new sparse structure as values. We refer to this approach as *ComSpark (sparse)*. For large graphs like Friendster, this optimization is able to overcome the scaling plateau from 32 threads to 64 threads.

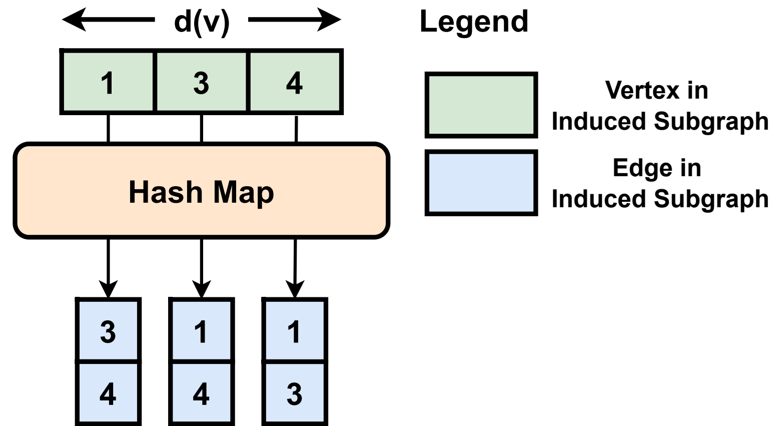

Figure 3.8: Sparse structure in ComSpark for storing neighbor lists in the first-level subgraph. Instead of a dense array to point to the locations of the neighbor lists, the sparse structure uses a hash map with vertex identifiers as keys and pointers to the neighbor lists as values.

In our experiments, we typically observe a query into a hash map to be $\approx 1.2\times$ slower than an array lookup. To combine the memory efficiency of a compressed subgraph structure with the benefit of fast indexing, we present a new subgraph structure. Instead of using a hash map, we remap the vertex identifiers in the

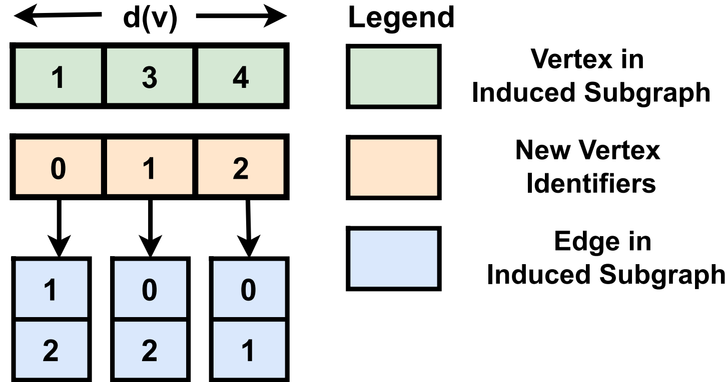

Figure 3.9: Optimized structure with remapping in ComSpark for storing neighbor lists in the first-level subgraph. Only the $d(\vec{v})$-sized remapped array (orange) and the neighbor lists are stored in memory.

first level subgraph to range from 0-$d(\vec{v})$, where v is the vertex currently being processed (Figure 3.9). We only do this remapping step in the first level of the recursion, and reuse the new identifiers in subsequent levels. Since the working set of the subgraph reduces in every recursion level, storing a $d(\vec{u})$-sized array per-thread is still fairly memory-efficient. We refer to this approach as *ComSpark (remap)*.

We find the remapped subgraphs to have the best combination of speed and memory usage, and we explore their tradeoffs relative to sparse and dense in the evaluation. The best ComSpark design uses the remapped subgraph structure with a core approximation or degree-based ordering determined by our heuristic.

3.6 Evaluation

In this evaluation section, we explore and analyze the various design tradeoffs to build ComSpark and compare it against prior work. We first analyze the tradeoffs for the ordering phase and validate our order-selecting heuristic. We then evaluate the scalability of the counting phase and our overall implementation. We

Graph	Description	|V| (M)	|E| (M)	$\vec{d}$	k_{max}
DBLP	Citation network	0.32	1.05	2.5	114
Soc-Pokec	Social network	1.6	22.3	13.7	29
As-Skitter	Internet topology	1.7	11.1	6.5	67
Baidu	Links between web pages	2.1	17.8	8.5	31
Wiki-Talk	Network of Wikipedia users	2.4	9.3	3.9	26
Orkut	Social network	3.1	117.2	38.2	51
LiveJournal	Social network	4.8	70.0	8.1	-
Web-Edu	Links between .edu web pages	9.85	46.23	19.1	449
Friendster	Social network	65.6	1,806.1	27.5	129

Table 3.1: Summary of the properties of input graphs used in the Evaluation. All graphs are unweighted and symmetrized prior to analysis. These graphs are taken from a variety of sources [33, 67, 101]. We only use Soc-Pokec for evaluating our heuristic, and LiveJournal for comparison against the GPU.

conclude by scaling the clique size and comparing against prior work.

3.6.1 Experimental Setup

We perform our experiments on a single-socket AMD EPYC 9554 (Genoa). The machine has 64 physical cores running at 3.1 GHz and 256 MB of shared L3 cache. Our system has 768 GB of RAM. For the experiments, we use 64 threads unless specified otherwise. We compile with g++ (version 12.2.0) with optimization -O3 and use OpenMP.

We use a variety of input graphs to evaluate the performance of our optimizations (Table 3.1). These graphs are taken from different sources [33, 67, 101]. Since clique finding is used heavily in social network analysis, we select graphs commonly used in this subfield to make the analysis more consistent with prior work. All graphs are unweighted and symmetrized to initially be undirected.

3.6.2 Accelerating the Ordering Phase

To appreciate the impact of the ordering phase on the overall execution time, we first consider the prior ordering approaches: core ordering and degree ordering (Table 3.2). Although the ordering phase is usually only a modest portion of the overall execution, it can have a significant impact because it can greatly affect the counting phase.

Graph	Core Ordering				Degree Ordering			
	Ordering Time (s) (1 thread)	Counting Time (s) (64 threads)	Total Time (s) (64 threads)	Max. Out-Degree	Ordering Time(s) (64 threads)	Counting Time (s) (64 threads)	Total Time (s) (64 threads)	Max. Out-Degree
DBLP	0.04	0.04	0.08	113	0.00	0.03	**0.03**	113
As-Skitter	0.37	0.95	**1.32**	111	0.01	3.20	3.21	231
Baidu	0.68	0.27	0.94	78	0.01	0.26	**0.27**	298
Wiki-Talk	0.17	1.33	**1.50**	131	0.01	4.10	4.11	340
Orkut	3.62	34.18	**37.80**	253	0.05	43.60	43.64	535
Web-Edu	1.35	2.27	**3.62**	448	0.02	4.84	4.86	448
Friendster	111.09	73.95	185.04	304	1.89	72.64	**74.53**	868

Table 3.2: Comparison between time taken to convert the input graph into a DAG using the sequential core and degree orderings and the associated counting times for counting 8-cliques. The fastest overall times are bolded. The core ordering is guaranteed to produce the lowest maximum out-degree, which typically reduces the work in the counting phase.

We observe that in most cases, the core ordering results in faster counting times. This is due to its increased algorithmic efficiency in the counting phase, stemming from the low maximum out-degree it produces. For DBLP, Baidu and Friendster, we observe that a degree ordering actually results in marginally faster counting time while being significantly faster in the ordering phase (Table 3.2).

We also observe that the time required to produce a sequential core ordering can be a significant fraction of the total runtime on larger graphs. To reduce the ordering phase execution time, we use Besta et. al's parallel core approximation algorithm to increase parallelism by removing a larger fraction of vertices at a time. We measure the time required to direct the input graph using a core ordering, the parallel core-approximation ordering with different values of the error parameter (ϵ), a degree-based ordering, and our centrality-based ordering. We also compare the associated maximum out-degrees, counting times, and the combined times for both phases (Figures 3.10, 3.11, 3.12, 3.13).

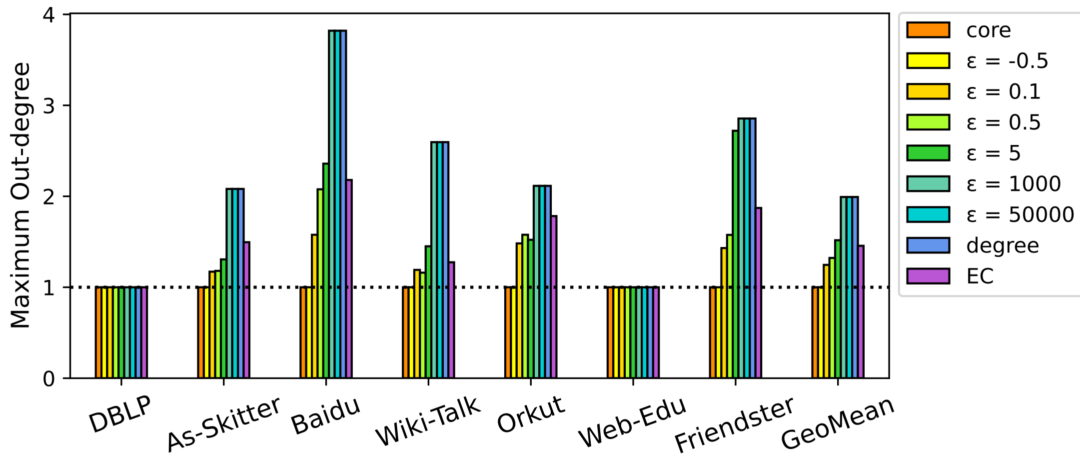

Figure 3.10: Maximum out-degrees produced by changing the error parameter in the parallel approximation algorithms. The maximum out-degrees for core, degree and centrality-based orderings are also included as a reference. As ϵ increases, and more vertices are removed in each round, the maximum out-degree produced by the ordering increases.

Graph	$\epsilon = -0.5$	$\epsilon = 0.1$	$\epsilon = 50000$
DBLP	165	7	1
As-Skitter	525	9	1
Baidu	1347	10	1
Wiki-Talk	499	8	1
Orkut	5936	10	1
Web-Edu	160	8	1
Friendster	6033	15	1

Table 3.3: The number of rounds required to order the graph for different values of ϵ. Setting ϵ to -0.5 results in the same maximum out-degree as the core ordering. The number of rounds in this case is still significantly less than the |V| rounds (on the order of millions) required by the sequential core ordering algorithm. Setting ϵ to 50000 effectively results in a degree-based ordering since only one round is required. $\epsilon = 0.1$ is a good compromise between parallelism (number of rounds) and ordering quality (Figure 3.10).

Unsurprisingly, we observe that the quality of the ordering decreases as we increase the value of ϵ. By setting ϵ to a very low value (-0.5), we are able to generate the same maximum out-degree as the sequential core ordering. Even for a low ϵ, more than 50% of vertices in the graph are removed in the first round. Removing so many low degree vertices can be done very quickly in parallel; and this does not affect the quality of the ordering. Increasing the value of ϵ decreases the number of rounds, and makes the ordering even faster to compute (Table 3.3). Since more vertices are removed in each round, the quality of the ordering suffers (Figure 3.10). At the other end of the spectrum, setting ϵ to a very large value (50000) removes all of the vertices in the graph in one round, effectively resulting in a degree-based ordering. By ranking vertices based on importance, the centrality ordering (EC) produces a reasonable quality ordering that lies in between that of the core and degree orderings.

There are some graphs (DBLP, Web-Edu) where the ordering quality remains the same for all values of ϵ. We find that the highest degree vertex in the DAG is connected to vertices with higher degrees and higher *importance* in the original

undirected graph, since its out-going edges do not change across different orderings. This implies the presence of a large maximal clique, which we confirm in Table 3.1. Further analysis of the DAG topologies show very similar degree distributions, but the degree-based ordering still results in marginally more higher degree vertices.

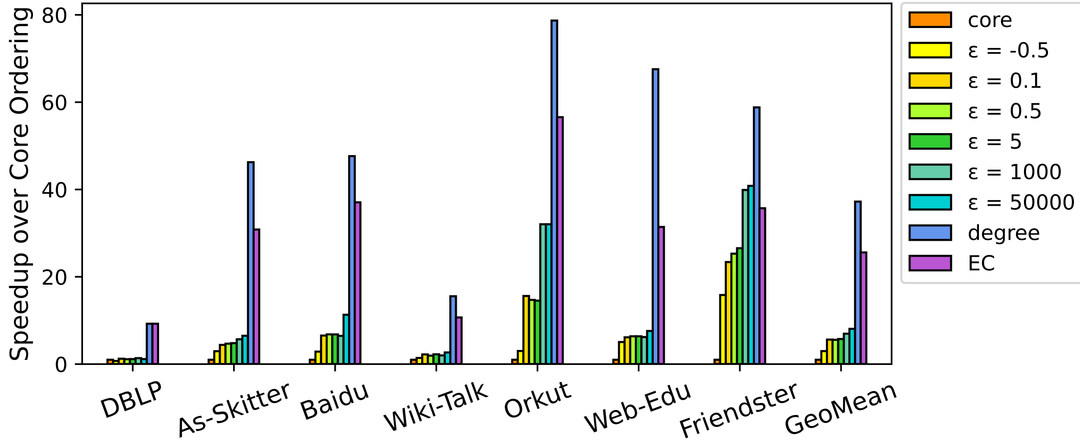

Figure 3.11: Comparison between time to produce various orderings. On larger graphs, our approximation is significantly faster. In addition to being fast, our approximation with $\epsilon = -0.5$ produces the same maximum out-degree as core ordering. This results in the counting phase time between both to be comparable. Degree ordering is always the fastest ordering, but it does not always result in the best counting times.

On average, we observe a 4.55× speedup over our exact sequential core ordering using our approximation, while producing the same maximum out-degree when $\epsilon = -0.5$. Our approximation results in larger speedups especially on larger graphs (Figure 3.11). Since a core ordering generally results in the fastest counting times, using the approximation with $\epsilon = -0.5$ allows us to enjoy the benefit of algorithmic efficiency in the counting phase, while being able to produce the ordering itself much faster in parallel. The degree and centrality-based orderings are easy to compute and very parallizable. Thus, they are much faster to generate than the core ordering or its approximation, even with a large ϵ in most cases. This also

shows that *importance* can quickly be approximated.

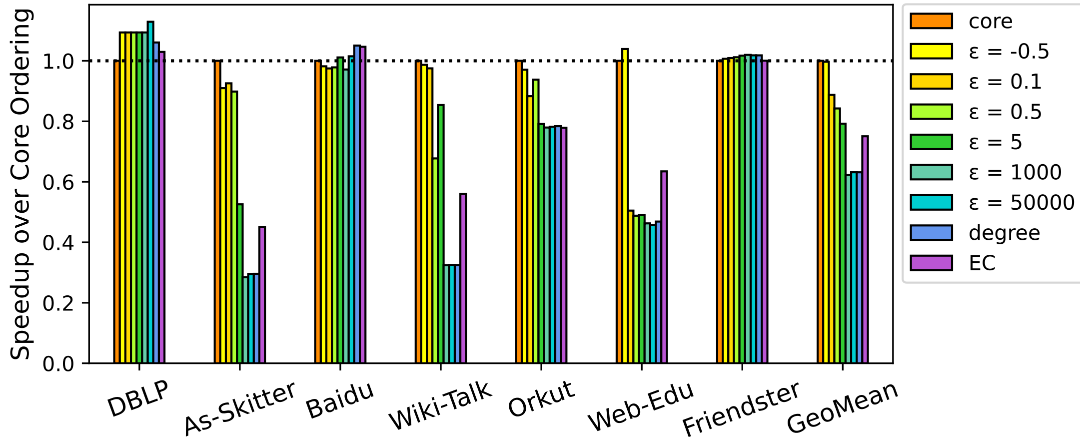

Figure 3.12: Comparison between the time to count 8-cliques for each ordering. The core ordering and our parallel approximation generally result in the best counting times due to its algorithmic efficiency. Graphs like DBLP, Baidu and Friendster benefit more from a degree ordering.

We observe that the core ordering typically results in the best counting times (Figure 3.12). The low maximum out-degrees it produces results in smaller subgraphs, and less work to build those subgraphs and check connectivity between the vertices in the subgraph. While the other orderings (core-approximation, degree-based, centrality-based) are faster to compute, the resulting DAG topology results in more work in the counting phase due to the higher maximum out-degree. However, graphs like DBLP, Baidu and Friendster have comparable counting phase performance across the different orderings.

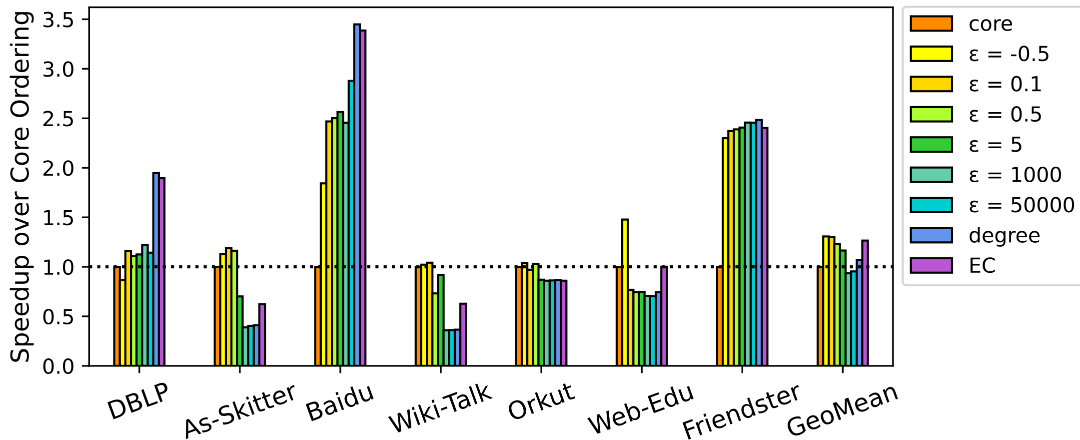

Figure 3.13: Comparison between the total execution times for counting 8-cliques using each ordering. In graphs where core ordering results in the fastest counting, our parallel approximation with $\epsilon = -0.5$ results in much faster overall time due to the fast, parallel ordering.

Combining the ordering and counting phase times, we also measure the total time required to count 8-cliques using each ordering (Figure 3.13). In graphs where core ordering is advantageous, we observe that the parallel approximation with $\epsilon = -0.5$ results in the best overall times since the ordering is faster to compute. Surprisingly, graphs like DBLP, Baidu and Friendster have better overall performance with the degree-based ordering, despite the ordering quality being inferior in some cases (Section 3.4.1). The centrality-based ordering is faster than the degree-based ordering on graphs where the core ordering is advantageous, and vice versa. However, it is never the fastest ordering. The degree distributions of the DAG produced by the core-approximation and the centrality-based orderings lie between that of the DAG produced by the core and degree-based orderings. Depending on the value of ϵ selected, we can tune the ordering towards core or degree.

3.6.3 Work-Locality Tradeoff and Maximum Neighbor Influence

To investigate why degree ordering results in better counting performance on certain graphs, we use hardware performance counters to profile the counting phases of both orderings (Table 3.4).

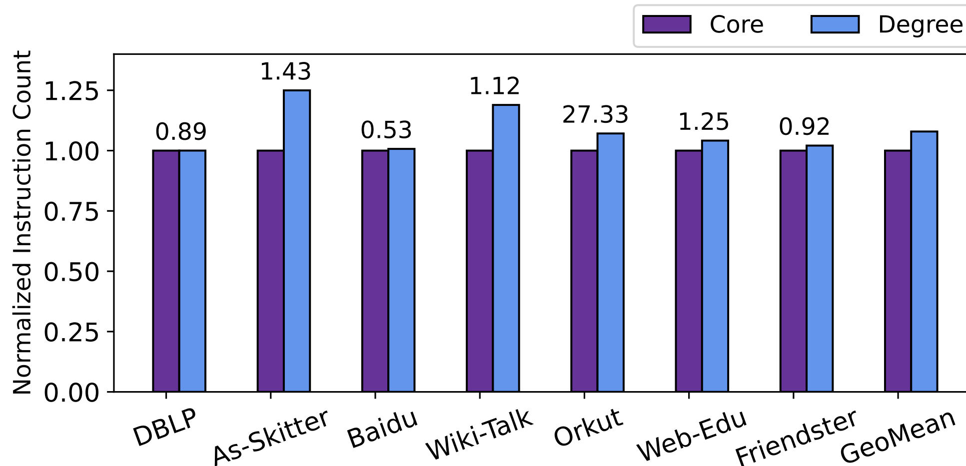

Figure 3.14: Differences in the number of instructions executed during the counting phase while counting 8-cliques using both orderings. Lower is better. The instruction count is normalized to that of core ordering. The number of million instructions per-vertex for core ordering is denoted at the top of each bar. Even though degree ordering results in marginally faster counting on some graphs, core ordering is always more algorithmically efficient as fewer instructions are always executed.

We observe that the core ordering always results in fewer instructions executed (Figure 3.14). This is expected since the maximum out-degree produced by the core ordering is lower than that of the degree-based ordering. A lower maximum degree means smaller induced subgraphs, and consequently, fewer instructions to build those smaller subgraphs. Larger, denser subgraphs generally lead to more recursive function calls and increased reuse, i.e. locality. To compare the amount of reuse and consequent locality, we compare the average number of recursions

per-vertex and the last-level cache misses per kilo-instruction (LLC MPKI) for both orderings (Figure 3.15). We measure the last-level cache misses using `perf`. For each graph, we normalize the values for both orderings by those for core ordering.

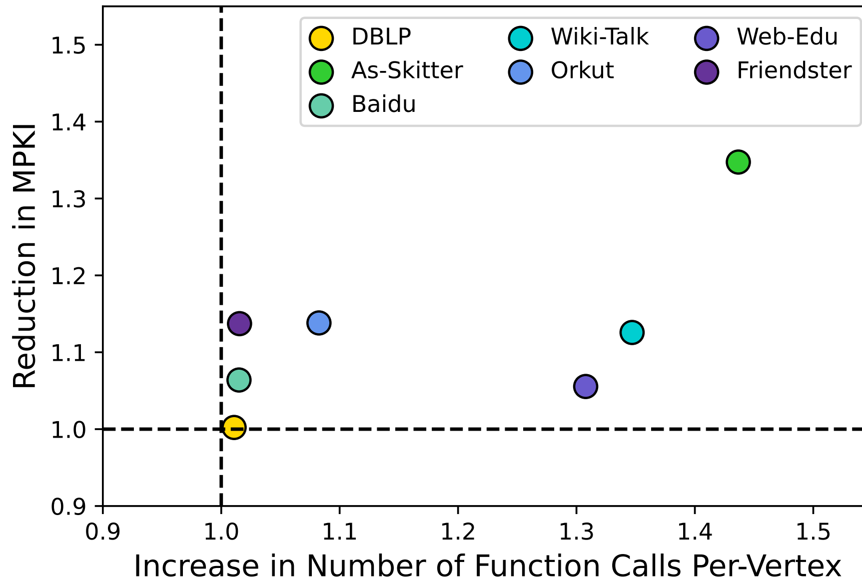

Figure 3.15: Ratio of the number of function calls per-vertex and MPKI between core and degree ordering in the counting phase while counting cliques using our dense structure. Since the maximum out-degree of degree ordering is typically higher, degree ordering results in more instructions than core ordering while counting cliques. More function calls lead to more cache locality since the first-level subgraph is reused in subsequent recursive calls. Degree ordering has a lower MPKI than core ordering for every graph.

We observe that degree-based ordering results in more recursive function calls per-vertex than core ordering for all the graphs (Figure 3.15). This is expected, since the first-level subgraphs for the degree-based ordering are larger. We also observe that the LLC MPKI for the degree-based ordering is always less than that for core ordering. Thus, we are able to confirm that the larger subgraphs for degree-based ordering enjoy more locality.

Graph	Normalized Instruction Count	Normalized Function Calls	Normalized LLC MPKI	Normalized IPC
DBLP	1.00	1.01	0.99	1.02
As-Skitter	1.25	1.44	0.74	1.06
Baidu	1.00	1.01	0.94	1.04
Wiki-Talk	1.19	1.35	0.89	1.01
Orkut	1.07	1.08	0.88	1.01
Web-Edu	1.04	1.31	0.95	1.01
Friendster	1.02	1.02	0.88	1.03
GeoMean	1.08	1.16	0.89	1.03

Table 3.4: Hardware performance counters for the counting phase of degree ordering normalized to core ordering. Degree ordering always executes more instructions, but executes them faster due to fewer cache misses (MPKI).

Like the number of instructions, we also observe that the amount of function calls is roughly equal across both orderings for DBLP, Baidu and Friendster. These graphs do not have as many densely connected subgraphs, resulting in fewer instructions per-vertex (Figure 3.14) and fewer cliques (Table 3.5). We find that when the amount of work for both orderings is comparable when the graphs are sparser, and the increased locality leads to slightly better counting performance. Since the degree ordering counts faster and there is no algorithmic advantage for core ordering, the degree ordering is faster. This demonstrates a lightweight ordering can produce the fastest overall time, so our our heuristic will need to consider using a degree ordering.

We summarize the relevant hardware performance counter data for both orderings in Table 3.4. Core ordering is more algorithmically efficient by executing fewer instructions, and degree ordering has a practical speed advantage by executing its instructions at a faster rate. Our heuristic can quickly compute whether a given graph will benefit from algorithmic efficiency or increased locality and faster ordering. We summarize various metrics related to our heuristic in Table 3.5.

We observe that our heuristic selects the correct ordering for almost all of the

Graph	**Best Ordering**	**a**	**Average Degree (δ)**	**aδ**	**Common Fraction**	**Heuristic Time (s)**
DBLP	degree	0.03	3.67	0.11	0.72	0.00
Soc-Pokec	degree	0.01	13.66	0.08	0.18	0.00
As-Skitter	core	0.31	6.54	2.03	0.84	0.01
Baidu	degree	0.02	7.94	0.13	0.00	0.01
Wiki-Talk	core	0.22	3.88	0.85	0.11	0.01
Orkut	core	0.03	37.81	1.13	0.12	0.01
LiveJournal	core	0.00	8.59	0.04	0.20	0.01
Web-Edu	core	0.04	4.67	0.19	0.90	0.04
Friendster	degree	0.00	27.53	0.00	0.00	0.24

Table 3.5: Order-selecting heuristic inputs, measurements, and decisions for counting 8-cliques. Our heuristic selects our core approximation $a\delta > 0.15$, or if there are more than 0.10 common neighbors. We select a degree ordering otherwise, or if the graph is very small ($|V| < 1M$). Our heuristic always selects the correct ordering for these graphs. The time to compute the heuristic is tiny.

graphs (Table 3.5). The lone exception is Soc-Pokec. For this graph, we observe that the counting phase times between both orderings is nearly identical, and that the faster ordering results in a faster total time. Our heuristic is designed for quickly approximating connectivity in large graphs with many cliques. Accurately testing for connectivity requires additional preprocessing with multiple set-intersections, resulting in a significantly slower heuristic. We note that the performance penalty for selecting the wrong ordering in this case is much less.

In this evaluation, we focus on analyzing the impact of different orderings for counting 8-cliques since the clique is big enough for pivoting to perform better than enumeration (Figure 3.21). To determine whether the input clique size (k) impacts the best ordering, we measure the total execution time of the core approximation with $\epsilon = -0.5$, the degree ordering, and our heuristic selecting the ordering for counting varied clique sizes (Figure 3.16). On larger graphs, degree ordering is usually faster for $k = 4$ due to the speed of ordering. However, once the clique size is sufficiently large for pivoting to be faster ($k \geq 8$), which ordering is best

does not change. Since the total execution time for pivoting does not change with k, our heuristic does not consider k.

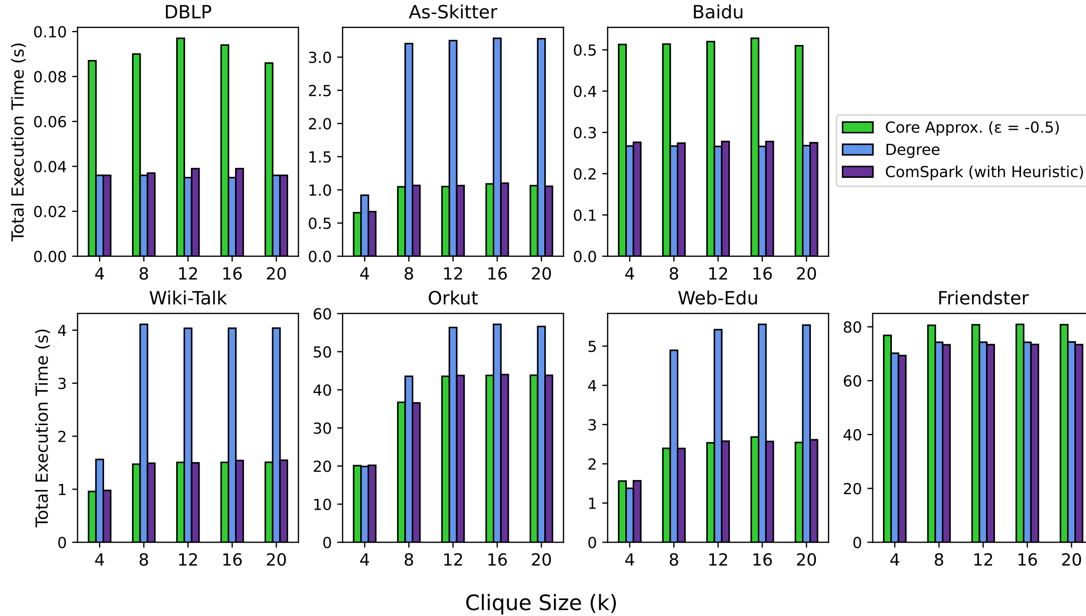

Figure 3.16: Total execution time for counting varied clique sizes using only our core approximation, only the degree ordering, or the ordering selected by our heuristic. Our heuristic always selects the correct ordering for these graphs and it does not add significant overhead.

3.6.4 Reduction in Memory Usage

We compare the performance of our different subgraph structures in terms of counting speed and memory consumption. We first measure the memory usage for our dense subgraph structure in ComSpark[1].

[1]Maximum memory usage collected via the `Maximum resident set size` reported by the command `/usr/bin/time -v`.

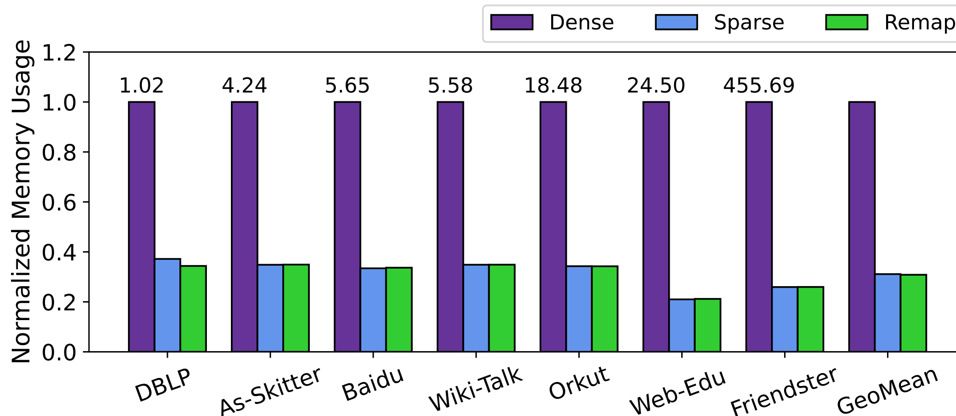

Figure 3.17: Comparing the total process memory usage between ComSpark's dense, sparse and remapped structures for counting 8-cliques in the input graphs. Lower is better. Memory usage is normalized to ComSpark's dense structure memory consumption. Dense structure memory consumption (GB) is denoted above each bar. The same ordering is used for all runs. On average, the sparse and remapped structures result in 3.31× less memory consumption.

As expected, we find that our compact sparse and remapped structures are very memory efficient, requiring 3.31× less space compared to our dense structure across the suite of input graphs (Figure 3.17). With these optimizations, we are able to fit a larger set of the subgraph in cache, resulting in better locality (Figure 3.18).

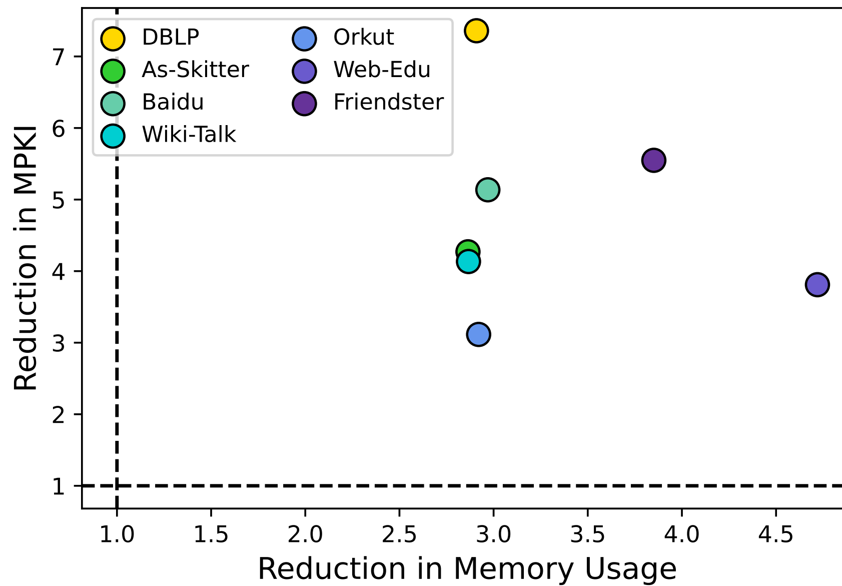

Figure 3.18: Comparing the reduction in memory usage and MPKI between ComSpark's dense and remapped structures for counting 8-cliques in the input graphs. Lower is better. Both metrics are normalized to ComSpark's dense structure. The same ordering is used for all runs. Compaction in the remapped structure allows a larger fraction of the induced subgraph to reside in cache, reducing the miss rate.

The remapped structure provides the fast access time of the dense structure (Figure 3.19), and the memory compression of the sparse structure (Figure 3.17). As core count, and consequently contention for shared memory, increases, our remapped structure becomes a more scalable solution.

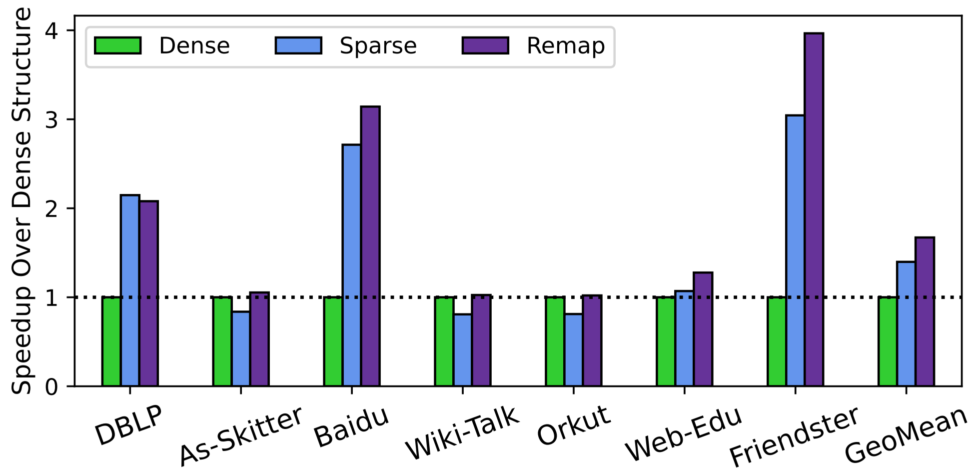

Figure 3.19: Comparing the performance of different ComSpark memory structures while counting 8-cliques on 64 threads. Higher is better. The same ordering is used for all runs. The remapped structure provides the fast access of the dense structure and the memory compression of the subgraph structure, resulting in good overall performance.

3.6.5 Parallel Scaling of k-Clique Counting

To compare the improvement in parallel scaling of our optimizations in the ordering and counting phases with Pivoter, we count 6 and 12-cliques on the input graphs on increasing thread counts up to 64 threads using the different subgraph structures (Figure 3.20). Each plot shows the parallel speedup of the entire algorithm (includes the time to compute the heuristic and times for both ordering and counting phases) relative to the single thread performance of that implementation, and each includes a dashed line to indicate ideal parallel scaling.

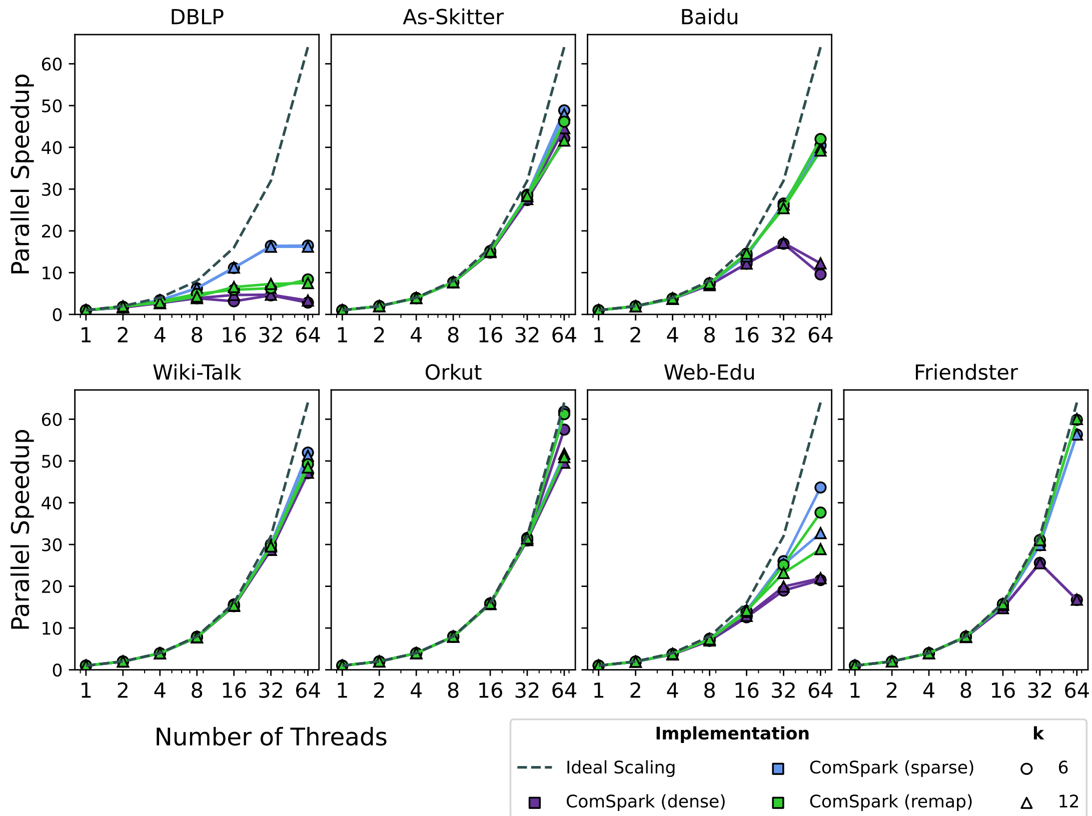

Figure 3.20: Comparing parallel scaling of between different subgraph structures in ComSpark for the entire process of counting 6, 12-cliques in all input graphs. The time for each run includes the time to compute the heuristic, and both ordering and counting phases. Both ComSpark structures scale linearly, resulting in better overall performance. For Baidu and Friendster, memory becomes a bottleneck for our dense implementation at 32 threads. Our more compact sparse and remapped structures avoid this and scales linearly even beyond 32 threads.

We observe that all implementations of ComSpark achieve near-linear scaling for most graphs. In Baidu and Friendster, we observe that scaling for ComSpark's dense structure plateaus at 32 threads. In contrast, by compressing the subgraph using ComSpark's sparse and remapped structures, we are able to achieve linear scaling for both of those graphs. This greatly increased scalability is largely enabled by the more efficient memory use of our subgraph data structures. Scaling for DBLP plateaus beyond 8 threads. We attribute this to being a very small

graph and not having enough parallel work to saturate all cores for the duration of the run. Testing various schedulers with different task granularities does not improve performance. Despite poor scalability, the total execution time is very small (0.04s). We use the remapped structure in our main implementation due to its low memory usage and speed advantage.

3.6.6 Total Execution Time Comparison

We next consider the overall execution times for counting cliques of various sizes on all the input graphs for each implementation (Figure 3.21 and Table 3.6). While reporting the execution time, we include all preprocessing, including the time required to compute our heuristic and directionalize the graph. Consistent with prior work, we exclude the time required to read and build the undirected input graph. Each execution uses 64 threads, and we report the average time of two trials to account for variance. We use the execution times for GPU-Pivot as reported [4] for the NVIDIA Volta V100 GPU for the input graphs in common. GPU-Pivot does not report times for cliques larger than $k = 11$. We report the times for all of our subgraph structures for comparison in Table 3.6.

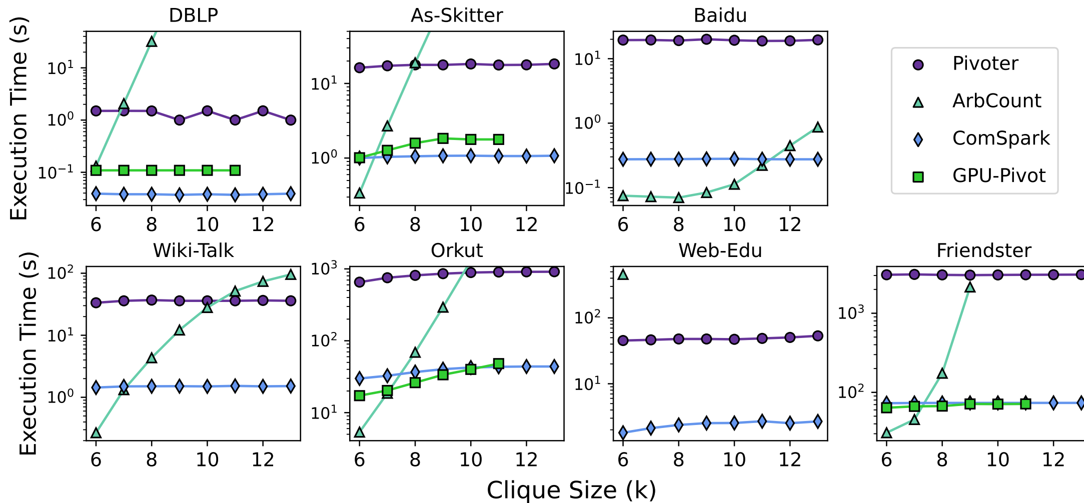

Figure 3.21: Total execution time (on a log scale) required for counting cliques of different sizes on the input graphs for each of the CPU algorithms (Pivoter [56], Arb-Count [104], ComSpark) and GPU-Pivot running on an NVIDIA Volta V100 GPU. Lower is better. GPU-Pivot does not report times for $k > 11$. We observe that the lone enumeration-based algorithm (Arb-Count) takes longer for higher values of k. In contrast, the pivoting-based approaches typically do not get slower for higher k. Due to improved parallel scaling, ComSpark is much faster than Pivoter, despite requiring constant time for various k. This allows the inflection point at which pivoting starts to win to decrease from $k = 10$ to $k = 8$ on larger graphs. Due to better scaling, ComSpark outperforms GPU-Pivot for all k on DBLP and larger k on two out of the four common graphs (As-Skitter and Orkut).

We observe that Arb-Count's execution time increases greatly with clique size (k) for most graphs, while the remaining pivoting-based approaches increase much more slowly (Table 3.6). This is consistent with findings from prior literature. A pivoting-based approach has a fixed initial cost due to a deeper recursive call stack irrespective of input clique size (k). However, once this process is finished, counting different sized cliques requires nearly negligible additional time. LiveJournal is the lone exception, where even pivoting-based methods do not have constant execution time for increasing k. LiveJournal has significantly more cliques/vertex than the rest of the graphs, and the time saved due to an early termination op-

Graph	Algorithm	k=6	k=7	k=8	k=9	k=10	k=11	k=12	k=13
DBLP	Pivoter	1.50	1.00	1.50	1.00	1.50	1.00	1.50	1.50
	Arb-Count	0.13	2.07	32.11	450.86	$> 2h$	$> 2h$	$> 2h$	$> 2h$
	GPU-Pivot	0.109	0.109	0.109	0.109	0.109	0.109	-	-
	ComSpark (Dense)	0.13	0.12	0.13	0.12	0.13	0.15	0.12	0.13
	ComSpark (Sparse)	0.07	0.07	0.06	0.07	0.07	0.08	0.06	0.06
	ComSpark (Remap)	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**
As-Skitter	Pivoter	16.26	17.27	17.77	17.74	18.26	17.69	17.78	18.29
	Arb-Count	**0.38**	2.51	18.34	125.52	754.08	4189.38	$> 2h$	$> 2h$
	GPU-Pivot	1.01	1.27	1.59	1.84	1.78	1.78	-	-
	ComSpark (Dense)	1.04	1.11	1.10	1.10	1.11	1.16	1.09	1.09
	ComSpark (Sparse)	1.24	1.31	1.31	1.34	1.36	1.31	1.32	1.32
	ComSpark (Remap)	1.00	**1.04**	**1.06**	**1.07**	**1.08**	**1.06**	**1.06**	**1.07**
Baidu	Pivoter	19.44	19.52	19.11	20.03	19.31	18.85	18.94	19.57
	Arb-Count	**0.07**	**0.07**	**0.07**	**0.08**	**0.11**	**0.22**	0.45	0.90
	ComSpark (Dense)	1.19	1.02	1.02	1.01	1.03	1.12	1.17	1.02
	ComSpark (Sparse)	0.38	0.38	0.37	0.39	0.40	0.39	0.39	0.38
	ComSpark (Remap)	0.28	0.28	0.28	0.28	0.28	0.28	**0.28**	**0.28**
Wiki-Talk	Pivoter	33.42	35.91	36.91	35.93	35.91	35.93	36.45	35.95
	Arb-Count	**0.28**	**1.32**	4.60	13.24	28.60	51.30	73.87	95.76
	ComSpark (Dense)	1.47	1.58	1.54	1.54	1.52	1.56	1.56	1.58
	ComSpark (Sparse)	1.77	1.85	1.90	1.87	1.88	1.91	1.88	1.91
	ComSpark (Remap)	1.44	1.50	**1.51**	**1.51**	**1.50**	**1.53**	**1.50**	**1.52**
Orkut	Pivoter	654.13	753.08	812.71	858.04	889.39	904.02	909.91	912.99
	Arb-Count	**5.35**	**18.58**	69.89	281.03	1294.34	$> 2h$	$> 2h$	$> 2h$
	GPU-Pivot	17.23	20.33	**26.18**	**33.64**	**39.96**	48.10	-	-
	ComSpark (Dense)	31.24	33.78	37.29	40.81	42.94	43.90	44.82	44.72
	ComSpark (Sparse)	37.50	41.04	45.95	50.46	53.21	54.58	55.39	55.27
	ComSpark (Remap)	29.86	32.58	**36.69**	**40.25**	**42.45**	**43.42**	**43.83**	**43.85**
LiveJournal	GPU-Pivot	**379.88**	**1639.54**	**6850.99**	-	-	-	-	-
	ComSpark (Remap)	1721.29	5991.74	18739.85	**49871.35**	**115426.48**	-	-	-
Web-Edu	Pivoter	45.29	46.36	47.84	47.82	47.25	48.79	50.47	53.35
	Arb-Count	456.47	$> 2h$	$> 2h$	$> 2h$	$> 2h$	$> 2h$	$> 2h$	$> 2h$
	ComSpark (Dense)	3.05	3.09	3.11	3.30	3.26	3.35	3.51	3.56
	ComSpark (Sparse)	2.16	2.59	2.91	3.07	3.06	3.09	3.14	3.13
	ComSpark (Remap)	**1.81**	**2.12**	**2.37**	**2.53**	**2.54**	**2.70**	**2.52**	**2.68**
Friendster	Pivoter	3064.48	3097.26	3054.73	3032.45	3050.13	3063.23	3070.55	3080.26
	Arb-Count	**30.77**	**44.19**	166.53	2132.27	$> 2h$	$> 2h$	$> 2h$	$> 2h$
	GPU-Pivot	63.87	66.54	**67.06**	**71.40**	**71.05**	**71.45**	-	-
	ComSpark (Dense)	301.55	297.45	298.61	296.48	298.48	297.94	298.58	296.30
	ComSpark (Sparse)	97.21	97.82	98.10	98.03	98.21	98.14	98.33	98.27
	ComSpark (Remap)	72.97	73.39	**73.53**	**73.55**	**73.55**	**73.58**	**73.62**	**73.62**

Table 3.6: Summary of total execution time for counting cliques using Pivoter [56], Arb-Count [104], GPU-Pivot [4] and ComSpark. We use the times reported by GPU-Pivot in their paper. Every other algorithm is executed using 64 threads on the same machine (CPU) under the same conditions. The execution times reported include any preprocessing, including graph ordering, but ignore graph reading times. The best CPU execution time is denoted in bold and if the GPU execution time is the fastest, it is denoted in green.

timization is more apparent. However, we observe that the rate of increase in execution time reduces as k increases. This implies that the execution time will be constant beyond a large enough k.

Initially, Pivoter starts to win out over enumeration-based algorithms on CPUs at $k = 10$. Most notably, we enable pivoting to be more advantageous earlier, at $k = 8$ for larger graphs with ComSpark's parallel scalability. ComSpark is the first work to count 10-cliques in LiveJournal in under two days. ComSpark outperforms GPU-Pivot on DBLP and has comparable performance on Friendster while using a midrange server that costs a fraction of the premier GPU.

3.6.7 Comparison Against GPU

Lastly, we consider the performance differences between the optimized parallel implementations of pivoting on the CPU (ComSpark) and the GPU (GPU-Pivot) for counting cliques of different sizes in the common graphs. The amount of work while counting smaller cliques is typically less than that for larger cliques due to an optimization in the original Pivoter code that allows the counting to terminate early (Figure 3.22). This allows GPU-Pivot to be faster than the CPU implementations for moderate k when the amount of work has not hit its peak (Table 3.6).

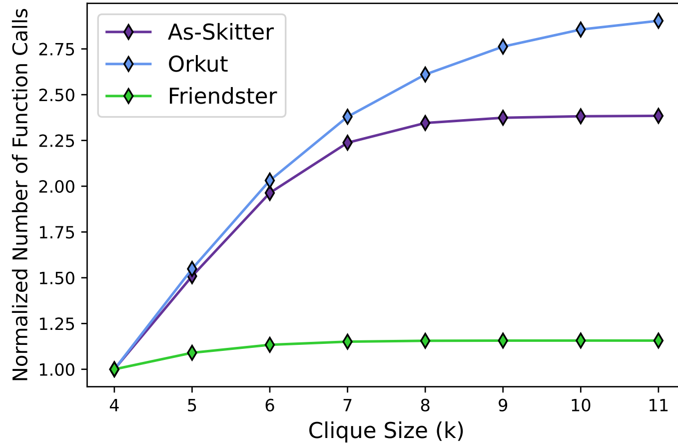

Figure 3.22: Number of function calls required in the counting phase while counting cliques of different sizes. The number of calls are normalized to that of counting 4-cliques in each graph. Counting larger cliques in denser graphs with many cliques like As-Skitter and Orkut result in more work until it plateaus. This is due to an optimization in the original Pivoter code that allows early termination in the counting process.

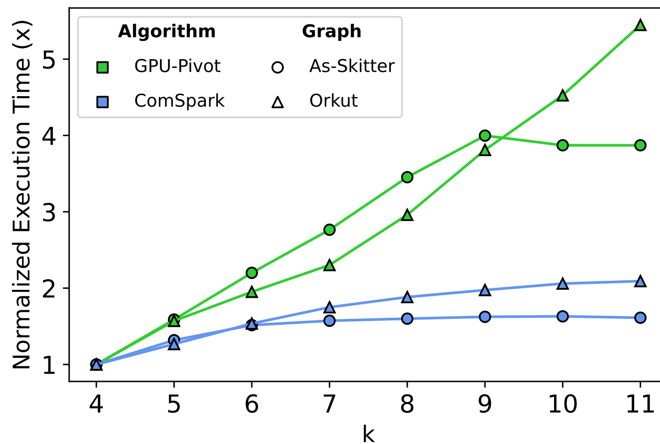

Figure 3.23: Self-normalized execution times for GPU-Pivot and ComSpark for As-Skitter and Orkut. We normalize the execution times for various values of k by dividing by the execution time of $k = 4$ for that specific implementation. We observe that the execution time for GPU-Pivot increases with k. In contrast, ComSpark's execution time does not increase significantly. Notably, in the case of As-Skitter, ComSpark has almost constant execution time for all values of k. This allows ComSpark to outcompete GPU-Pivot for larger k.

We also observe that the total execution time for GPU-Pivot increases linearly with k as the amount of work increases when there are a lot of cliques in the graph (Figure 3.23). We suspect this is due to an increased number in intersection operations while building the subgraph in each recursive level. Additionally, the nature of the pivoting algorithm does not allow for efficient use of the GPU's hardware resources, inhibiting scalability. In contrast, the execution time for ComSpark does not increase significantly with k. This allows ComSpark to outperform GPU-Pivot for larger k on these two graphs. Our approach is more scalable for counting large cliques on challenging graphs like those. Both GPU-Pivot and ComSpark do not show significant variation in execution time relative to k on graphs where work does not increase significantly, like Friendster.

Comparison on LiveJournal

We also include a comparison against LiveJournal, since it is a computationally challenging graph with many cliques.

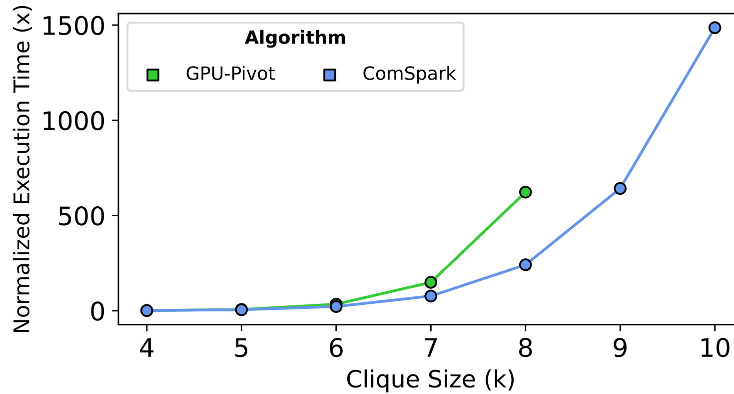

Figure 3.24: Self-normalized execution times for GPU-Pivot and ComSpark for LiveJournal. We normalize the execution times for various values of k by dividing by the execution time of $k = 4$ for that specific implementation. While the execution times for both algorithms increase with k, ComSpark shows a slower rate of increase.

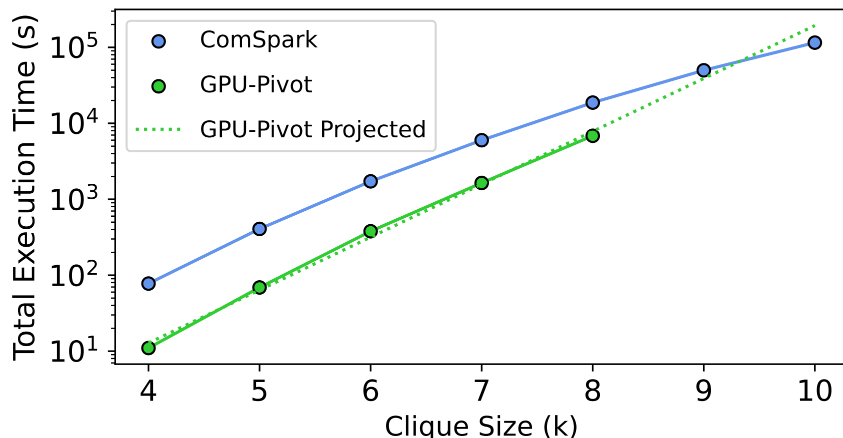

Figure 3.25: Total execution times for counting various k-cliques in LiveJournal using ComSpark and GPU-Pivot on a log-scale. Since GPU-Pivot only reports times up to $k = 8$, we project the performance for counting $k >= 9$ for GPU-Pivot using a regression. We use the equation $y = 0.021 * e^{1.6037x}$ to model the performance of GPU-Pivot. The R-squared value for our regression is 0.99.

We observe that the execution time of both the CPU and GPU increases significantly with k for LiveJournal, compared to As-Skitter and Orkut, other relatively challenging graphs (Figures 3.23 and 3.24). We also observe that the execution time of the GPU increases significantly more with k than ComSpark. Consequently, the performance gap between ComSpark and GPU-Pivot reduces as k increases (Figure 3.25). Using an exponential regression ($y = 0.021 * e^{1.6037x}$, $R^2 = 0.99$) to project the performance for GPU-Pivot for $k >= 9$, we predict that ComSpark will outperform GPU-Pivot at $k = 10$. This is consistent with what we observe with Orkut, another graph with many cliques (Figure 3.21). In their paper, the authors of Pivoter report an execution time of 5.9 days for counting 10-cliques in LiveJournal [56]. With ComSpark's improved scalability, we are able to perform this in 1.3 days.

Even though time-to-solution is the most important metric, we also we compare the performance normalized to power, number of transistors and area for

both systems since their architectures are fundamentally different (Table 3.8). We compute performance as $\frac{1000}{time}$ (higher is better). Since the implementation details for the CPU are not public, we use estimates obtained from [112] (Table 3.7). For the Volta V100, we use details provided in the V100 whitepaper [89].

System	Model	TDP (W)	Transistors (B)	Area (mm^2)
CPU	AMD EPYC 9554	360	52.6	576
GPU	NVIDIA Volta V100	300	21.1	815

Table 3.7: Comparison of CPU and GPU specifications. These values were obtained from [89, 112].

Graph	Perf/Watt		Perf/transistor		Perf/mm^2	
	k = 8	k = 11	k = 8	k = 11	k = 8	k = 11
DBLP	0.44	0.44	0.91	0.91	0.26	0.26
As-Skitter	0.80	0.73	1.66	1.51	0.47	0.43
Orkut	1.68	1.08	3.49	2.25	0.99	0.64
Friendster	1.32	1.24	2.73	2.57	0.77	0.73
GeoMean	0.94	0.81	1.95	1.68	0.55	0.88

Table 3.8: Performance of the GPU normalized to that of the CPU. We use the power (TDP), transistor and area (mm^2) for each system as reported in Table 3.7.

On average, we observe that the CPU results in better performance per-Watt and per-mm^2 (Table 3.8). The GPU has better energy efficiency on the larger graphs (Orkut and Friendster), and consistently better performance per-transistor on all graphs. Finally, we observe that the CPU results in consistently better area efficiency. We also observe that the efficiency of the GPU drops as we count larger cliques. For counting sufficiently large cliques, the CPU could potentially become

the leading energy and area-efficient solution.

Pivoting algorithms are more suited for counting large cliques in graphs and enumeration algorithms perform well for smaller cliques. A hybrid algorithm which performs well for all clique sizes can easily be implemented by switching with a simple heuristic e.g. ($k \geq 8$). Due to its improved parallel scalability in both ordering and counting phases, ComSpark typically outperforms current state-of-the-art CPU and GPU algorithms for large clique sizes ($k \geq 8$).

3.7 Related Work

One of the fastest sequential algorithms for clique counting is provided by Chiba and Nishizeki [28]. While the algorithm is simple and efficient, it includes a step which makes it sequential and suboptimal for dealing with the massive real-world graphs of today. Finocchi et al. present a scalable algorithm for counting k-cliques using the MapReduce framework [43]. They use a degree ordering to direct the graph. KClist was the first work to parallelize Chiba and Nishizeki's algorithm by separating the ordering and counting phases [32]. More recently, Shi et al. present Arb-Count, a work-efficient parallel algorithm with polylogarithmic span [104]. Aside from the core and degree ordering, they also implement the Barenboim-Elkin [6] and Goodrich-Pszona [45] orderings. This work is considered to be the state-of-the-art parallel clique counting algorithm amongst the enumeration-based methods. Li et al. provide an optimized parallel algorithm which orders the graph on a coloring-based method [72]. Lastly, Almasri et al. implement ennumeration and pivoting-based k-clique counting algorithms on GPUs (GPU-Pivot) [4]. Aside from the aforementioned dedicated solvers, various GPM frameworks [25, 26, 36, 58, 113] are also capable of counting cliques in large graphs. In addition to exact clique counting, related problems like maxi-

mal clique enumeration and approximate clique counting have also been heavily studied. Maximal clique enumeration involves finding cliques which cannot be extended any further [17, 27, 35, 39, 114]. Other recent works attempt to approximate k-clique counts in graphs through sampling [24, 57, 120, 121] or probabilistic hashing [15].

3.8 Conclusion

In this chapter, we take a holistic approach to improve the parallel scalability of clique counting. We provide in-depth analysis of how the ordering phase impacts the counting phase during clique counting. We take inspiration from prior work to find a parallel approximation of the core ordering which allows ComSpark to benefit from faster ordering times without sacrificing algorithmic efficiency while counting. We use our analysis to develop a heuristic which predicts which ordering algorithm will lead to the fastest counting, and consequently, fastest overall execution time for a given input graph topology. We also improve the parallel scalability of pivoting-based clique counting by reducing memory usage through a compact subgraph structure. While several optimizations have been presented for optimizing the counting phase on GPUs, including an even more memory-efficient subgraph structure, we hope that our parallel ordering methods and heuristic can be used to further improve performance on the GPU. Our remapped subgraph structure may potentially reduce the increase in GPU execution time as k increases. We emphasize that a similar methodology to the analysis provided in this paper can be followed for improving the performance of other Graph Pattern Mining algorithms.

With improved overall performance, we are able to count large cliques in large graphs very efficiently. ComSpark achieves total speedups of $16.26 - 71.78\times$

(Geometric Mean: 28.97×) over Pivoter while counting k-cliques. ComSpark is also faster than the fastest enumeration-based implementation (Arb-Count) for a smaller clique size than before. On challenging graphs with many cliques, ComSpark scales better with clique size, allowing us to outperform the GPU while counting larger cliques.

While we focus on counting the total number of k-cliques in this work, we are easily able to obtain per-vertex k-clique counts with a few simple changes to our code. Since more research is ongoing in the field of Graph Pattern Mining, we hope our optimizations can be used as a performance benchmark to compare the performance of new clique counting algorithms.

Chapter 4

Actor-Based Distributed Breadth-First Search

In this chapter, we optimize communication across the network layer while performing a Breadth-First Search (BFS) traversal on large graphs stored across multiple compute nodes. We implement the leading direction-optimizing BFS algorithm to reduce the amount of data sent across the network during the traversal [9]. We reduce the amount of synchronization while implementing the bottom-up step of this algorithm by leveraging actor messages using the Fine-grained Asynchronous Bulk Synchronous (FABS) programming model.

4.1 Introduction

Traversing a graph in a breadth-first search (BFS) order is a fundamental algorithm in computer science, and is commonly used to test connectivity and find shortest paths. The widespread use of BFS makes it a commonly used graph benchmark [86]. BFS has low computational intensity and requires a large amount of communication to check for visited vertices, resulting in low IPC. We consis-

tently measure IPCs less than 0.3 for a variety of graphs on our Intel Xeon system. The impact of communication is exacerbated when the graph is distributed across many servers, and has to send high-latency messages over a network. As systems and graphs get bigger, it is necessary to have an algorithm which scales well.

A BFS traversal starts at a source vertex and attempts to find every vertex in the graph that is accessible from the source. The conventional BFS approach is level-synchronous top-down, starting at the source vertex and expanding its frontier outwards while visiting every vertex at that level before moving on to the next level. Since real-world graphs tend to be low-diameter and follow a power law distribution, most vertices are visited within the first few steps. This approach benefits from a lot of parallelism due to the low-diameter topology of many real-world graphs like social networks.

Direction-optimizing BFS, an optimzed algorithm by Beamer et. al. [9], reduces the amount of communication required by dynamically switching to a *bottom-up* approach when the frontier becomes large. Existing distributed bottom-up methods require a large amount of barrier synchronization instructions, and they quickly become a performance bottleneck [13, 18, 21]. In this work, we leverage asynchronous actor messages to efficiently implement both top-down and bottom-up steps. We implement different messaging schemes for bottom-up BFS and analyze the tradeoffs in communication volume and performance between them. We also implement an actor-based direction-optimizing BFS which dynamically switches between top-down and bottom-up based on the size of the frontier. Our actor-based approach is competitive with prior leading approaches. Actor-based top-down BFS has been implemented before [23,98]. To the best of our knowledge, ours is the first actor-based implementation of direction-optimizing BFS.

4.2 Background

4.2.1 Conventional Top-Down BFS

The conventional BFS approach is top-down, starting at a source vertex and expanding its frontier outwards while visiting every vertex at that level before moving on to the next level. In each level, every vertex in the frontier checks all of its neighbors to find those that are unvisited. Every unvisited neighbor is marked as visited and then added to the frontier for the next level. Thus, the total number of edge checks required by the top-down approach is the same as the number of edges in the connected component containing the source vertex.

Due to the scale-free topologies of real-world social networks, the frontier exponentially grows in the first few rounds and then rapidly decreases. By virtue of being low-diameter, these networks tend to have a small number of levels. Since most of the vertices are visited in the first few levels, the later levels result in many failed edge checks to find unvisited vertices.

Algorithm 8 Breadth-First Search

```
1: function Breadth-First Search(graph)
2:     frontier ← { source }
3:     next ← { }
4:     parents ← [-1, -1, -1, ..., -1]          ▷ Mark all vertices as unvisited
5:     parents[source] ← source
6:     while frontier ≠ { } do
7:         Top-Down-Step(graph, frontier, next, parents)
8:         frontier ← next
9:         next ← {}
```

Algorithm 9 Top-DownStep

```
1: function Top-Down-Step(graph, frontier, next, parents)
2:     for all u ∈ frontier do
3:         for all v ∈ neighbors[u] do
4:             if parents[v] = -1 then                    ▷ v is unvisited
5:                 parents[v] ← u
6:                 next ← next ∪ {v}
```

Distributed implementations of the top-down approach are level synchronous, requiring a barrier to allow for each of the processing elements (PEs) to have a global view of the frontier.

4.2.2 Bottom-Up BFS

In contrast to the conventional top-down approach, the bottom-up approach attempts to find visited parents of unvisited vertices. This approach can be expensive when there are a large number of unvisited vertices. However, this approach can be more algorithmically efficient than top-down when the frontier is large, avoiding many redundant edge checks to already visited vertices. Beamer et. al. [9] leverage this observation and combine the top-down approach with a bottom-up approach to reduce the amount of edges checked in their novel direction-optimizing BFS.

Algorithm 10 Bottom-Up Step

```
1: function BOTTOM-UP-STEP(graph, frontier, next, parents)
2:     for all u ∈ vertices do
3:         if parents[u] = -1 then
4:             for all v ∈ neighbors[u] do
5:                 if v ∈ frontier then
6:                     parents[u] ← v
7:                     next ← next ∪ {u}
8:                     break
```

There is a performance tradeoff between parallelizing and serializing the inner for-loop (Step 4 in Algorithm 10). Exposing parallelism in the inner for-loop allows for efficient utilization of a large number of PEs at larger scale. In contrast, serializing the inner loop combined with the *break* statement (Step 7 in Algorithm 10) reduces the number of edges checked during the traversal, making it more algorithmically efficient.

In order to get the benefits of early termination for the inner-loop, prior MPI-based distributed implementations like CombBLAS are forced to communicate greatly, and bitmap compression only partially constrains that growing amount of communication. Bottom-up also requires a significant amount of synchronization in MPI due to the request-response communication pattern. There needs to be a barrier after every message to ensure it is received before it can begin computation. Unsurprisingly, interprocessor communication is a performance bottleneck in distributed graph algorithms [77]. Beamer et. al. [13] observe sub-linear scaling while implementing distributed direction-optimizing BFS, as the bottom-up step suffers from high communication overhead.

4.2.3 The Actor Model

The Actor Model is a concurrent programming model where different actors can communicate with each other via asynchronous message-passing [53]. We treat each core/processing element (PE) in the system as an actor. Upon receiving a message, an actor invokes a message handler which can perform local computation or message another remote actor.

Since the actor's local state is private, no locks are required. This allows the Actor Model to scale to a large number of PEs, making it an ideal fit to implement in distributed-memory systems.

Common Actor Terms

- Processing Element (PE): A unit of computation within a parallel system. We treat each core in the system as a unique PE.

- Mailbox: Inbox of an actor which stores messages sent to it [34]. Each mailbox has its own message handler which calls a message handler. The message handler can perform some local computation or message another actor.

- Selector: An actor with multiple mailboxes. Each mailbox has its own message handler that can perform a different task.

- Non-blocking actor message: An asynchronous message sent by an actor. Non-blocking implies that the sender PE does not need to wait to resume computation after sending a message. The receiver PE does not need to immediately drop its current task to process the received message.

4.2.4 HCLib-Actor Framework

A defining feature of the Actor Model is its inherent asynchrony, i.e. there is no requirement for an order in which messages must be processed. The HCLib-Actor framework provides a productive programming interface for asynchronous actor-based applications in combination with the scalable and performant HCLib runtime [96, 97]. It is built on the Conveyors library [80] which performs tasks like message aggregation, but exposes a very complicated programmer interface. Selectors, i.e. actors with multiple mailboxes, are a key concept in the HCLib-Actor framework [55]. Multiple mailboxes allow for different computations to be triggered by the message handlers, enabling complex dataflows in graph processing among other applications.

The traditional Bulk-Synchronous Parallel (BSP) approach for parallel computation consists of multiple ”supersteps” in which barriers ensure the completion of any communication (one-way messages) within the superstep before they are viewable. BFS requires sending a large number of messages, resulting in many barriers. Certain PEs may be forced to spend a significant amount of time waiting at barriers in the case of uneven work distribution amongst the PEs due to graph topology. Thus, BSP may not result in effective utilization of system resources for irregular, communication-intensive workloads like BFS.

In contrast, the Fine-grained Asynchronous Bulk Synchronous (FABS) model proposed by Paul et. al. [96] extends this model to allow for asynchronous two-way messaging within supersteps. Overlapping multiple rounds of communication and computation within a superstep results in fewer idle cycles and better utilization of resources. This is supported by fine-grained, atomically processed actor messages, automatic message aggregation, and an asynchronous tasking runtime. FABS allows us to efficiently sequentialize the inner for-loop of Bottom-Up BFS without

significant synchronization overhead.

4.2.5 The Actor Graph Library (AGL)

The Actor Graph Library is a streamlined distributed-memory graph processing library based on the Actor Model and built on top of the HCLib-Actor framework [48]. AGL recognizes that there are many common steps for implementing various graph kernels on distributed-memory systems that do not involve the development and optimization of the kernel itself. These include things like generating or reading the graph, distributing the graph amongst the PEs intelligently, easily accessing data from the graph, etc. This enables the developer to focus on optimizing their graph processing tasks. We use AGL to implement our actor-based BFS.

Data distribution is a critical factor for achieving high performance on distributed-memory systems. AGL allows for multiple methods to assign the portions of the graph to different PEs to improve locality or communication efficiency [60]. Each PE stores the local portion of the graph in the Compressed Sparse Row (CSR) format. Each vertex in the graph contains two identifiers. One is a unique identifier referred to as the 'global ID'. The second identifier is referred to as the 'local ID', and it denotes its position within a host PE. After the graph has been distributed between different PEs, AGL provides static functions to access vertex-specific data. These include functions for finding the host PE of a vertex given its global ID, and converting a global ID to a local ID to access data within the local structures.

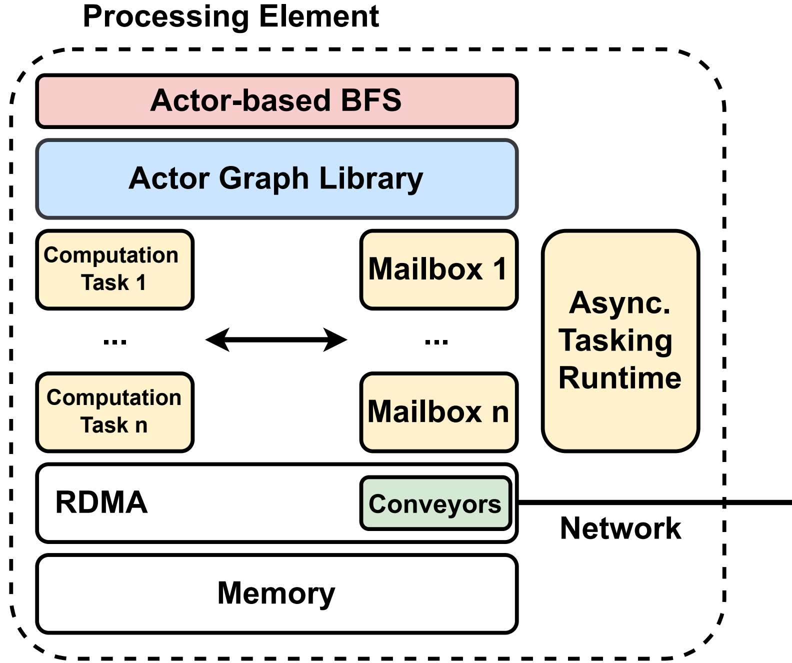

Figure 4.1: Software stack for implementing Actor-based BFS. We implement BFS in the Actor Graph Library (AGL), which is built on top of the HCLib-Actor framework (yellow). The HCLib-Actor framework leverages the Conveyor library (green) for message aggregation.

4.3 Actor-Based Direction-Optimizing BFS

In this section, we detail our implementations of parallel top-down and bottom-up BFS and how we combine them to implement a direction-optimizing BFS.

4.3.1 Parallel Top-Down BFS

Our top-down algorithm is the same in spirit as the pseudocode we highlight in Algorithm 9. We treat each processing element (PE) as an actor, which maintains the parent array for its local vertices as its state. Each vertex in the frontier sends a non-blocking actor messages to all of its neighbors. Our message consists of the two vertices that constitute the edge which is being checked. Upon receiving

the message, a PE checks whether the local vertex in question has already been visited. If the vertex is unvisited, its parent is updated (to mark it as visited), and it is added to the local frontier. Since the parent array is private to each actor, no atomics or locks are required during the update. We include a barrier at the end of each level so each PE knows it is safe to move to the next level.

Algorithm 11 Actor-based Top-Down BFS

```
1:  function Breadth-First Search(graph)
2:      frontier ← { source }                        ▷ Only on PE owning the source
3:      frontier_size ← 1
4:      next ← { }
5:      parents ← [-1, -1, -1, ..., -1]
6:      while frontier_size > 0 do
7:          for all u ∈ frontier in parallel do
8:              for all v ∈ neighbors[u] do
9:                  Send(FINDOWNER(v), {u, v})   ▷ Non-blocking Actor Message
10:         BARRIER()
11:         frontier_size ← ALLREDUCE(next.size())       ▷ Global frontier size
12:         frontier ← next
13:         next ← {}
14: function Top-Down Message Handler({u, v})
15:     if parents[v] = -1 then
16:         parents[v] ← u                 ▷ Local state, no atomics/locks required
17:         next ← next ∪ {v}
```

4.3.2 Parallel Bottom-Up BFS

Our parallel bottom-up algorithm is the same in spirit as the pseudocode we provide in Algorithm 10. Similar to our top-down implementation, we treat each PE as an actor. As we mention previously, there is a performance trade-off between parallelizing and sequentializing the inner for-loop (Step 4 in Algorithm 10). In this work, we explore this tradeoff by implementing various messaging schemes for bottom-up BFS.

Distributed-memory bottom-up requires a request-response communication pattern to find a valid parent. This requires each actor to have multiple mailboxes: one for generating requests to other PEs and one for responding to requests from other PEs. Our message needs to be slightly modified from the top-down case to include which mailbox the message is being sent to. Based on this value, the appropriate message handler is triggered. Selectors allow us to define these extra mailboxes seamlessly.

In our parallel inner-loop version, we send messages to all of the neighbors of every unvisited vertex at once. Of those neighbors, the vertex whose *visited* response is recorded first is assigned to be the parent of the original requesting vertex. The vertex is then marked as visited and then added to the frontier. Sending multiple requests in parallel can be advantageous when the frontier is small, since the latencies of the responses are amortized. However, when the frontier is large, this can result in many redundant edge checks. We refer to this approach as *Parallel*.

Algorithm 12 Actor-based Bottom-Up BFS with Parallel Inner-Loop

```
1:  function BREADTH-FIRST SEARCH(graph)
2:      frontier ← { source }                         ▷ Only on PE owning the source
3:      frontier_size ← 1
4:      next ← { }
5:      parents ← [-1, -1, -1, ..., -1]
6:      while frontier_size > 0 do
7:          for all u ∈ local vertices do
8:              if parents[u] = -1 then
9:                  for all v ∈ neighbors[u] do ▷ Sends multiple requests at a time
10:                     Send(FINDOWNER(v), REQUEST_HANDLER, {u, v})
11:         BARRIER()
12:         frontier_size ← ALLREDUCE(next.size())        ▷ Global frontier size
13:         frontier ← next
14:         next ← {}
15: function REQUEST_HANDLER({u, v})
16:     if parents[v] ≠ -1 then                  ▷ Only send response if valid parent
17:         Send(FINDOWNER(u), RESPONSE_HANDLER, {u, v})
18: function RESPONSE_HANDLER({u, v})
19:     if parents[u] = -1 then               ▷ Only execute if no responses recorded
20:         parents[u] ← v
21:         next ← next ∪ {v}
```

Typically, implementing the sequentialized version of the algorithm is more difficult. Prior works accomplish this by paritioning the search step into multiple sub-steps in which a dense bitmap of the frontier is rotated amongst the processor grid [18]. During each sub-step, only a single processor examines a particular

vertex's edges and then passes on those vertices to the next processor in the subsequent sub-step. In contrast, the HCLib-Actor framework provides enough messaging flexibility to implement a serial request-response pattern at a finer granularity, thus improving the communication efficiency of the algorithm. This requires the addition of an extra mailbox to handle each type of response. If the first neighbor of an unvisited vertex has already been visited, we mark the requesting vertex as visited and add it to the frontier. If the neighbor has not been visited, we send the response to a different mailbox. The handler for this mailbox then sends a request to the next neighbor, and this process continues until the original vertex finds a valid parent or all neighbors are queried. Thus, we only perform an edge check when we know for sure that no valid parent has claimed the vertex yet. We refer to this approach as *Sequential*.

Algorithm 13 Actor-based Bottom-Up BFS with Sequential Inner-Loop

```
1: function BREADTH-FIRST SEARCH(graph)
2:     frontier ← { source }                          ▷ Only on PE owning the source
3:     frontier_size ← 1
4:     next ← { }
5:     parents ← [-1, -1, -1, ..., -1]
6:     while frontier_size > 0 do
7:         for all u ∈ local vertices do
8:             if parents[u] = -1 then
9:                 v ← first neighbor of u      ▷ Only sending a single request at a time
10:                Send(FINDOWNER(v), REQUEST_HANDLER, {u, v})
11:        BARRIER()
12:        font_size ← ALLREDUCE(next.size())                 ▷ Global frontier size
13:        frontier ← next
14:        next ← {}
15: function REQUEST_HANDLER({u, v})
16:    if parents[v] ≠ -1 then
17:        Send(FINDOWNER(u), RESPONSE_HANDLER_FOUND, {u, v})
18:    else
19:        Send(FINDOWNER(u), RESPONSE_HANDLER_NOTFOUND, {u, v})
20: function RESPONSE_HANDLER_FOUND({u, v})
21:    parents[u] ← v
22:    next ← next ∪ {v}
23: function RESPONSE_HANDLER_NOT_FOUND({u, v})
24:    v ← next neighbor of u
25:    Send(FINDOWNER(v), REQUEST_HANDLER, {u, v})
```

We find that the sequential inner-loop version is much slower in the initial

levels when the highest-degree vertices have not been visited yet, and faster in the later levels. During the intial levels, the message latency is a performance bottleneck since the PE has to wait for the prior neighbor to respond before sending the next request. To accelerate the process of finding valid parents without messaging every neighbor at once, we explore different messaging schemes and observe improvements in performance over *Sequential* at modest batch sizes. We take advantage of flexible communication in the Actor model to parameterize our messaging schemes to evaluate its effect on performance. We summarize our different messaging schemes in the following table:

Name	**Messaging Scheme**
Parallel	All neighbors at once
Sequential	One neighbor at a time until valid parent found
Threshold	First t neighbors one-by-one, remaining all at once if parent not found
Exponential	Exponentially increasing number of messages (2^n) every time
Batch	Fixed number of neighbors (b) at a time

Table 4.1: Various messaging schemes for parallel bottom-up BFS

4.3.3 Parallel Direction-Optimizing BFS

We combine our top-down and sequential bottom-up implementations to form a performant parallel direction-direction BFS. The top-down approach is advantageous when the frontier is small, since it requires fewer edge checks. The bottom-up approach is more efficient when the frontier is large, since it easier to find a valid parent. At each level, we implement whichever algorithm will be faster. To decide which algorithm should be implemented at a particular level, we use heuristics from prior work [9] that consider the number of edges emanating from the frontier, and the remaining number of edges to check.

4.4 Evaluation

In this section, we analyze the performance and communication efficiency of our different BFS implementations. We also compare the performance of our parallel distributed-memory direction-optimizing algorithm with one of the leading implementations, CombBLAS [21], and the Graph500 reference code [86].

4.4.1 Experimental Setup

We perform our experiments on the Phoenix cluster at Georgia Tech [59]. Each compute node has dual Intel Xeon Gold 6226 CPUs running at 2.7 GHz, 24 physical cores, 19.25MB shared L3 cache and 192GB of RAM. The nodes are connected together using Infiniband 100HDR interconnect. The cluster has 851 total compute nodes. We compile our code using g++ (version 10.3.0) with -O3 -loshmem flags.

We use a variety of synthetic graphs and one real-world graph (Twitter) [64] for evaluating our implementation. The Twitter graph has 61.5 million vertices and 1.47 billion edges. The first type of synthetic graph we use is based on the R-MAT random graph model [22]. The R-MAT generator creates graphs with very skewed degree distributions and low-degree. We set the R-MAT parameters a, b, c, d to 0.57, 0.19, 0.19, 0.05. The second type of synthetic graph we use is the Uniform Random graph based on Erdős-Réyni model [40]. In this model, the two vertices making up an edge are generated from a uniform random distribution. This graph represents the worst case in terms of locality since every vertex has an equal probability of being the neighbor of another vertex. Similar to the Graph500 benchmark, we denote the size of a graph with a $SCALE$ variable for both types of synthetic graphs. This implies that the graph has 2^{SCALE} vertices. We set the edge-factor (average degree) to 16. All of the graphs we use are undirected and

do not contain duplicate edges.

Our main metric of interest to judge performance is the search rate *Traversed Edges per Second (TEPS)*. Since different algorithms may explore a different number of edges, we maintain consistency by dividing the total number of edges in the graph by the total execution time, which may not be the number of edges actually traversed. To increase robustness, we perform between 8 to 64 trials by randomly selecting different source vertices in the graph to start the traversal. To be consistent with prior work, we report the harmonic mean of the TEPS for each trial.

4.4.2 Bottom-Up Comparison

In this section, we compare the communication efficiency and performance of our different bottom-up implementations. To measure communication efficiency, we first count the number of messages sent to check for valid parents in every level for each messaging scheme. We also consider the total number of messages sent.

We first measure the communication efficiency between the parallel and sequential inner-loop versions. We measure the number of messages sent to check for valid parents in every level for each case.

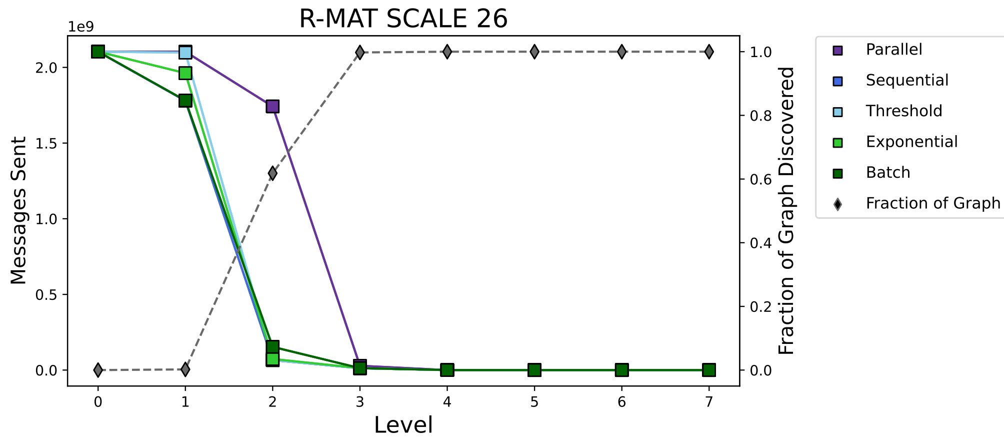

Figure 4.2: Edges checked in each level by different bottom-up implementations for a SCALE 26 R-MAT graph.

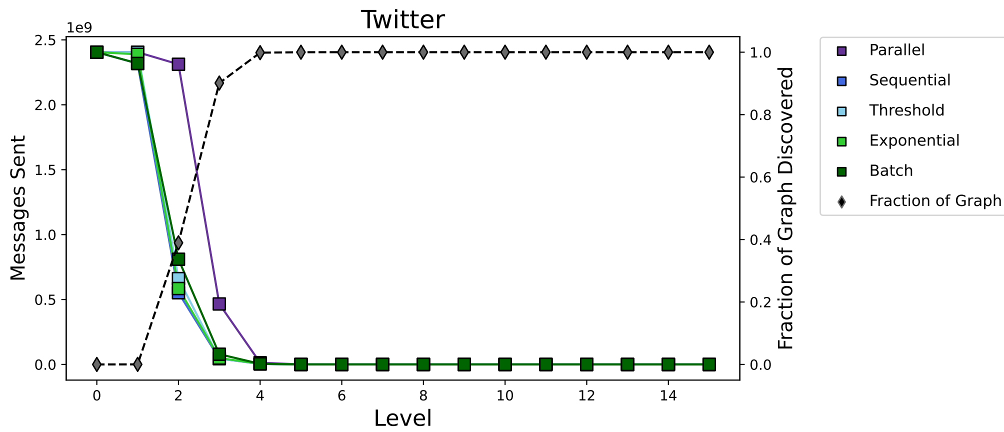

Figure 4.3: Edges checked in each level by different bottom-up implementations for the Twitter graph.

In the intial few levels, we observe that all of the bottom-up versions send a large number of messages (Figures 4.2, 4.3). This is expected, since the frontier is very small at that point, and each unvisited vertex sends a message to all of its neighbors at once trying to find a valid parent. Once the frontier is sufficiently large ($\approx 40 - 60\%$ of the graph), the number of messages sent during *Sequential* is significantly less than that of *Parallel*, since unvisited vertices can find visited

neighbors quickly. This greatly reduces the amount of algorithmic work performed by *Sequential*. We also observe that *Threshold*, *Exponential*, and *Batch* send slightly more messages than *Sequential*, but still significantly fewer messages than *Parallel* (Figures 4.2, 4.3, 4.4). Once the frontier is large enough, and the high-degree vertices have been visited, the efficiency of bottom-up shines through and all versions send a small number of messages for the last remaining levels.

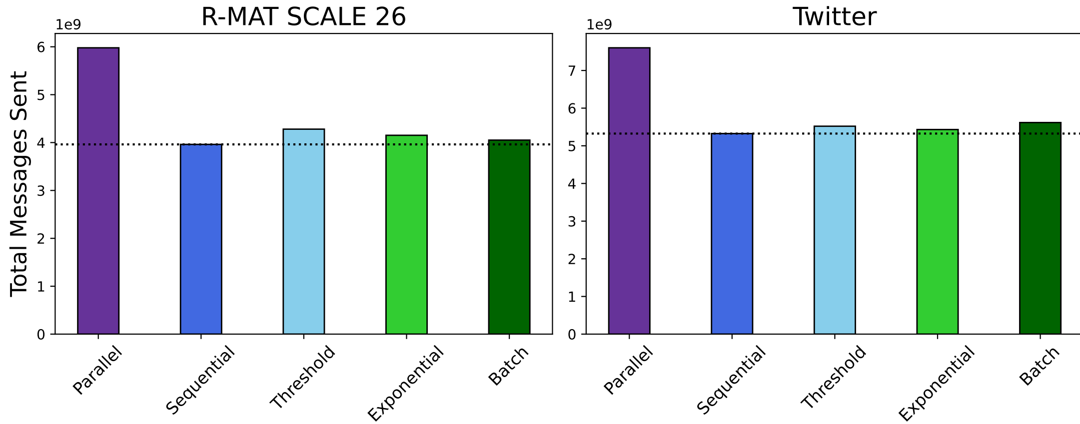

Figure 4.4: Total number of messages sent during different bottom-up implementations for a SCALE 26 R-MAT graph and the Twitter graph.

Despite being more algorithmically efficient, we find that *Sequential* results in the worst performance (Figures 4.5, 4.6). The difference in performance is largely in the first few levels when the frontier is not large enough for bottom-up to be beneficial. A large number of messages are sent in the early levels, and the sequential request-response pattern suffers from high network latency. In contrast, *Parallel* amortizes this latency by sending all the messages in parallel at once.

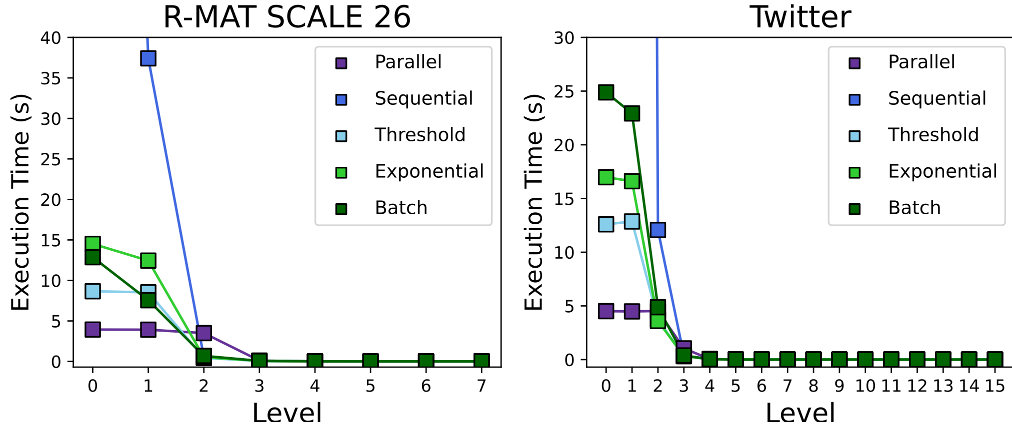

Figure 4.5: Time for each level for different bottom-up implementations for a SCALE 26 R-MAT graph and the Twitter graph. *Sequential* takes 198.61s for the first level on R-MAT and 580.09s and 517.28s for the first two levels on Twitter respectively.

We also evaluate weak scaling performance of the different actor-based bottom-up implementations. We increase graph size as we increase the number of nodes (and cores) and measure the TEPS.

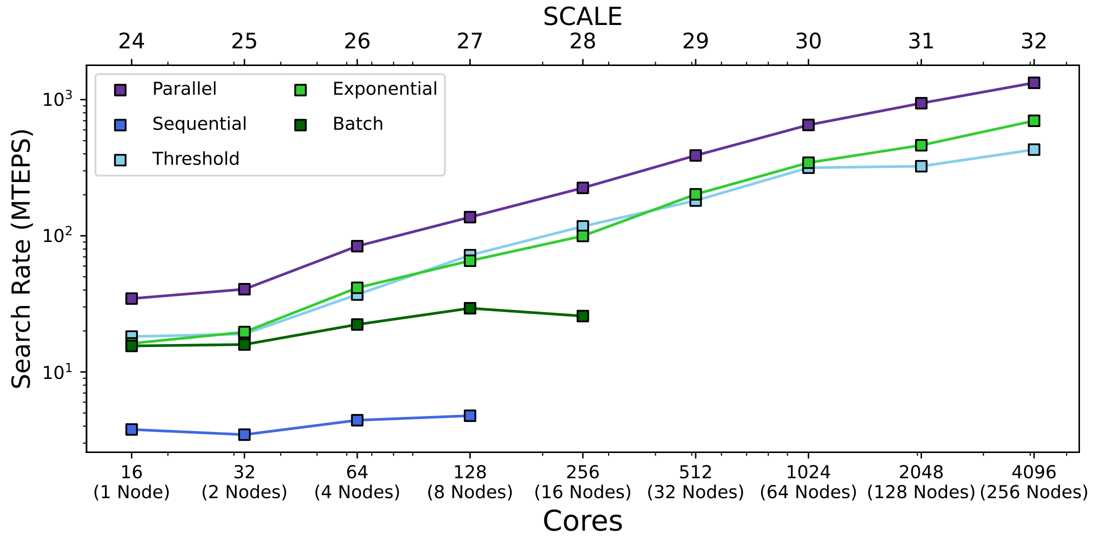

Figure 4.6: Comparison between weak scaling for various bottom-up implementations.

We observe that *Parallel* has the best overall performance (Figure 4.6). Since

a bulk of the time ($\approx 70-90\%$ of total time) is spent in the early levels when the frontier is very small, messaging all of the neighbors in parallel is the best strategy due to overlapping the messaging latency. After the high degree vertices are discovered in the first few levels, sending multiple messages does not significantly increase the amount of algorithmic work. *Sequential* suffers due to a large number of non-overlapping high-latency network messages in the initial levels. Batching multiple messages at a time (*Threshold*, *Exponential* and *Batch*) achieves a good compromise between reducing message count and practical performance.

Since direction-optimizing BFS performs the top-down step in the first few levels when the frontier is small and most of the graph is unvisited, we focus our analysis of the bottom-up portion on the later levels. Bottom-Up is typically advantageous at level 2 on an R-MAT SCALE 26 graph and at level 3 on the Twitter graph (Figure 4.12).

We first attempt to find the best parameters for *Threshold* and *Batch* implementations by testing various values, and measuring the execution time of the relevant BFS levels. For *Threshold*, we vary the number of single request-response queries we perform before messaging all of the remaining neighbors at once. For *Batch*, we vary the number of messages sent at a time and measure its impact on performance.

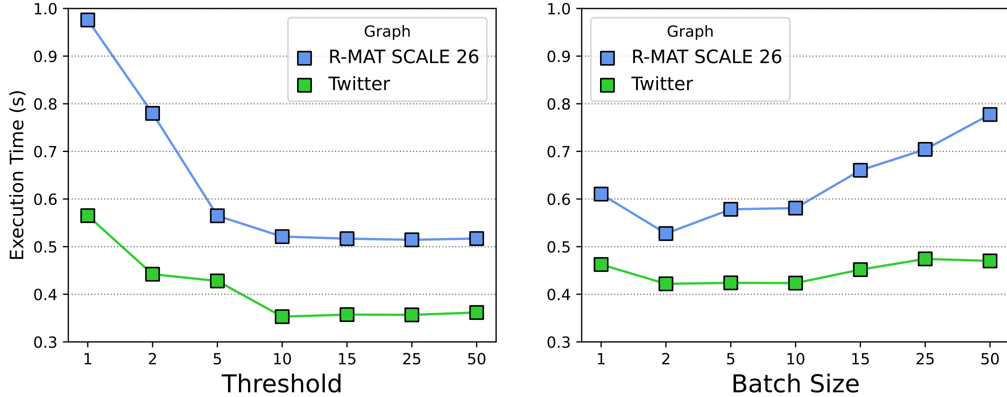

Figure 4.7: Impact of different batch parameters on performance. We vary the number of request-response queries for *Threshold* and the number of messages in a batch for *Batch* (right). We measure the time taken for levels 2 and beyond for the R-MAT graph, and levels 3 and beyond for Twitter.

For *Threshold*, we observe an improvement in performance by increasing the number of communication rounds until it flattens out around 5-10 (Figure 4.7). When threshold $t = 1$, we send a singular message to the first neighbor to check if it is visited. If the first neighbor is unvisited, we message all of the remaining neighbors in parallel. Thus, $t = 1$ effectively behaves like *Parallel*. Note that by level 2 or 3, over 60% of the graph has been visited (Figure 4.2, 4.3). Since most high-degree vertices have been visited, the remaining vertices have a higher chance of finding a visited parent earlier. A moderate threshold (10) allows for sufficient algorithmic efficiency by implementing a sequential request-response pattern but benefits from *Parallel*-like behavior when chances of finding a parent are low. As we increase the threshold beyond 10, the bottom-up starts to behave like *Sequential* and we do not observe any performance improvement by increasing it any further.

While increasing the batch size, we find that marginally increasing batch size to 2 results in the best performance. Messaging two neighbors at a time instead

of one helps to overlap the network latency without sending extraneous messages. Increasing the batch size beyond that reduces the algorithmic efficiency by sending more messages at a time than needed. A batch size $b = 1$ is *Sequential* and a large enough batch size effectively behaves like *Parallel*.

Next, we consider the amount of messages sent and the execution time for the relevant bottom-up BFS levels.

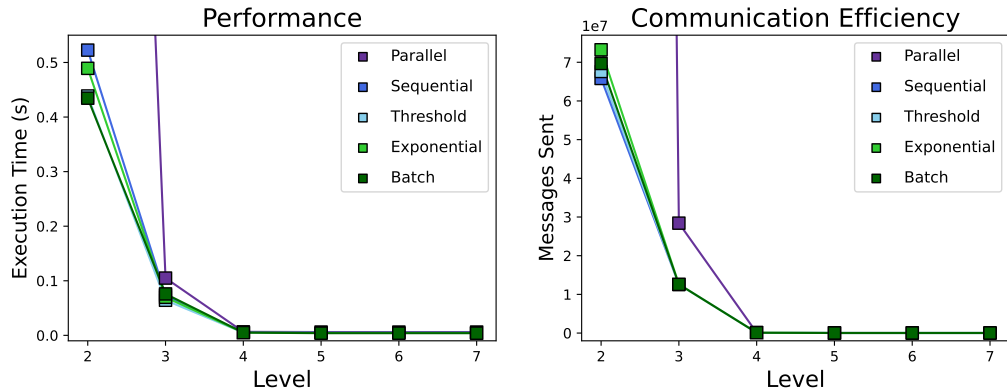

Figure 4.8: Execution time (left) and messages sent (right) for levels 2-7 of bottom-up BFS on an R-MAT SCALE 26 graph. *Parallel* takes 3.50s for level 2.

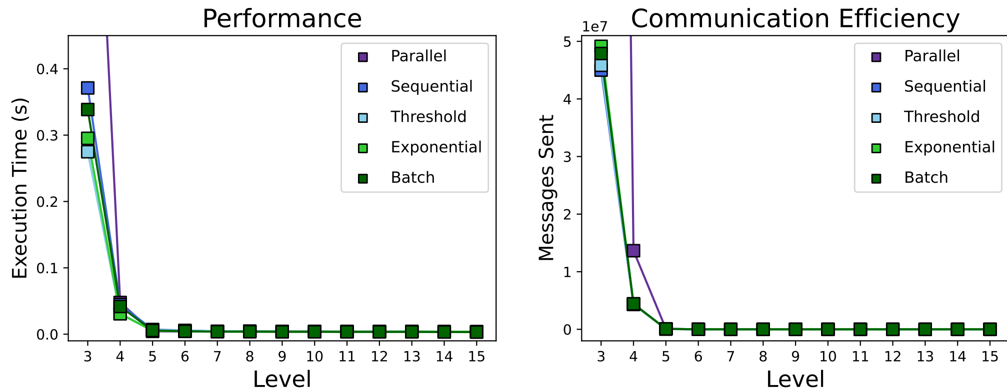

Figure 4.9: Execution time (left) and messages sent (right) for levels 2-7 of bottom-up BFS on the Twitter graph. *Parallel* takes 1.04s for level 3.

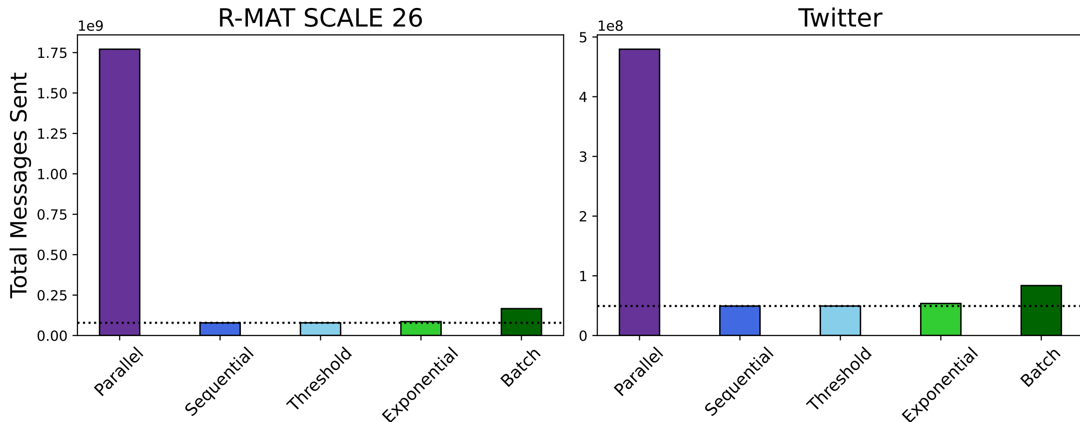

Figure 4.10: Total number of messages sent during the relevant bottom-up levels using different messaging schemes for a SCALE 26 R-MAT graph and the Twitter graph.

On both graphs, we observe that *Parallel* is much slower than the rest of the implementations in the first relevant level (Figures 4.8, 4.9). We also observe that it sends over 10× more messages than the others (Figure 4.10). Unlike the first two levels when *Parallel* is significantly faster due to increased bandwidth utilization, the algorithmic efficiency of fewer messages benefits performance more when the frontier is large enough. In the first relevant level, we observe that *Threshold* and *Batch* are marginally faster than *Sequential*. We attribute this to the fact that some high unvisited high degree vertices remain that are two hops away from a visited vertex. *Batch* sends a fraction of those messages in parallel, reducing average latency. In the subsequent levels after ≈ 99% of the graph is discovered, and the last few vertices remain, all implementations perform equally efficiently since the number of messages is very low. On aggregate, *Threshold* is the fastest bottom-up version in the relevant levels, closely followed by *Batch*. We emphasize that the best bottom-up implementation for direction-optimizing BFS (*Threshold*) is different than the best bottom-up implementation in isolation (*Parallel*).

4.4.3 Direction-Optimizing BFS

In this section, we explain how we combine our top-down and bottom-up BFS to form a hybrid.

For hybrid BFS to perform at its best, it must always select the fastest implementation at each level. Prior works present a simple heuristic based on the number of edges to check in the frontier (m_f) and the number of unexplored edge checks (m_u) [9]. Since these metrics are easy to compute, we use a similar heuristic. To compute (m_f) for a particular level, we sum up the degrees of all the vertices in the frontier. This only requires a single ALLREDUCE operation. We subtract this value from the total number of edges to explore in the graph to get m_u. To switch between top-down and bottom-up, we use the formula $m_f > \frac{m_u}{\alpha}$. We set $\alpha = 10$ since it works well for a majority of graphs (Figure 4.11).

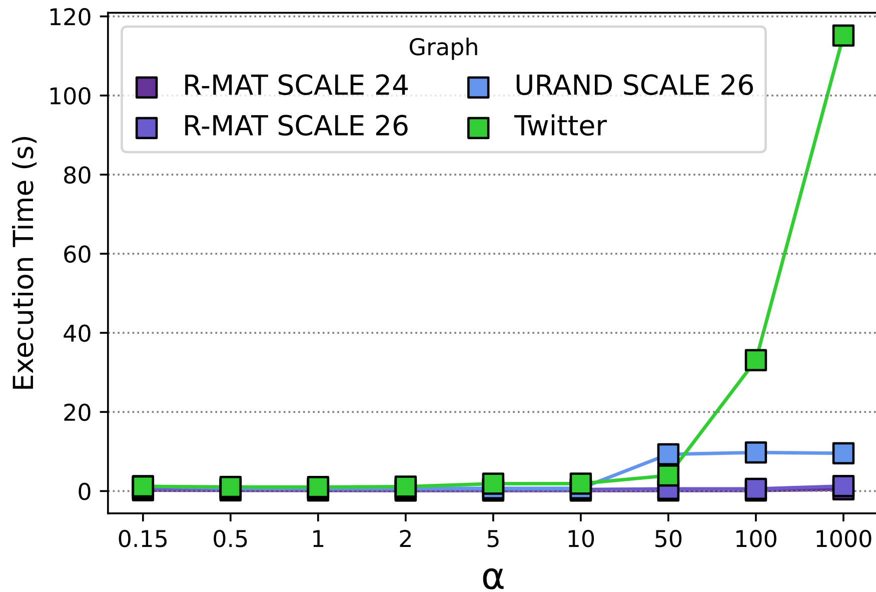

Figure 4.11: Effect of switching heuristic on hybrid BFS performance.

Our tuned hybrid always executes the correct version at each level (Fig-

ure 4.12).

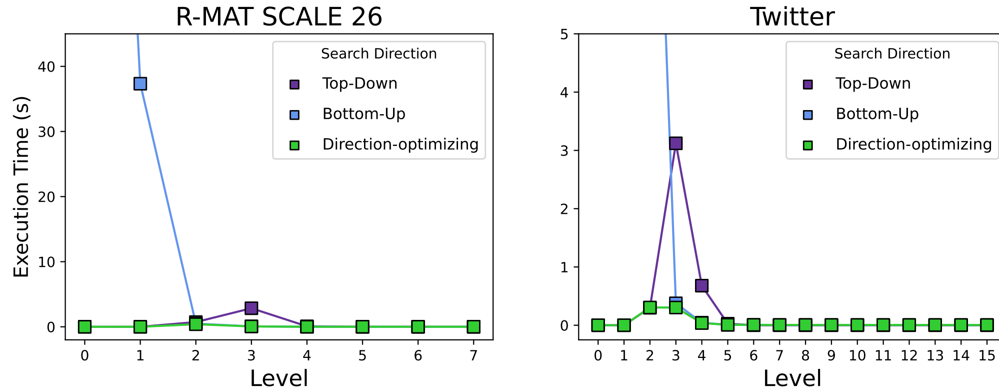

Figure 4.12: Time for each level for the top-down, bottom-up and hybrid implementations for a SCALE 26 R-MAT graph and the Twitter graph.

We find that the performance of our direction-optimizing implementation is the best when using *Sequential* or *Threshold* in the bottom-up step. We use *Sequential* since it results in the fewest messages sent (Figure 4.4).

4.4.4 Comparison Against Prior Work

We compare how our Actor-based BFS performs relative to two existing BFS implementations: the Graph500 reference code and CombBLAS (Figure 4.13). The Graph500 reference code implements top-down BFS using MPI. CombBLAS implements direction-optimizing BFS in MPI.

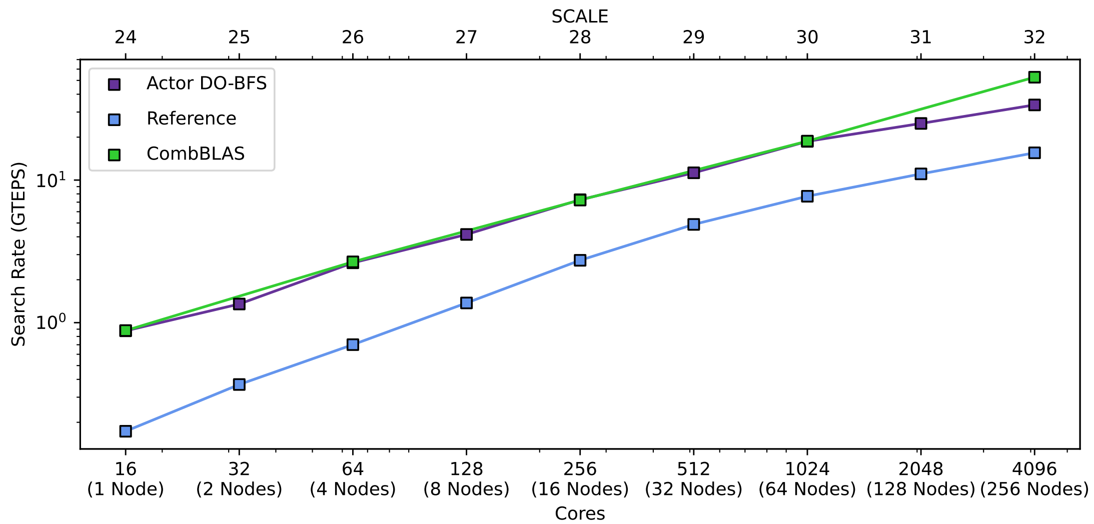

Figure 4.13: Comparison between weak scaling for various implementations.

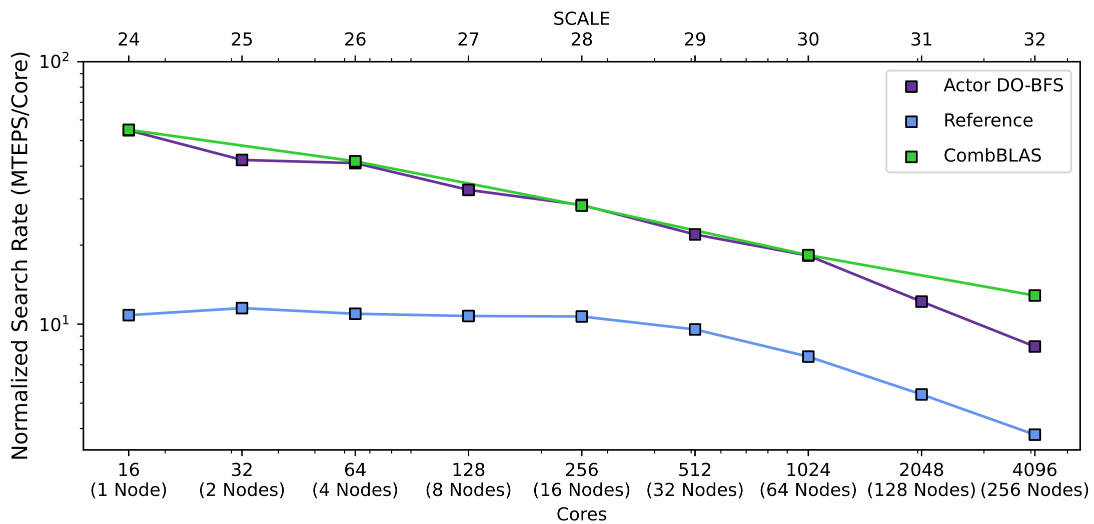

Figure 4.14: Normalized search rate (MTEPS/Core) for various implementations.

Since it only implements top-down BFS, the Graph500 reference is the slowest (Figure 4.13). Its performance is comparable to the performance of our top-down implementation. Our hybrid BFS matches the performance to CombBLAS up to SCALE 30. Since many different factors affect the performance of our

Actor-based hybrid BFS at larger SCALE, we compare our communication volume and synchronization with that of CombBLAS (Table 4.2). While considering the normalized search rate (MTEPS/Core), we find that performance drops across the different implementations when the number of cores increases (Figure 4.14). This implies that larger graphs spread across more cores require more communication, resulting in worse performance. We identify improving the per-core efficiency at larger scale as an avenue for future work.

SCALE	**Actor-based BFS**		**CombBLAS**	
	Average Bytes (MB)	**Barriers**	**Average Bytes (MB)**	**Barriers**
24	531.31	14	560.70	192
26	2196.71	14	4057.19	192
28	8643.25	16	30958.91	192

Table 4.2: Difference in amount of communications and synchronization between our Actor-based hybrid BFS and CombBLAS. We use mpiP [116] to profile CombBLAS.

We observe that our implementation results in fewer bytes transferred and significantly fewer barriers during the BFS traversal (Table 4.2). While the CombBLAS implementation does early termination of the inner-loop, it requires a large amount of communication. Even sending a compressed bitmap between PEs results in significant data movement. Fine-grained asynchronous messages from actors allow us to improve communication efficiency. Additionally, overlapping computation with communication within supersteps in FABS results in significantly fewer barriers. Even though the practical performance is currently on par with existing MPI-based methods, we show the potential of Actors as a solution for contemporary, large-scale communication-bound algorithms.

In addition to better communication efficiency, we also find that programming Actors using HCLib is much simpler. For reference, our direction-optimizing BFS implementation only requires 271 lines of code. In contrast, the MPI-based source

code of CombBLAS is over 1100 lines of code. Application developers can significantly increase productivity by leveraging existing messaging frameworks like HCLib that abstract away complex tasks related to messaging. Programmers only need to specify the contents of a message, its destination, what to do when a message is received and how to convey to all PEs that the work is done for termination.

4.5 Related Work

Parallel BFS has been a well studied topic for many years [100]. Most parallel implementations follow the *level-synchronous* algorithm which requires a communication barrier at the end of each level, giving the PEs a global view of the frontier. Techniques to improve parallel performance of BFS include ensuring proper load balance given the skewed degree distributions of real-world graphs, reducing the amount of synchronization (atomics, locks, barriers) for parallel updates and optimal data placement to improve locality while accessing neighbors of a vertex. The direction-optimizing BFS by Beamer et. al. [9] is the seminal algorithm for parallel BFS. In this algorithm, the conventional top-down step is combined with a bottom-up approach to reduce the amount of algorithmic work required while performing the traversal.

BFS on distributed-memory systems also has a rich history of prior work. Yoo et. al. present a novel 2-D graph partitioning method for top-down BFS on the IBM BlueGene/L [122]. Buluc and Madduri present a novel hybrid 2-D algorithm for distributed-memory systems targeted towards graphs with skewed degree distributions [21]. By combining 2-D partitioning and intra-node multithreading, their algorithm significantly reduces communication overhead. Additionally, Buluc et. al. present their distributed-memory implementation of

direction-optimizing BFS in [18]. Ueno et. al. present their optimizations for their leading distributed-memory implementation of the direction-optimizing BFS algorithm [115]. Their optimizations include a new efficient data structure, vertex reordering and batching update communication messages during the bottom-up step. They are able to perform a BFS traversal on a trillion-vertex graph on the RIKEN K-Computer within half a second. Multiple distributed-memory graph processing frameworks such as Pregel [79], Parallel Boost Graph Library [37, 46] and Gemini [124] include implementations of level-synchronous top-down BFS.

There have been developments in asynchronous graph processing in recent years [49, 117]. Pearce et. al. present a novel shared-memory algorithm for asynchronous graph traversal using top-down BFS [98]. They extend that work to perform asynchronous traversals on scale-free graphs on distributed-memory systems with local non-volatile memory (NAND-flash) [99]. Elmougy et. al. present a study on large-scale asynchronous graph processing using the HCLib-Actor framework where they evaluate PageRank and Jaccard Index [38]. More recently, Chandio et. al. present techniques for optimizing BFS on dynamic graphs by using asynchronous actor messages on decentralized systems [23].

4.6 Conclusion

In this work, we implement an actor-based direction-optimizing BFS in distributed-memory systems. BFS is challenging on these systems due its low computational intensity and high communication needs, which are further exacerbated by the network latency. Due to the increased cost of communication, reducing the amount of edge checks during the traversal is critical for good performance. The direction-optimizing BFS algorithmically reduces the amount of redundant communication by combining the conventional top-down approach with a bottom-up

approach when the frontier gets significantly large. Despite a reduction in the amount of algorithmic work, current practical implementations of distributed-memory Direction-Optimizing BFS are unable to achieve the ideal communication efficiency in the bottom-up phase due to excessive synchronization in existing parallel programming models. To that end, we explore the tradeoffs between algorithmic efficiency and practical performance between different messaging schemes in bottom-up BFS using point-to-point messages in the HCLib Actor Framework. We apply this analysis to creating the most efficient direction-optimizing version. Our hybrid heuristic picks the best version for each level of the traversal. We also compare our communication efficiency and performance with existing leading distributed-memory BFS implementations and show that we reduce the amount of communication and barriers required during the traversal while providing comparable performance. As we grow increasingly dependent on GPU clusters, our work motivates the need for a performant Actor-based messaging library for GPUs that can improve the scalability of communication-bound applications by reducing the communication between GPUs.

Chapter 5

Conclusion

In this dissertation, we present optimizations for improving communication efficiency for different sparse tensor and graph algorithms on both shared and distributed-memory systems.

Our analysis shows that modern general-purpose CPUs are not designed well for the irregular, communication-intensive sparse algorithms we consider in this work. The cost of communication in terms of the sheer number of cycles spent waiting on data severely hampers performance. We typically achieve the best parallel performance improvements at larger scale by reducing the amount of data transferred between compute and DRAM, or between multiple compute nodes connected across a network. By improving performance, we enable solving larger and more complex problem sizes on the same hardware. Additionally, moving less data potentially reduces energy consumption of these systems.

To conclude, we summarize a list of contributions and present ideas for future work.

5.1 Summary of Contributions

We make the following contributions in this work:

1. A workload characterization of MTTKRP, a sparse tensor kernel (Chapter 2). Across multiple leading tensor frameworks, we show that only a few last-level cache misses significantly impact processor performance due to memory latency. We present two optimizations (SIMD and Software Prefetching) to improve performance by increasing memory-level parallelism.

2. ComSpark, a parallel pivoting-based clique counting algorithm (Chapter 3). This is the first work to showcase the benefits of a parallel ordering for this problem, as prior work only uses a sequential core ordering due to its algorithmic efficiency in the counting phase. We provide detailed analysis for tradeoffs involved with using different orderings and present a heuristic to select the best ordering for performance. We also present compact subgraph structures which significantly reduce memory consumption, improve locality, and improve the parallel scalability of the counting phase.

3. Actor-based Direction-Optimizing Breadth-First Search (Chapter 4). We leverage the Actor model to develop multiple Bottom-Up BFS implementations with different fine-grained messaging schemes. We evaluate the tradeoff between increasing parallelism by sending more messages to reduce latency, and algorithmic efficiency by sending fewer messages. We combine our most efficient Bottom-Up implementation with Top-Down BFS to achieve a performant hybrid. This is the first work to implement actor-based direction-optimizing BFS.

5.2 Future Work

The analysis presented in this thesis leads way to many interesting directions for future work on similar problems. These include applying our analysis techniques to improve performance of Graph Pattern Mining, as well as hardware support for sparse data processing.

Graph Pattern Mining

Future work in clique counting can take many directions based on the type of input graphs and hardware systems. Dynamic graphs present an additional set of challenges with the addition and removal of edges over time. Clique counting on dynamic graphs is a problem that has not been studied in prior literature. Since real-world data is rarely static over time, this is an important problem to consider. Due to the added time dimension in the dataset, storing intermediate data without significantly increasing memory consumption is a challenge. How the best ordering changes as the graph changes, and if recomputing the ordering improves performance are also important questions to consider.

Counting cliques on graphs that are too large to fit in a single compute node is also an interesting problem. Computationally intensive algorithms like clique counting are better suited to distributed-memory systems than algorithms like BFS, since there is enough algorithmic work to offset the cost of communication. However, scaling beyond a single compute node does not come without its challenges. Building subgraphs on distributed-memory systems requires a large amount of communication across low latency networks, and may require a significant amount of memory. Data placement for improving locality and load balance, and handling cases where even the induced subgraph is too large to fit in memory are additional challenges to consider.

Lastly, the ordering analysis and heuristic we present in Chapter 3 can easily be applied to other subgraph counting algorithms on real-world graphs.

Hardware Support for Sparse Data Processing

We show that optimizing data movement in software benefits performance. However, the benefits of those optimizations are still limited by the interaction between software and the underlying hardware. While sparse tensor accelerators exist [44, 51, 109], hardware support for data prefetching is missing from current literature. Prior hardware-software codesign solutions like Prodigy [111] show the potential for improving performance with more intelligent prefetching. It is interesting to think of the Power-Performance-Area tradeoffs for adding this feature in sparse tensor accelerators. Similarly, hardware support for Actors in distributed-memory systems can accelerate tasks like message aggregation, improving the performance of irregular applications like graph-processing.

Additional Tensor Optimizations

To improve MLP for MTTKRP by increasing the number of outstanding memory requests, we batch together phases of address generation and memory requests. Such a batching requires hardcoding certain loops for tensors with different dimensions. From our experiments, we find that hardcoded tensor codes perform better than flexible code that can handle any dimensional tensor which performs the same amount of algorithmic work. The hardcoded implementation potentially reduces some instruction overhead or benefits from compiler optimizations which enable additional performance. Understanding the performance gap may lead to opportunities for more software optimizations which can be applied to different sparse applications.

Bibliography

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[2] Santiago Aja-Fernández, Rodrigo de Luis Garcia, Dacheng Tao, and Xuelong Li. *Tensors in image processing and computer vision.* Springer Science & Business Media, 2009.

[3] George Almasi. PGAS (partitioned global address space) languages., 2011.

[4] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Parallel k-clique counting on gpus. In *International Conference on Supercomputing (ICS)*, pages 1–14, 2022.

[5] K. C. Dukka Bahadur, Tatsuya Akutsu, Etsuji Tomita, Tomokazu Seki, and Asao Fujiyama. Point matching under non-uniform distortions and protein side chain packing based on an efficient maximum clique algorithm. *Genome Informatics*, 13:143–152, 2002.

[6] Leonid Barenboim and Michael Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.

[7] Muthu Baskaran, Tom Henretty, Benoit Pradelle, M Harper Langston, David Bruns-Smith, James Ezick, and Richard Lethin. Memory-efficient parallel tensor decompositions. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[8] Scott Beamer. *Understanding and Improving Graph Algorithm Performance.* PhD thesis, University of California, Berkeley, 2016.

[9] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10. IEEE, 2012.

[10] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[11] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *2015 IEEE International Symposium on Workload Characterization*, pages 56–65. IEEE, 2015.

[12] Scott Beamer, Krste Asanović, and David Patterson. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 820–831. IEEE, 2017.

[13] Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1618–1627. IEEE, 2013.

[14] Maciej Besta, Armon Carigiet, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, and Torsten Hoefler. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–17. IEEE, 2020.

[15] Maciej Besta, Cesare Miglioli, Paolo Sylos Labini, Jakub Tětek, Patrick Iff, Raghavendra Kanakagiri, Saleh Ashkboos, Kacper Janda, Michał Podstawski, Grzegorz Kwaśniewski, et al. Probgraph: High-performance and high-accuracy graph mining with probabilistic set representations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–17. IEEE, 2022.

[16] Phillip Bonacich. Factoring and weighting approaches to status scores and clique identification. *Journal of mathematical sociology*, 2(1):113–120, 1972.

[17] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[18] Aydin Buluç, Scott Beamer, Kamesh Madduri, Krste Asanovic, and David Patterson. Distributed-memory breadth-first search on massive graphs. *arXiv preprint arXiv:1705.04590*, 2017.

[19] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2008.

[20] Aydın Buluç and John R Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications (IJHPCA)*, 25(4):496–509, 2011.

[21] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2011.

[22] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[23] Bibrak Qamar Chandio, Maciej Brodowicz, and Thomas Sterling. Structures and techniques for streaming dynamic graph processing on decentralized message-driven systems. In *Workshop Proceedings of the 53rd International Conference on Parallel Processing (ICPP)*, pages 1–6, 2024.

[24] Lijun Chang, Rashmika Gamage, and Jeffrey Xu Yu. Efficient k-clique count estimation with accuracy guarantee. *Proceedings of the VLDB Endowment*, 17(11):3707–3719, 2024.

[25] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: a two-level framework for efficient graph pattern mining. In *International Conference on Supercomputing (ICS)*, pages 378–391, 2021.

[26] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *VLDB*, 13(8):1190–1205, 2020.

[27] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1240–1248, 2012.

[28] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[29] Flavio Chierichetti, Ravi Kumar, and Bo Pang. On the power laws of language: Word frequency distributions. In *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval*, pages 385–394, 2017.

[30] Jee Choi, Xing Liu, Shaden Smith, and Tyler Simon. Blocking optimization techniques for sparse tensor computation. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 568–577. IEEE, 2018.

[31] Geir Dahl, Jon Magne Leinaas, Jan Myrheim, and Eirik Ovrum. A tensor product matrix approximation problem in quantum physics. *Linear algebra and its applications*, 420(2-3):711–725, 2007.

[32] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. In *World Wide Web Conference (WWW)*, pages 589–598, 2018.

[33] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[34] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 31–40, 2016.

[35] Wen Deng, Weiguo Zheng, and Hong Cheng. Accelerating maximal clique enumeration via graph reduction. *Proceedings of the VLDB Endowment*, 17(10):2419–3431, 2024.

[36] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *International Conference on Management of Data (MOD)*, pages 1357–1374, 2019.

[37] Nicholas Edmonds and Andrew Lumsdaine. The Parallel Boost Graph Library 2.0: Active messages as a spanning model for parallel graph computation. In *Massive Graph Analytics*, pages 459–481. Chapman and Hall/CRC, 2022.

[38] Youssef Elmougy, Akihiro Hayashi, and Vivek Sarkar. Highly scalable large-scale asynchronous graph processing using actors. In *International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, pages 242–248. IEEE, 2023.

[39] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 403–414. Springer, 2010.

[40] Paul Erdos, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. math. inst. hung. acad. sci*, 5(1):17–60, 1960.

[41] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks VS Lakshmanan, and Xuemin Lin. Efficient algorithms for densest subgraph discovery. *arXiv preprint arXiv:1906.00341*, 2019.

[42] Sofia Fernandes, Hadi Fanaee-T, and João Gama. Tensor decomposition for analysing time-evolving social networks: An overview. *Artificial Intelligence Review*, 54(4):2891–2916, 2021.

[43] Irene Finocchi, Marco Finocchi, and Emanuele G Fusco. Clique counting in mapreduce: Algorithms and experiments. *Journal of Experimental Algorithmics (JEA)*, 20:1–20, 2015.

[44] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–165, 2019.

[45] Michael T Goodrich and Paweł Pszona. External-memory network analysis algorithms for naturally sparse graphs. In *European Symposium on Algorithms (ESA)*, pages 664–676. Springer, 2011.

[46] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2(1), 2005.

[47] Enrico Gregori, Luciano Lenzini, and Simone Mainardi. Parallel k-clique community detection on large-scale networks. *Transactions on Parallel and Distributed Systems*, 24(8):1651–1660, 2013.

[48] Tanuj Gupta. Actor Graph Library. Master's thesis, University of California, Santa Cruz, Santa Cruz, USA, 2023.

[49] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[50] Fei Hao, Geyong Min, Zheng Pei, Doo-Soon Park, and Laurence T. Yang. k-clique community detection in social networks based on formal concept analysis. *Systems Journal*, 11(1):250–259, 2017.

[51] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–333, 2019.

[52] Ahmed E Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. ALTO: adaptive linearized storage of sparse tensors. In *Proceedings of the ACM International Conference on Supercomputing*, pages 404–416, 2021.

[53] Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance papers of the conference*, volume 3, page 235. Stanford Research Institute Menlo Park, CA, 1973.

[54] Frank L Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 6(1-4):164–189, 1927.

[55] Shams M Imam and Vivek Sarkar. Selectors: Actors with multiple guarded mailboxes. In *International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 1–14, 2014.

[56] Shweta Jain and C Seshadhri. The power of pivoting for exact clique counting. In *International Conference on Web Search and Data Mining (WSDM)*, pages 268–276, 2020.

[57] Shweta Jain and C Seshadhri. Provably and efficiently approximating near-cliques using the Turán shadow: PEANUTS. In *Proceedings of The Web Conference 2020*, pages 1966–1976, 2020.

[58] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: a pattern-aware graph mining system. In *European Conference on Computer Systems (EuroSys)*, pages 1–16, 2020.

[59] Aaron Jezghani, Semir Sarajlic, Michael Brandon, Neil Bright, Mehmet Belgin, Gergory Beyer, Christopher Blanton, Pam Buffington, J Eric Coulter, Ruben Lara, et al. Phoenix: The revival of research computing and the launch of the new cost model at Georgia Tech. In *Practice and Experience in Advanced Research Computing*, pages 1–9. 2022.

[60] Nishant Khanorkar. Enabling flexible data placement in the Actor Graph Library. Master's thesis, University of California, Santa Cruz, Santa Cruz, USA, 2023.

[61] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[62] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[63] Tamara G Kolda and Jimeng Sun. Scalable tensor decompositions for multi-aspect data mining. In *2008 Eighth IEEE international conference on data mining*, pages 363–372. IEEE, 2008.

[64] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[65] Kartik Lakhotia, Shreyas Singapura, Rajgopal Kannan, and Viktor Prasanna. Recall: Reordered cache aware locality based graph processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 273–282. IEEE, 2017.

[66] Victor E Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. A survey of algorithms for dense subgraph discovery. *Managing and mining graph data*, pages 303–336, 2010.

[67] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[68] Jiajia Li and Kevin Barker. PASTA: A parallel sparse tensor algorithm benchmark suite, Dec 2019.

[69] Jiajia Li, Mahesh Lakshminarasimhan, Xiaolong Wu, Ang Li, Catherine Olschanowsky, and Kevin Barker. A parallel sparse tensor benchmark suite on CPUs and GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 403–404, 2020.

[70] Jiajia Li, Yuchen Ma, Xiaolong Wu, Ang Li, and Kevin Barker. Pasta: a parallel sparse tensor algorithm benchmark suite. *CCF Transactions on High Performance Computing*, 1(2):111–130, 2019.

[71] Jiajia Li, Jimeng Sun, and Richard Vuduc. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '18, New York, NY, USA, 2018. ACM.

[72] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. Ordering heuristics for k-clique listing. *VLDB*, 2020.

[73] John DC Little and Stephen C Graves. Little's law. *Building intuition: Insights from basic operations management models and principles*, pages 81–100, 2008.

[74] Bangtian Liu, Chengyao Wen, Anand D Sarwate, and Maryam Mehri Dehnavi. A unified optimization approach for sparse tensor operations on GPUs. In *2017 IEEE international conference on cluster computing (CLUSTER)*, pages 47–57. IEEE, 2017.

[75] Amogh Lonkar and Scott Beamer. Accelerating clique counting in sparse real-world graphs via communication-reducing optimizations. *arXiv preprint arXiv:2112.10913*, 2021.

[76] Andrew Lumsdaine, Luke Dalessandro, Kevin Deweese, Jesun Firoz, and Scott McMillan. Triangle counting with cyclic distributions. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2020.

[77] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[78] Alan M Mainwaring and David E Culler. *Active Message applications programming interface and communication subsystem organization*. Citeseer, 1996.

[79] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[80] F Miller Maley and Jason G DeVinney. Conveyors for streaming many-to-many communication. In *Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 1–8. IEEE, 2019.

[81] S. Manoharan and Sathish. Patient diet recommendation system using k clique and deep learning classifiers. *Journal of Artificial Intelligence and Capsule Networks*, 2(2):121–130, 2020.

[82] Tobias Marschall, Ivan G. Costa, Stefan Canzar, Markus Bauer, Gunnar W. Klau, Alexander Schliep, and Alexander Schönhuth. CLEVER: clique-enumerating variant finder. *Bioinformatics*, 28(22):2875–2882, 10 2012.

[83] David W Matula and Leland L Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM (JACM)*, 30(3):417–427, 1983.

[84] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. *arXiv preprint arXiv:2006.06890*, 2020.

[85] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Cache-guided scheduling: Exploiting caches to maximize locality in graph processing. *AGP'17*, 2017.

[86] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the Graph 500. *Cray Users Group (CUG)*, 19(45-74):22, 2010.

[87] Mark EJ Newman. Assortative mixing in networks. *Physical review letters*, 89(20):208701, 2002.

[88] Andy Nguyen, Ahmed E Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Yongseok Soh, Teresa Ranadive, Fabrizio Petrini, and Jee W Choi. Efficient, out-of-memory sparse MTTKRP on massively parallel architectures. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pages 1–13, 2022.

[89] NVIDIA. NVIDIA TESLA V100 GPU ARCHITECTURE, 2017. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

[90] K. Xinchang P. Vilakone and D. Park. Personalized movie recommendation system combining data mining with the k-clique method. *Journal of Information Processing Systems*, 15(5):1141–1155, October 2019.

[91] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *nature*, 435(7043):814–818, 2005.

[92] Long Pan and Eunice E . Santos. An anytime-anywhere approach for maximal clique enumeration in social network analysis. In *International Conference on Systems, Man and Cybernetics (SMC)*, pages 3529–3535, 2008.

[93] Yannis Panagakis, Jean Kossaifi, Grigorios G Chrysos, James Oldfield, Mihalis A Nicolaou, Anima Anandkumar, and Stefanos Zafeiriou. Tensor methods in computer vision and deep learning. *Proceedings of the IEEE*, 109(5):863–890, 2021.

[94] Evangelos E Papalexakis, Christos Faloutsos, and Nicholas D Sidiropoulos. Tensors for data mining and data fusion: Models, applications, and scalable algorithms. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(2):1–44, 2016.

[95] J-S Park, Michael Penner, and Viktor K Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on parallel and distributed systems*, 15(9):769–782, 2004.

[96] Sri Raj Paul, Akihiro Hayashi, Kun Chen, Youssef Elmougy, and Vivek Sarkar. A fine-grained asynchronous bulk synchronous parallelism model for PGAS applications. *Journal of Computational Science*, 69:102014, 2023.

[97] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. A productive and scalable actor-based programming system for PGAS applications. In *Computational Science–ICCS 2022: 22nd International Conference, London, UK, June 21–23, 2022, Proceedings, Part I*, pages 233–247. Springer, 2022.

[98] Roger Pearce, Maya Gokhale, and Nancy M Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2010.

[99] Roger Pearce, Maya Gokhale, and Nancy M Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 825–836. IEEE, 2013.

[100] Eshrat Reghbati and Derek G. Corneil. Parallel computations in graph theory. *SIAM Journal on Computing*, 7(2):230–237, 1978.

[101] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[102] Ryan A Rossi, David F Gleich, and Assefaw H Gebremedhin. Parallel maximum clique algorithms with applications to network analysis. *Journal on Scientific Computing*, 37(5):C589–C616, 2015.

[103] Amnon Shashua and Tamir Hazan. Non-negative tensor factorization with applications to statistics and computer vision. In *Proceedings of the 22nd international conference on Machine learning*, pages 792–799, 2005.

[104] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. In *Conference on Applied and Computational Discrete Algorithms (ACDA21)*, pages 135–146. SIAM, 2021.

[105] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[106] Shaden Smith and George Karypis. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit. `http://cs.umn.edu/~splatt/`, 2016.

[107] Shaden Smith, Jongsoo Park, and George Karypis. Sparse tensor factorization on many-core processors with high-bandwidth memory. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1058–1067. IEEE, 2017.

[108] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 61–70. IEEE, 2015.

[109] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 689–702. IEEE, 2020.

[110] E. Tomita T. Matsunaga, C. Yonemori and M. Muramatsu. Clique-based data mining for related genes in a biomedical database. *BMC Bioinformatics*, 10(205), 2009.

[111] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 654–667. IEEE, 2021.

[112] Techpowerup. AMD EPYC Embedded 9554, 2024. `https://www.techpowerup.com/cpu-specs/epyc-embedded-9554.c3190`.

[113] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Symposium on Operating Systems Principles (SOSP)*, pages 425–440, 2015.

[114] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical computer science*, 363(1):28–42, 2006.

[115] Koji Ueno, Toyotaro Suzumura, Naoya Maruyama, Katsuki Fujisawa, and Satoshi Matsuoka. Efficient breadth-first search on massively parallel and distributed-memory machines. *Data Science and Engineering*, 2:22–35, 2017.

[116] Jeffrey Vetter and Chris Chambreau. mpip: Lightweight, scalable mpi profiling. 2005.

[117] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, volume 13, pages 3–6, 2013.

[118] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828, 2016.

[119] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[120] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. Lightning fast and space efficient k-clique counting. In *The Web Conference*, pages 1191–1202, 2022.

[121] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. Efficient k-clique counting on large graphs: The power of color-based sampling approaches. *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[122] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 25–25. IEEE, 2005.

[123] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017.

[124] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 301–316, 2016.