

Bridging Theory and Practice in Parallel Clustering

by

Jessica Shi

A.B., Princeton University (2018)

S.M., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer
Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

© 2023 Jessica Shi. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide,
irrevocable, royalty-free license to exercise any and all rights under
copyright, including to reproduce, preserve, distribute and publicly
display copies of the thesis, or release the thesis under an open-access
license.

Authored by: Jessica Shi

Department of Electrical Engineering and Computer Science
May 13, 2023

Certified by: Julian Shun

Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski

Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Bridging Theory and Practice in Parallel Clustering

by
Jessica Shi

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Large-scale graph processing is a fundamental tool in modern data mining, with wide-ranging applications in domains including social network analysis, bioinformatics, and machine learning. In particular, graph clustering, or community detection, is an important problem in graph processing that addresses tangible problems including fraud and threat detection, recommendation and search system design, and the detection of functional contributions of proteins and genes in biological systems. At its core, identifying the underlying substructures of a graph can indicate essential functional groups, such as people with similar interests, news articles on similar topics, or proteins with similar utilities, which can then be synthesized for a variety of applications. However, as the need to analyze larger and larger data sets increases, graph processing poses a major computational challenge, and designing scalable algorithms that can handle billions of edges while maintaining fast performance and high quality becomes crucial.

This thesis addresses the challenges of designing highly scalable graph clustering solutions by bridging theory and practice in parallel algorithms. The thesis takes a multi-faceted approach, where first, we develop algorithms with strong theoretical guarantees, which often translates to significant performance improvements in practice, and second, we use performance engineering techniques implemented on top of these theoretically efficient algorithms to achieve fast implementations on real-world data sets. The results are highly scalable and provably-efficient algorithms for a broad class of computationally graph clustering problems, and the first practical solutions to a number of problems on graphs with hundreds of billions of edges. Some of the implementations in this thesis are used in production environments in industry, and have significant real-world impacts.

The first part of this thesis studies the efficient counting and enumeration of small subgraphs, including small cycles and cliques, which has applications in clustering metrics and graph statistics. We design new theoretically efficient parallel algorithms for exact and approximate butterfly (four-cycle), five-cycle, and k -clique counting, and demonstrate significant performance improvements over the prior state-of-the-art small subgraph counting implementations. This part of the thesis also provides algorithms for low out-degree orientations, which are crucial as subroutines in our counting and enumeration algorithms to reduce the required work. Notably, we are

the first to report four-clique counts for the largest publicly available graph with over two hundred billion undirected edges. We also explore the batch-dynamic setting, in which graph properties are maintained over batches of multiple edge updates applied simultaneously, and present a novel parallel batch-dynamic data structure that we leverage in a wide variety of classic graph processing applications, including the k -core decomposition, low out-degree orientations, and k -clique counting. Importantly, our algorithm is the first parallel batch-dynamic algorithm for k -clique counting to achieve polylogarithmic span, or a polylogarithmic longest chain of sequential dependencies.

The second part of this thesis addresses a class of problems relating to the discovery and classification of dense substructures within a graph, focusing on hierarchical decompositions that reveal structural properties of the underlying graph with different notions or levels of density. We address bi-core decomposition and butterfly peeling, which are specialized algorithms for bipartite graphs, and we study k -clique peeling and nucleus decomposition, which generalize classic decomposition algorithms to higher order structures. This part leverages our subgraph counting algorithms to present new theoretically efficient parallel algorithms for these decomposition problems, and shows that many of these problems are P-complete, suggesting that solutions that take polylogarithmic span are unlikely. We also explore approximation algorithms as a result, and conduct thorough experimental evaluations comparing speed and accuracy trade-offs between our exact and approximate implementations.

The final part of this thesis focuses on highly scalable graph clustering algorithms that are effective in practice and give good quality clusters compared to ground truth data, considering a broad class of classification tasks. We study classic graph clustering algorithms including correlation clustering, modularity clustering, and hierarchical agglomerative clustering. This part develops heuristic and approximation algorithms for classic graph clustering objective functions, and additionally demonstrates important relationships between graph clustering algorithms and their counterparts in pointset clustering.

Thesis Supervisor: Julian Shun

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank the many people who have made this thesis possible. First, I am sincerely grateful to my advisor, Julian Shun, for his guidance and encouragement over the years. His support and mentorship have been invaluable throughout my Ph.D., and he has always made himself available no matter how insignificant my questions are. Julian was my introduction to parallel algorithms and programming, and he was truly the impetus behind this entire thesis. I have thoroughly enjoyed my Ph.D. experience largely because of him and the wonderful lab environment that he has fostered.

I am additionally grateful to all of the members of my thesis committee, for their support and valuable feedback to this thesis. I am incredibly grateful to Laxman Dhulipala, who has been a major source of inspiration and a fantastic collaborator ever since we met when he first came to MIT as a visiting student. He is a veritable fountain of ideas, and he has been an enormous part of many of the projects in this thesis. This thesis would not have been possible without his insights, advice, and dedication. I am very thankful for Vahab Mirrokni, who has been a great support during my time at Google. His vision and insights into the real-world problems faced in industry environments have been instrumental to the direction of this thesis, and I am tremendously grateful for his help and advice. I would also like to thank Ronitt Rubinfeld for her guidance and support. Her broad knowledge of theoretical graph algorithms have led to many insightful conversations, and I hope that we can collaborate on research projects in the future.

I am extremely grateful to Jakub Łącki, who has been my host and mentor during my many years as an intern and as a student researcher at Google, and who has been incredibly encouraging and helpful in launching and developing many research projects, some of which are in this thesis. Thank you for taking a chance on a student looking for an internship, and for all of the advice and hard work that have been indispensable to this thesis.

I would like to thank all of the members of my lab, past and present, who have made my time at MIT fun, joyous, and exciting. Thank you to Changwan Hong, Siddhartha Jayanti, Junhong Lin, Quanquan C. Liu, Tom Tseng, Yiqiu Wang, and Shangdi Yu, for all of the fantastic times that we have spent together eating lunch, playing games, and commiserating. A big thank you to Linda Lynch, our wonderful administrative assistant who has steadfastly solved all of our administrative problems, no matter how small. I would also like to extend a very special acknowledgment to Nami, who is a very cute poodle and an honorary lab member. Thank you to all of my other collaborators, Guy Blelloch, David Eisenstat, Louisa Ruixue Huang, Yihao Huang, Sofya Raskhodnikova, and Claire Wang, without whom this thesis would not exist. Their insightful contributions and hard work have been instrumental over the years.

I also thank the wonderful members of the OMEGA team at Google, who have collaborated with me on various projects and who have been great, engaging conversationalists in virtual and in-person team lunches throughout the years.

Thank you to Vincent Cohen-Addad, Mohammad Hossein Bateni, Xiaoen Ju, Sasan Tavakkol, David Applegate, Kevin Aydin, Matthew Fahrbach, and Lars Gottesbüren, for making Google a truly enjoyable place to work. I am thankful to the wonderful mentors I have had throughout my Ph.D, including my Graduate Women at MIT (GWAMIT) mentoring group members, Ylaine Gerardin and Michaela Gold, and my graduate counselor, Joel Emer, who have been extremely encouraging over the course of my Ph.D.

This research was supported in part by the National Science Foundation (NSF) under the Graduate Research Fellowship #1122374 and under grant numbers CCF-1845763, CCF-1910030, and CCF-1919223; the Department of Energy (DOE) under grant number DE-SC0018947; the Defense Advanced Research Projects Agency (DARPA) under grant number HR0011-18-3-0007; Google Faculty Research and Google Research Scholar Awards; the Massachusetts Institute of Technology (MIT) FinTech@CSAIL Initiative and Program for Research in Mathematics, Engineering, and Science for High School Students (PRIMES); cloud computing credits from Google-MIT; and the Applications Driving Architectures (ADA) Research Center, a Joint University Microelectronics Program (JUMP) Center co-sponsored by the Semiconductor Research Corporation (SRC) and DARPA. Thank you to all of the funding parties for supporting this work. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF, or other funding parties.

I am especially grateful to my amazing friends, who have stuck with me throughout this process, listened to my endless ramblings, and added much-needed levity, amusement, and joy to my life. A special thank you to Sarah Cen, Cathy Chen, Ivy Chen, Savannah Du, Albert Ho, Nathanael Ji, William Kuszmaul, Jackey Liu, Sara Liu, Matthew Lucas, Zachary Newman, Catherina Pan, Vanessa Phan, Katherine Pizano, Rose Silver, Kevin Sun, Caelin Tran, Nicholas Yang, Kristy Yeung, and Patrick Zeng, who have been positively delightful throughout my journey. I would also like to thank the many dogs on campus at MIT, who have unfailingly brought a smile to my face every day, and I would like to thank Taylor Swift, for releasing four whole albums during my Ph.D., which, long story short, have single-handedly gotten me through all of this writing.

I am deeply grateful to my fiancé, Heesu Hwang, who has been by my side this entire time, and whose unwavering love and encouragement has been undeniably the driving force behind my success. I would not be the person I am today without him. He has filled my days as a graduate student with laughter and happiness, and he has been my motivation in completing this Ph.D. I would also like to thank his parents and his sister, Heemyung Hwang, for welcoming me into the family and supporting me throughout these years. Finally, I am deeply appreciative and grateful to my parents, who have unconditionally loved and supported me my entire life. They have been an enormous source of inspiration, and they have been the foundation for all of my achievements over the course of my life. Their endless encouragement and counsel have been instrumental in shaping me to be who I am today, and this thesis would not have been completed without them. I would like to reserve a very special acknowledgment for Avery, the best, smartest, most well-behaved dog in the whole

world, who somehow occupied all of my time during my visits home and without whom this thesis would have been completed 5% faster.

To my fiancé, Heesu Hwang.

To my parents.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 37 |
| 1.1 | Subgraph Counting and Listing | 42 |
| 1.1.1 | Butterfly Counting | 42 |
| 1.1.2 | Five-cycle Counting | 43 |
| 1.1.3 | k -clique Counting and Listing | 44 |
| 1.1.4 | Batch-dynamic k -core Decomposition and Clique Counting | 46 |
| 1.2 | Subgraph Decomposition | 47 |
| 1.2.1 | Bi-core Decomposition | 47 |
| 1.2.2 | Butterfly Peeling | 48 |
| 1.2.3 | k -clique Densest Subgraph | 49 |
| 1.2.4 | Nucleus Decomposition | 50 |
| 1.2.5 | Nucleus Decomposition Hierarchy | 51 |
| 1.3 | Graph Clustering | 52 |
| 1.3.1 | Correlation Clustering | 53 |
| 1.3.2 | Hierarchical Agglomerative Clustering | 54 |
| 1.4 | Research Presented in This Thesis | 54 |
| 1.5 | Thesis Statement | 56 |
| 2 | Preliminaries and Notation | 59 |
| 2.1 | Graph Notation | 59 |
| 2.1.1 | Degeneracy and Arboricity | 59 |
| 2.1.2 | Other Graph Concepts | 60 |
| 2.1.3 | Graph Storage | 61 |
| 2.2 | Model of Computation | 61 |
| 2.2.1 | Atomic Operations | 61 |
| 2.3 | Parallel Primitives and Data Structures | 62 |
| 2.3.1 | Parallel Sequence Operations | 62 |
| 2.3.2 | Parallel Aggregation | 62 |
| 2.3.3 | Parallel Bucketing | 63 |
| 2.3.4 | Other Parallel Data Structures | 64 |
| 2.4 | Experimental Setup | 64 |

| | |
|---|-----------|
| I Subgraph Counting and Listing | 66 |
| 3 Butterfly Counting | 69 |
| 3.1 Introduction | 69 |
| 3.2 Preliminaries | 71 |
| 3.3 PARBUTTERFLY Framework | 71 |
| 3.3.1 Counting Framework | 71 |
| 3.3.1.1 Ranking | 72 |
| 3.3.1.2 Wedge Aggregation | 73 |
| 3.3.1.3 Butterfly Aggregation | 74 |
| 3.3.1.4 Other Options | 74 |
| 3.4 PARBUTTERFLY Algorithms | 75 |
| 3.4.1 Preprocessing | 75 |
| 3.4.2 Counting Algorithms | 76 |
| 3.4.2.1 Wedge Retrieval | 77 |
| 3.4.2.2 Per Vertex | 78 |
| 3.4.2.3 Per Edge | 79 |
| 3.4.3 Approximate Counting | 80 |
| 3.4.4 Approximate Degree Ordering | 81 |
| 3.4.5 Complement Degeneracy Ordering | 81 |
| 3.5 Experiments | 82 |
| 3.5.1 Environment | 82 |
| 3.5.2 Results | 82 |
| 3.5.2.1 Butterfly Counting | 82 |
| 3.5.2.2 Ranking | 84 |
| 3.5.2.3 Approximate Counting | 85 |
| 3.5.3 Cache Optimization for Butterfly Counting | 85 |
| 3.5.4 Butterfly Counting | 86 |
| 3.5.4.1 Ranking | 89 |
| 3.5.4.2 Approximate Counting | 90 |
| 3.6 Related Work | 92 |
| 3.7 Discussion | 93 |
| 4 Five-cycle Counting | 95 |
| 4.1 Introduction | 95 |
| 4.2 Background and Related Work | 96 |
| 4.3 Preliminaries | 97 |
| 4.4 Five-cycle Counting Algorithms | 98 |
| 4.4.1 Preprocessing: Graph Orientation | 98 |
| 4.4.2 Kowalik’s Algorithm | 99 |
| 4.4.3 ESCAPE Algorithm | 102 |
| 4.4.4 Approximate Counting | 105 |
| 4.4.4.1 Edge Sparsification | 105 |
| 4.4.4.2 Colorful Sparsification | 107 |
| 4.4.5 Implementation | 109 |

| | | |
|----------|---|------------|
| 4.5 | Experiments | 110 |
| 4.6 | Discussion | 118 |
| 5 | <i>k</i>-clique Counting and Listing | 121 |
| 5.1 | Introduction | 121 |
| 5.2 | Related Work | 123 |
| 5.3 | Clique Counting | 124 |
| 5.3.1 | Low Out-degree Orientation (Ranking) | 125 |
| 5.3.2 | Counting Algorithm | 127 |
| 5.3.3 | Sampling | 128 |
| 5.3.4 | Practical Optimizations | 129 |
| 5.4 | Experiments | 130 |
| 5.5 | Discussion | 136 |
| 6 | Batch-Dynamic <i>k</i>-core Decomposition and Clique Counting | 137 |
| 6.1 | Introduction | 137 |
| 6.2 | Preliminaries | 138 |
| 6.3 | Technical Overview | 140 |
| 6.4 | Comparisons with Other Related Work | 144 |
| 6.5 | Batch-Dynamic <i>k</i> -Core Decomposition | 146 |
| 6.5.1 | Algorithm Overview | 146 |
| 6.5.2 | Sequential Level Data Structure (LDS) | 147 |
| 6.5.3 | Detailed PLDS Algorithm | 147 |
| 6.5.4 | Estimating the Coreness and Orientation | 153 |
| 6.5.5 | $O(\alpha)$ Out-Degree Orientation | 154 |
| 6.6 | Experimental Evaluation | 154 |
| 6.6.1 | PLDS Implementation Details | 157 |
| 6.6.2 | Accuracy vs. Running Time | 158 |
| 6.6.3 | Batch Size vs. Running Time | 160 |
| 6.6.4 | Thread Count vs. Running Time | 160 |
| 6.6.5 | Results on Large Graphs | 161 |
| 6.6.6 | Accuracy of Approximation Algorithms | 163 |
| 6.6.7 | Sensitivity of PLDS and PLDSOpt to δ and λ | 163 |
| 6.6.8 | Space Usage | 164 |
| 6.7 | Static $(2 + \varepsilon)$ -Approximate <i>k</i> -Core | 164 |
| 6.8 | Framework for Batch-Dynamic Graph Algorithms from Low Out-Degree Orientations | 168 |
| 6.9 | Clique Counting | 170 |
| 6.9.1 | Algorithm Overview | 170 |
| 6.9.2 | BatchFlips Implementation | 173 |
| 6.9.3 | BatchInsert Implementation | 173 |
| 6.9.4 | BatchDelete Implementation | 176 |
| 6.9.5 | Correctness | 176 |
| 6.9.6 | Work and Span Analysis | 180 |
| 6.9.7 | Comparison with Previous Work | 181 |

| | |
|---------------------------|-----|
| 6.10 Discussion | 182 |
|---------------------------|-----|

II Subgraph Decomposition 183

| | |
|--|------------|
| 7 Bi-core Decomposition | 187 |
| 7.1 Introduction | 187 |
| 7.2 Related Work | 189 |
| 7.3 Preliminaries | 190 |
| 7.4 Parallel Bi-core Decomposition | 192 |
| 7.4.1 Background | 192 |
| 7.4.2 Parallel Bi-core Decomposition Algorithm | 192 |
| 7.4.3 Peeling Space Pruning Optimization | 196 |
| 7.5 P-completeness of Bi-core Decomposition | 197 |
| 7.6 Parallel Bi-core Index Structure | 200 |
| 7.6.1 Parallel Index Construction | 201 |
| 7.7 Experiments | 202 |
| 7.7.1 Experimental Setup | 203 |
| 7.7.2 Implementation and Other Optimizations | 203 |
| 7.7.3 Bi-core Decomposition | 204 |
| 7.7.4 Bi-core Index | 206 |
| 7.8 Discussion | 207 |
| 8 Butterfly Peeling | 209 |
| 8.1 Introduction | 209 |
| 8.2 Preliminaries | 210 |
| 8.3 PARBUTTERFLY Framework | 211 |
| 8.3.1 Peeling Framework | 211 |
| 8.4 PARBUTTERFLY Algorithms | 212 |
| 8.4.1 Peeling Algorithms | 212 |
| 8.4.1.1 Per Vertex | 214 |
| 8.4.1.2 Per Edge | 215 |
| 8.4.1.3 Per Vertex/Edge Storing All Wedges | 217 |
| 8.5 Parallel Fibonacci Heap | 218 |
| 8.5.1 Batch Insertion | 219 |
| 8.5.2 Delete-min | 219 |
| 8.5.3 Batch Decrease-key | 221 |
| 8.5.4 Application to Bucketing | 222 |
| 8.5.4.1 Retrieving the Minimum Bucket | 223 |
| 8.5.4.2 Updating the Bucketing Structure | 223 |
| 8.6 Experiments | 224 |
| 8.6.1 Environment | 224 |
| 8.6.1.1 Butterfly Peeling | 225 |
| 8.7 Related Work | 227 |
| 8.8 Discussion | 228 |

| | | |
|-----------|--|------------|
| 9 | <i>k</i>-clique Densest Subgraph | 229 |
| 9.1 | Introduction | 229 |
| 9.2 | Related Work | 230 |
| 9.3 | <i>k</i> -Clique Densest Subgraph | 230 |
| 9.3.1 | Vertex Peeling | 231 |
| 9.3.2 | Approximate Vertex Peeling | 234 |
| 9.3.3 | Practical Optimizations | 235 |
| 9.3.4 | Parallel Complexity of c -($1, k$)-nucleii | 235 |
| 9.4 | Experiments | 237 |
| 9.5 | Discussion | 242 |
| 10 | Nucleus Decomposition | 243 |
| 10.1 | Introduction | 243 |
| 10.2 | Related Work | 244 |
| 10.3 | Preliminaries | 245 |
| 10.4 | (r, s) Nucleus Decomposition | 246 |
| 10.4.1 | Recursive s -clique Counting Algorithm | 246 |
| 10.4.2 | (r, s) Nucleus Decomposition Algorithm | 246 |
| 10.4.3 | Discussion of Theoretically Efficient Bounds | 252 |
| 10.5 | Practical Optimizations | 255 |
| 10.5.1 | Number of Parallel Hash Table Levels | 255 |
| 10.5.2 | Contiguous Space | 257 |
| 10.5.3 | Binary Search vs. Stored Pointers | 258 |
| 10.5.4 | Graph Relabeling | 260 |
| 10.5.5 | Obtaining the Set of Updated r -cliques | 261 |
| 10.5.6 | Graph Contraction | 262 |
| 10.6 | Evaluation | 262 |
| 10.6.1 | Environment and Graph Inputs | 262 |
| 10.6.2 | Tuning Optimizations | 263 |
| 10.6.3 | Performance | 268 |
| 10.7 | Discussion | 271 |
| 11 | Nucleus Decomposition Hierarchy | 273 |
| 11.1 | Introduction | 273 |
| 11.2 | Related Work | 275 |
| 11.3 | Preliminaries | 276 |
| 11.4 | Nucleus Decomposition Hierarchy | 276 |
| 11.5 | Approximate Nucleus Decomposition | 282 |
| 11.6 | Practical Implementations | 287 |
| 11.6.1 | Interleaved Hierarchy Framework | 287 |
| 11.6.2 | Basic Version of LINK | 289 |
| 11.6.3 | Efficient Version of LINK | 290 |
| 11.6.4 | Practical Version of ARB-NUCLEUS-HIERARCHY | 297 |
| 11.7 | Evaluation | 297 |
| 11.7.1 | Environment and Graph Inputs | 297 |

| | |
|--|------------|
| 11.7.2 Comparison of ANH-TE, ANH-EL, and ANH-BL | 299 |
| 11.7.3 Performance of Exact Hierarchy | 300 |
| 11.7.4 Performance of Approximate Hierarchy | 302 |
| 11.8 Discussion | 303 |
| III Graph Clustering | 304 |
| 12 Correlation Clustering | 307 |
| 12.1 Introduction | 307 |
| 12.2 Preliminaries | 308 |
| 12.3 Algorithm and Optimizations | 309 |
| 12.3.1 Sequential Louvain Method | 309 |
| 12.3.2 Parallel Louvain Method and Optimizations | 311 |
| 12.3.2.1 Optimization: Synchronous vs Asynchronous | 312 |
| 12.3.2.2 Optimization: All Vertices vs Neighbors of Clusters vs Neighbors of Vertices | 313 |
| 12.3.2.3 Optimization: Multi-level Refinement | 314 |
| 12.3.2.4 Other Optimizations | 314 |
| 12.3.3 P-completeness of Louvain | 315 |
| 12.4 Experiments | 318 |
| 12.4.1 Tuning Optimizations | 320 |
| 12.4.2 Speedups and Scalability | 323 |
| 12.4.3 Quality Compared to Ground Truth | 329 |
| 12.5 Discussion | 332 |
| 13 Hierarchical Agglomerative Clustering | 333 |
| 13.1 Introduction | 333 |
| 13.1.1 Related Work | 336 |
| 13.2 Preliminaries | 337 |
| 13.2.1 Graph-Based HAC | 337 |
| 13.3 Sequential HAC | 338 |
| 13.4 Parallel Approximate HAC | 339 |
| 13.4.1 Overview | 339 |
| 13.4.2 ParHAC Algorithm | 341 |
| 13.5 ParHAC Implementation | 346 |
| 13.6 Empirical Evaluation | 348 |
| 13.6.1 Quality | 351 |
| 13.6.2 Scalability on Large Real-World Graphs | 353 |
| 13.6.3 Pointset Clustering: End-to-End Evaluation | 356 |
| 13.7 Discussion | 357 |

| | |
|--|------------|
| IV Conclusion and Future Directions | 358 |
| 14 Conclusion and Future Work | 359 |
| 14.1 Conclusion | 359 |
| 14.2 Future Work | 360 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | Running times in seconds for (3, 4) nucleus decomposition on a 30-core machine with two-way hyper-threading of Sariyüce <i>et al.</i> 's [284] sequential implementation, Sariyüce <i>et al.</i> 's [283] parallel implementation, our parallel theoretically efficient implementation, and our parallel theoretically efficient implementation with practical optimizations. Note that the number of edges in each graph is labeled underneath the corresponding graph. The running times are given in log-scale. | 39 |
| 1.2 | An overview of my body of work, including research presented in this thesis. Results for various topics are divided based on whether they contribute to theory, systems, or both. | 41 |
| 3.1 | The butterflies in this graph are $\{u_1, v_1, u_2, v_2\}$, $\{u_1, v_1, u_2, v_3\}$, and $\{u_1, v_2, u_2, v_3\}$. The red, blue, and green edges each produce a wedge ($\{u_1, v_1, u_2\}$, $\{u_1, v_2, u_2\}$, and $\{u_1, v_3, u_2\}$). Note that these wedges all share the same endpoints, namely u_1 and u_2 . Thus, any combination of two of these wedges forms a butterfly. For example, $\{u_1, v_1, u_2\}$ and $\{u_1, v_2, u_2\}$ combine to form a butterfly $\{u_1, v_1, u_2, v_2\}$. However, the dashed edges produce another wedge, $\{u_2, v_3, u_3\}$, which has different endpoints, namely u_2 and u_3 . This wedge cannot be combined with any of the previous wedges to form a butterfly. | 70 |
| 3.2 | The PARBUTTERFLY framework for counting. | 72 |
| 3.3 | We execute butterfly counting per vertex on the graph in Figure 3.1. In Step 1, we rank vertices in decreasing order of degree. In Step 2, for each vertex v in order, we retrieve all wedges where v is an endpoint and where the other two vertices have higher rank (the wedges are represented as $((x, z), y)$ where x and z are endpoints and y is the center). In Step 3, we aggregate wedges by their endpoints, and this produces the butterfly counts for Step 4. Note that if we have w wedges that share the same endpoint, this produces $(^w_2)$ butterflies for each of the two endpoints and $w - 1$ butterflies for each of the centers of the w wedges. | 72 |

| | | |
|------|--|----|
| 3.4 | These are the parallel runtimes for butterfly counting per vertex, considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking, and \diamond refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses. | 84 |
| 3.5 | These are the parallel runtimes for butterfly counting per edge, considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking, and \diamond refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses. | 85 |
| 3.6 | These are the parallel runtimes for butterfly counting in total, considering different wedge aggregation methods (butterfly aggregation does not apply). We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking, and \diamond refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses. | 86 |
| 3.7 | These are the runtimes for butterfly counting per vertex on livejournal using side ranking, over different numbers of threads. The self-relative speedups are between 13.7–28.3x. | 86 |
| 3.8 | These are the runtimes for butterfly counting per edge on livejournal using approximate degree ranking, over different numbers of threads. The self-relative speedups are between 15.9–38.0x. | 86 |
| 3.9 | These are the runtimes for butterfly counting per vertex, considering different rankings. We use simple batching as our wedge aggregation method. All times are scaled by the fastest runtime, as indicated in parentheses. Moreover, the time taken to rank each graph is included in the runtimes. | 87 |
| 3.10 | These are the runtimes for colorful sparsification and edge sparsification over different probabilities p . We considered both the runtimes on 48 cores hyperthreaded and on a single thread. We ran these algorithms on orkut, using simple batch aggregation and side ranking. | 89 |
| 3.11 | These are the parallel runtimes for butterfly counting per vertex (using the cache optimization), considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, \diamond refers to degree ranking, and \circ refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses. | 89 |

- 3.12 These are the parallel runtimes for butterfly counting per edge (using the cache optimization), considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, \diamond refers to degree ranking, and \circ refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses. 90
- 3.13 These are the parallel runtimes for butterfly counting in total (using the cache optimization), considering different wedge aggregation methods (butterfly aggregation does not apply). We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking, and \circ refers to approximate degree ranking. All times are scaled by the fastest parallel time, as parentheses. 90
- 3.14 These are the runtimes for butterfly counting per vertex on livejournal using side ranking and using the cache optimization, over different numbers of threads. The self-relative speedups are between 14.1–37.4x. 91
- 3.15 These are the runtimes for butterfly counting per edge on livejournal using approximate degree ranking and using the cache optimization, over different numbers of threads. The self-relative speedups are between 9.8–38.9x. 91
- 3.16 These are the parallel runtimes for butterfly counting per vertex (using the cache optimization), considering different rankings. We use simple batching as our wedge aggregation method. All times are scaled by the fastest parallel time, as indicated in parentheses. Moreover, the time taken to rank each graph is included in the runtimes. 91
- 3.17 These are the runtimes for colorful sparsification and edge sparsification over different probabilities p (using the cache optimization). We considered both the runtimes on 48 cores hyperthreaded and on a single thread. We ran these algorithms on orkut, using simple batch aggregation and side ranking. 92

| | | |
|------|--|-----|
| 4.5 | Running time of the exact parallel Kowalik algorithm vs. number of threads. “36h” is 36 cores with hyper-threading. Dashed lines indicate that the work scheduling optimization is disabled and solid lines indicate that the work scheduling optimization is enabled. The lines for Goodrich-Pszona, Barenboim-Elkin, and degree ordering overlap each other for the most part. | 115 |
| 4.6 | Exact five-cycle counting times using our parallel Kowalik implementation, excluding preprocessing steps like relabeling and orienting the graph, under different orientation schemes for each of the graphs, using 36 cores with hyper-threading. The fastest time is labeled under each graph. | 115 |
| 4.7 | Breakdown of time spent in the exact parallel Kowalik implementation on preprocessing (T_p) vs. counting (T_c) for different orientation subroutines, using 36 cores with hyper-threading. G-P is Goodrich-Pszona; B-E is Barenboim-Elkin. For the orkut graph, the time spent on preprocessing is not visible. | 116 |
| 4.8 | These are the parallel speedups over exact running times for approximate five-cycle counting using edge sparsification, varying over different p . The best runtimes considering all orientations were used. The speedups are given in a log-scale. | 116 |
| 4.9 | These are the parallel speedups over exact running times for approximate five-cycle counting using colorful sparsification, varying over $p = 1/c$, where c is the number of colors used. The best runtimes considering all orientations were used. The speedups are given in a log-scale. | 116 |
| 4.10 | These are the fractional errors of the approximate five-cycle count obtained using edge sparsification, varying over different p . The errors are taken over all graphs, with the red circle marking the median error and the bars marking the minimum and maximum errors. | 117 |
| 4.11 | These are the fractional errors of the approximate five-cycle count obtained using colorful sparsification, varying over $p = 1/c$, where c is the number of colors used. The errors are taken over all graphs, with the red circle marking the median error and the bars marking the minimum and maximum errors. | 117 |
| 4.12 | These are the self-relative speedups of approximate five-cycle counting over the single-threaded running times, using edge sparsification with $p = 1/8$ over varying numbers of workers. “36h” refers to 36 cores with hyper-threading. The Goodrich-Pszona orientation was used. | 118 |
| 4.13 | These are the self-relative speedups of approximate five-cycle counting over the single-threaded running times, using colorful sparsification with 8 colors ($p = 1/8$) over varying numbers of workers. “36h” refers to 36 cores with hyper-threading. The Goodrich-Pszona orientation was used. | 118 |

| | | |
|-----|---|-----|
| 5.1 | Parallel runtimes for k -clique counting (COUNT-CLIQUEs) on com-orkut, considering different orientations using node parallelism. Note that all times are scaled by the fastest parallel runtime, as indicated in parentheses, and the first set of bars indicate the preprocessing overhead of the different orientations. The rest of the k -clique counting runtimes include time spent preprocessing and counting. | 134 |
| 5.2 | These are the parallel runtimes for k -clique counting (COUNT-CLIQUEs) on com-orkut, considering node parallelism and edge parallelism, and fixing the orientation given by degree ordering. Note that all times are scaled by the fastest parallel runtime, as indicated in parentheses. . . | 135 |
| 5.3 | These are the parallel runtimes for approximate k -clique counting (COUNT-CLIQUEs) using colorful sparsification on com-orkut, varying over $p = 1/c$ where c is the number of colors used. Note that these runtimes were obtained using the orientation given by degree ordering and node parallelism. The runtimes are given in a log-scale. | 135 |
| 5.4 | These are the parallel runtimes for approximate k -clique counting (COUNT-CLIQUEs) using colorful sparsification on com-friendster, varying over $p = 1/c$ where c is the number of colors used. Note that these runtimes were obtained using the orientation given by degree ordering and node parallelism. The runtimes are given in a log-scale. | 135 |
| 6.1 | Exact k -core decomposition (left) and $(3/2)$ -approximate k -core decomposition (right). | 139 |
| 6.2 | This figure shows what parts of the PLDS are used in each result. The level of each vertex is used to determine the k -core decomposition (Theorem 6.1) and low out-degree orientation (Theorem 6.2 and Corollary 6.1). The orientation of the edges is used for maximal matching (Theorem 6.3), implicit $O(2^\alpha)$ -coloring (Theorem 6.4), and k -clique counting (Theorem 6.5). Finally, both are used for $O(\alpha \log n)$ -coloring (Theorem 6.6). | 144 |
| 6.3 | Previous best-known sequential algorithm results. | 145 |
| 6.4 | Example of a cascade of vertex movements caused by an edge deletion on u (shown by the dashed red line). | 147 |
| 6.5 | Example of invariants maintained by the PLDS for $\delta = 0.4$ and $\lambda = 3$. There are $\Theta(\log n)$ groups, each with $\Theta(\log n)$. Each vertex is in exactly one level of the structure and moves up and down by some movement rules. For example, vertex x (blue) is on level 3 and in group 1. . . . | 148 |
| 6.6 | Example of RebalanceInsertions described in the text for $\delta = 0.4$ and $\lambda = 3$. The red lines represent the batch of edge insertions. | 150 |
| 6.7 | Example of RebalanceDeletions described in the text for $\delta = 1$ and $\lambda = 3$. The red dotted lines represent the batch of edge deletions. . . | 152 |

| | |
|---|-----|
| 6.8 Comparison of the average per-batch time versus the average (top row) and maximum (bottom row) per-vertex core estimate error ratio of PLDSOpt, PLDS, Sun, and LDS, using varying parameters, on the <i>dblp</i> and <i>livejournal</i> graphs, with batch sizes 10^5 and 10^6 , respectively. Experiments were run for Ins , Del , and Mix . The data uses theoretically-efficient parameters as well as the heuristic parameters where $(2 + 3/\lambda) = \alpha_{sun} = 1.1$. Runtimes for Hua and Zhang are shown as horizontal lines. | 158 |
| 6.9 Average Ins , Del , and Mix per-batch running times on varying batch sizes for PLDSOpt, PLDS, LDS, Zhang, and Hua on <i>dblp</i> and <i>livejournal</i> | 159 |
| 6.10 Parallel speedup of PLDSOpt, PLDS, and Hua, with respect to their single-threaded running times on <i>dblp</i> and <i>livejournal</i> on Ins , Del , and Mix batches of size 10^6 for all algorithms. The “60” on the x -axis indicates 30 cores with hyper-threading. LDS, Sun, and Zhang are shown as horizontal lines since they are sequential. | 160 |
| 6.11 Average per-batch running times for PLDSOpt, Hua, PLDS, Sun, Zhang, ApproxKCore, and ExactKCore, on <i>dblp</i> , <i>youtube</i> , <i>wiki</i> , <i>ctr</i> , <i>usa</i> , <i>stack-overflow</i> , <i>livejournal</i> , <i>orkut</i> , <i>brain</i> , <i>twitter</i> , and <i>friendster</i> with batches of size 10^6 (and approximation settings $\delta = 0.4$ and $\lambda = 3$ for PLDSOpt and PLDS). All benchmarks (except PLDSOpt and PLDS) timed out (T.O.) at 3 hours for <i>twitter</i> and <i>friendster</i> for Ins and Del . Hua and Sun timed out on <i>twitter</i> and <i>friendster</i> for Mix . The top graph shows insertion-only, middle graph shows deletion-only, and bottom graph shows mixed batch runtimes. | 162 |
| 6.12 Sensitivity analysis of PLDSOpt and PLDS on <i>livejournal</i> . The first three plots fix δ ; each line is a fixed δ value and each point is a different λ value. The last three plots fix λ and vary δ | 163 |
| 6.13 Maximum space usage in bytes for PLDSOpt, Hua, Zhang, PLDS, and Sun in terms of the average error. We varied δ and λ and computed the error ratio and space usage for the programs on <i>dblp</i> and <i>livejournal</i> . We tested against Ins and Del batches of size 10^5 for <i>dblp</i> and batches of size 10^6 for <i>livejournal</i> | 165 |
| 6.14 Example of a run of Algorithm 6.6 described in Theorem 6.8. | 167 |

| | |
|--|-----|
| 6.15 Example of incomplete cliques in our counting algorithm for counting $k = 5$ cliques. In (1), c is the source of a potential 5-clique. $\{a, b, c, d, e\}$ represents a potential 5-clique. We do not yet know the source of the 4-clique consisting of $\{a, b, d, e\}$ (the purple edges represent potential edges), and so we store $\{a, b, d, e\}$ in table I_4 . (2) shows a set of edge insertions (indicated by the red edges) which determines a source (e) for the 4-clique. Thus, we insert $\{a, b, d\}$ in table I_3 . Suppose that this is the only clique that would be counted when edges are inserted between all pairs in $\{a, b, d\}$. Thus, we associate with this key, a count of 1 in table I_3 . Finally, (3) shows two edge insertions which completes the triangle; hence, we count the clique using the key $\{a, b, d\}$ and increment the k -clique count using the count associated with it (in this example, the count is 1) in table I_3 . | 172 |
| 7.1 An example peeling process of PAR-PEEL-FIX- β where the input β' is 2. From Step 0 to Step 1, all vertices in V with induced degree < 2 are removed. From Step 1 to Step 2, all vertices in U with degree ≤ 2 are removed. | 195 |
| 7.2 This shows the peeling space of the example graph in Figure 7.1, and is discussed in more detail in Section 7.4.3. | 197 |
| 7.3 This figure compares the unoptimized and optimized peeling paths of the example graph in Figure 7.1. The top figure shows the unoptimized peeling paths, while the bottom figure shows the optimized peeling paths. | 198 |
| 7.4 This figure shows the index structure \mathbb{PI}^V for the example graph in Figure 7.1. The first level is indexed by β values, and the second level is indexed by α values, which then point to the corresponding set of vertices in the core. The lines between levels represent pointers in the structure. | 200 |
| 7.5 This figure compares the running time (in seconds) of various bi-core decomposition algorithms, namely LIU-PAR, LIU-SEQ, and PAR. LIU-PAR on 60 hyper-threads runs out of memory for the Orkut and Web Trackers graphs, hence the missing bars. However, LIU-PAR is able to finish running on all graphs on 30 threads. | 205 |
| 7.6 This figure compares the parallel self-relative speedups of PAR (in black) and LIU-PAR (in green) over different numbers of threads. LIU-PAR on 60 hyper-threads runs out of memory for the Orkut and Web Trackers graphs. Also, “60h” stands for 60 hyper-threads. | 205 |
| 7.7 This figure shows the running times of Liu et al.’s sequential index construction algorithm, LIU-CONS and our parallel index construction algorithm, IND-CONS. | 206 |
| 7.8 This figure shows the parallel self-relative speedup of IND-CONS over different numbers of threads. Note that “60h” stands for 60 hyper-threads. | 206 |

| | | |
|------|---|-----|
| 7.9 | This graph shows the running times of Liu et al.’s sequential query algorithm, LIU-QUERY, and our parallel query algorithm, QUERY, on a batch of 10,000 queries. | 207 |
| 7.10 | This graph shows the parallel self-relative speedups of PAR-QUERY over different numbers of threads, running on a batch of 10,000 queries. Note that “60h” stands for 60 hyper-threads. | 207 |
| 8.1 | The PARBUTTERFLY framework for peeling. | 211 |
| 8.2 | These are the parallel runtimes for butterfly vertex peeling with different wedge aggregation methods (these runtimes do not include the time taken to count butterflies). All times are scaled by the fastest parallel time, as indicated in parentheses. Also, note that the runtimes for discogs_style represent single-threaded runtimes; this is because we did not see any parallel speedups for discogs_style, due to the small number of vertices that were peeled. | 225 |
| 8.3 | These are the parallel runtimes for butterfly edge peeling with different wedge aggregation methods (these runtimes do not include the time taken to count butterflies). All times are scaled by the fastest parallel time, as indicated in parentheses. | 225 |
| 9.1 | Gadget used for k -(1, 3) nucleus reduction. Note that the figure uses the notation k -(r, s) to refer to nuclei, whereas the text uses the notation c -(r, k). The red values show the initial clique-degree (K_s degree in this terminology, K_k degree in the main text) of each vertex before peeling. | 235 |
| 9.2 | These are the frequencies of the number of vertices peeled in a parallel round using PEEL-CLIQUEs, for k -clique peeling on com-orkut ($4 \leq k \leq 6$). Note that rounds with more than 1000 vertices peeled have been truncated; these truncated round frequencies are very low, most often consisting of 0 rounds. Also, note that the frequencies are given in a log scale. | 239 |
| 9.3 | Parallel runtimes for approximate k -clique peeling using APPROX-PEEL-CLIQUEs (solid lines) and KCLIST (dashed lines). These runtimes were obtained on com-orkut, varying over ϵ , giving a $1/(k(1 + \epsilon))$ -approximation of the k -clique densest subgraph. Note that these runtimes were obtained using the orientation given by degree ordering, and the runtimes are given in a log scale. Moreover, we cut off KCLIST’s runtimes at 5 hours, which occurred for $k = 10$ over all ϵ | 239 |
| 9.4 | These are the parallel runtimes for approximate k -clique peeling using APPROX-PEEL-CLIQUEs (solid lines) and KCLIST (dashed lines). These runtimes were obtained on com-friendster, varying over ϵ , giving a $1/(k(1 + \epsilon))$ -approximation of the k -clique densest subgraph. Note that these runtimes were obtained using the orientation given by degree ordering, and the runtimes are given in a log scale. Moreover, we cut off KCLIST’s runtimes at 5 hours, which occurred for $k = 9$ over all ϵ . | 239 |

| | |
|---|-----|
| 10.1 An example of our parallel nucleus decomposition algorithm ARB-NUCLEUS for $(r, s) = (3, 4)$. At each step, we peel in parallel all triangles (3-cliques) with the minimum 4-clique count; the vertices and edges that compose of these triangles are highlighted in red. We then recompute the 4-clique count on the remaining triangles. Vertices and edges that no longer participate in any active triangles, due to previously peeled triangles, are shown in gray. Each step is labeled with the k - $(3,4)$ nucleus discovered, where k is the 4-clique count of the triangles highlighted in red. The figure below labels each k - $(3,4)$ nucleus. | 247 |
| 10.2 An example of the data structures in ARB-NUCLEUS during each round of $(3, 4)$ nucleus decomposition, on the graph in Figure 10.1. | 248 |
| 10.3 An example of the initial parallel hash table T for $(3, 4)$ nucleus decomposition on the graph from Figure 10.1, considering different numbers of levels. Note that Figure 10.2 shows a one-level parallel hash table T . If we consider each vertex and each pointer to take a unit of memory, the one-level T takes 42 units, while the two-level T takes 35 units, thus saving memory. However, the 3-multi-level T takes 50 units, because $r = 3$ is too small to give memory savings for this graph. | 256 |
| 10.4 An example of the initial parallel hash table T for $(4, 5)$ nucleus decomposition on the graph from Figure 10.1, considering different numbers of levels. If we consider each vertex and each pointer to take a unit of memory, the one-level T takes 24 units, while the 3-multi-level T takes 22 units, thus saving memory. We see memory savings with more levels in T compared to in Figure 10.3, because $r = 4$ is sufficiently large. | 257 |
| 10.5 Using the same graph as shown in Figure 10.1 and the two-level T from Figure 10.3, for $(3, 4)$ nucleus decomposition, an example of the binary search and stored pointers methods. | 258 |
| 10.6 An example of the simple array, list buffer, and hash table options in aggregating the set U of r -cliques with updated s -clique counts. Processors P_1 , P_2 , and P_3 are storing r -cliques C_{r_1} , C_{r_2} , and C_{r_3} , respectively, in U | 259 |
| 10.7 Sizes of our input graphs, all of which are from [207], on the left, and the number of rounds required $\rho_{(r,s)}$ and the maximum (r, s) -core numbers for $r < s \leq 7$ for each graph on the right. | 263 |
| 10.8 On the left, multiplicative speedups of different combinations of optimizations on T in ARB-NUCLEUS, over an unoptimized setting of ARB-NUCLEUS, for $(3, 4)$ nucleus decomposition. On the right, multiplicative space savings for T of different combinations of optimizations on T in ARB-NUCLEUS, over the unoptimized setting, for $(3, 4)$ nucleus decomposition. Note that the space usage between the non-contiguous option and the contiguous option is equal. Friendster is omitted because ARB-NUCLEUS runs out of memory for this graph. | 264 |

| | |
|---|-----|
| 10.9 Multiplicative speedups of different combinations of optimizations on T in ARB-NUCLEUS, over an unoptimized setting of ARB-NUCLEUS, for (4, 5) nucleus decomposition. Livejournal, orkut, and friendster are omitted, because ARB-NUCLEUS runs out of memory for these instances. | 264 |
| 10.10 Multiplicative space savings for T of different combinations of optimizations on T in ARB-NUCLEUS, over the unoptimized setting, for (4, 5) nucleus decomposition. Note that the space usage between the non-contiguous option and the contiguous option is equal. Also, livejournal, orkut, and friendster are omitted, because ARB-NUCLEUS runs out of memory for these instances. | 265 |
| 10.11 Multiplicative speedups of the graph relabeling, update aggregation, and graph contraction optimizations in ARB-NUCLEUS, over a two-level setting with contiguous space and stored pointers, and using the simple array for U . Friendster is omitted from the (2, 4) and (3, 4) nucleus decomposition experiments, because the unoptimized ARB-NUCLEUS times out for (2, 4), and ARB-NUCLEUS runs out of memory for (3, 4). | 265 |
| 10.12 Multiplicative slowdowns over our parallel ARB-NUCLEUS of PKT-OPT-CPU and PKT for (2, 3) nucleus decomposition, and of single-threaded ARB-NUCLEUS, PND, AND, AND-NN, and ND for (2, 3) and (3, 4) nucleus decomposition. The label ARB-NUCLEUS in the legend refers to our single-threaded running times. We have omitted bars for PND, AND, AND-NN, and ND where these implementations run out of memory or time out. We have included in parentheses the times of our parallel ARB-NUCLEUS on 30 cores with hyper-threading. We have also included a line marking a multiplicative slowdown of 1 for $r = 2$, $s = 3$, and we see that PKT-OPT-CPU outperforms ARB-NUCLEUS on skitter, livejournal, orkut, and friendster. | 266 |
| 10.13 Multiplicative slowdowns of parallel ARB-NUCLEUS for each (r, s) combination over the fastest running time for parallel ARB-NUCLEUS across all $r < s \leq 7$ for each graph (excluding (2, 3) and (3, 4), which are shown in Figure 10.12). The fastest running time is labeled in parentheses below each graph. Also, livejournal is excluded because for these r and s , ARB-NUCLEUS is only able to complete (2, 4) nucleus decomposition, in 484.74 seconds, and the rest timed out. We have omitted bars where ARB-NUCLEUS runs out of memory or times out. | 267 |
| 10.14 Speedup of ARB-NUCLEUS over its single-threaded running times. "30h" denotes 30-cores with two-way hyper-threading. | 268 |
| 10.15 Running times of parallel ARB-NUCLEUS on rMAT graphs of varying densities for (2, 3), (3, 4), and (4, 5) nucleus decomposition. We remove duplicate generated edges. | 268 |
| 11.1 An example of the (1, 2) nucleus (k -core) hierarchy. | 276 |

| | |
|---|-----|
| 11.2 An example of the L_i data structures maintained by ARB-NUCLEUS-HIERARCHY while computing the k -core hierarchy on the graph in Figure 11.1. For each round i of the hierarchy construction, the connected components of H and the ID_i table used to construct H is shown (except ID_4 , which maps every r -clique to itself). | 278 |
| 11.3 An example of the intermediate hierarchy trees T after each round i in ARB-NUCLEUS-HIERARCHY, while computing the k -core hierarchy on the graph in Figure 11.1. | 278 |
| 11.4 An example of the uf and L data structures maintained by LINK-EFFICIENT when computing the k -core hierarchy on the graph in Figure 11.1. The data structures are shown after the third and fourth rounds of peeling in ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK, and after intermediate calls to LINK-EFFICIENT within the fourth round. | 292 |
| 11.5 An example of the k -core hierarchy tree on the graph in Figure 11.1, constructed by CONSTRUCT-TREE-EFFICIENT. Stage 1 shows an intermediate tree constructed in the CONSTRUCT-TREE-EFFICIENT subroutine, and Stage 2 depicts the final hierarchy tree. | 293 |
| 11.6 Multiplicative slowdowns of our parallel nucleus decomposition hierarchy implementations ANH-TE, ANH-EL, and ANH-BL, over the fastest of the three for each graph, for $r < s \leq 5$. We have omitted bars where our implementations run out of memory or time out after 4 hours, and we have omitted graphs where only one of ANH-TE, ANH-EL, and ANH-BL completes. Below each graph in parentheses is the fastest running time among the three implementations. We have also included a line marking a multiplicative slowdown of 1. | 298 |
| 11.7 Multiplicative slowdowns of parallel ARB-NUCLEUS-HIERARCHY for each (r, s) combination over the fastest running time for parallel ARB-NUCLEUS-HIERARCHY across all $r < s \leq 7$ for each graph, considering the fastest of ANH-TE, ANH-EL, and ANH-BL. The fastest running time is labeled in parentheses below each graph. We have omitted bars where ARB-NUCLEUS-HIERARCHY runs out of memory or times out after 4 hours. | 301 |
| 11.8 Speedup of ANH-TE on the left and ANH-EL on the right over their respective single-threaded running times, on dblp and skitter for various r and s . “30h” denotes 30-cores with two-way hyper-threading. | 301 |
| 11.9 Multiplicative slowdowns, comparing ANH-TE, ANH-EL, the parallel PHCD [76], and the sequential ND [281], for $(1, 2)$, $(2, 3)$, and $(3, 4)$ nucleus decomposition. We give the multiplicative slowdown over the fastest implementation for each graph and each (r, s) , where the fastest running time is labeled in parentheses below each graph. We include end-to-end running times in this comparison, excluding only the time required to load the graph. We have omitted bars where the implementation runs out of memory or times out after 4 hours. We have also included a line marking a multiplicative slowdown of 1. | 302 |

| | |
|--|-----|
| 12.1 An example where parallel local vertex moves lowers the total objective. Assume $\lambda = 0$ and initial clusters are singletons with objective 0. If b and c are both scheduled to move at the same time, they each choose cluster $\{a\}$, leading to a single cluster $\{a, b, c\}$ with objective -1. | 311 |
| 12.2 The vertices and edges added to the graph G , given a gate $g_i \vee g_j$ which outputs to g_k . | 317 |
| 12.3 Multiplicative slowdown in average time of each optimization. Also shown is the slowdown for no optimizations (base) over every optimization. Running times were obtained for PAR-CC and PAR-MOD on the graphs amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. | 320 |
| 12.4 CC objective for PAR-CC on a symmetric log scale ¹ (top) and multiplicative increase in the modularity for PAR-MOD over no optimizations (bottom), of each optimization and every optimization. Objectives were obtained on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. | 321 |
| 12.5 Number of vertices in V' per best move iteration of PAR-CC, for amazon (left) and orkut (right), considering neighbors of clusters and neighbors of vertices as the subset V' . | 322 |
| 12.6 Speedup of PAR-CC over SEQ-CC (left) and of PAR-MOD over SEQ-MOD (right), on amazon, dblp, livejournal, orkut, twitter, and friendster, for varying resolutions. SEQ-CC timed out on twitter for $\lambda = 0.01, 0.1$, and 0.25, and on friendster for $\lambda < 0.75$. | 324 |
| 12.7 Multiplicative increase in the number of rounds until convergence or until the default number of iterations is reached, of PAR-CC over SEQ-CC (left) and of PAR-MOD over SEQ-MOD (right), on amazon, dblp, livejournal, orkut, twitter, and friendster, for varying resolutions. SEQ-CC timed out on twitter for $\lambda = 0.01, 0.1$, and 0.25, and on friendster for $\lambda < 0.75$. | 325 |
| 12.8 Scalability of PAR-CC over rMAT graphs of varying sizes. | 325 |
| 12.9 Scalability of PAR-MOD over rMAT graphs with varying numbers of vertices. | 325 |
| 12.10 Scalability of PAR-CC over different numbers of threads, on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. 30h and 48h indicate 30 and 48 cores respectively, with two-way hyper-threading. | 326 |
| 12.11 Scalability of PAR-MOD over different numbers of threads, on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. Note that 30h and 48h indicate 30 and 48 cores respectively, with two-way hyper-threading. | 326 |
| 12.12 Multiplicative memory overhead, over the size of the input graph, of PAR-CC and PAR-MOD, on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. The labels below denote the size of the input graph. | 326 |
| 12.13 Speedup of PAR-MOD over NETWORKKIT on amazon, dblp, livejournal, and orkut, for varying resolutions. | 328 |

| | | |
|-------|---|-----|
| 12.14 | Average precision and recall compared to ground truth communities on amazon (left) and orkut (right) of the clusters obtained from PAR-CC and PAR-MOD, compared to SEQ-CC and SEQ-MOD, using varying resolutions. | 329 |
| 12.15 | Average precision and recall compared to ground truth communities on dblp and livejournal of the clusters obtained from our parallel implementations PAR-CC and PAR-MOD, using varying resolutions. | 330 |
| 12.16 | Average precision and recall compared to ground truth communities on amazon and dblp (left), and on livejournal and orkut (right), of the clusters obtained from PAR-CC using varying resolutions, and from TECTONIC using varying θ | 330 |
| 12.17 | Average precision-recall and ARI-NMI scores for the digits graph, from PAR-CC, PAR-MOD, PAR-CC ^w , and NETWORKKIT. | 331 |
| 12.18 | Average precision-recall and ARI-NMI scores for the letter graph, from PAR-CC, PAR-MOD, PAR-CC ^w , and NETWORKKIT. | 331 |
| 13.1 | The number of rounds required by the ParHAC (using $\epsilon = 0.1$) and reciprocal agglomerative clustering (RAC) algorithms for six large real-world graphs. The ClueWeb (CW) and Hyperlink (HL) graphs are two of the largest publicly available graphs, with 978M and 1.72B vertices, and 74B and 124B edges, respectively. We mark experiments where RAC does not finish after 6 hours with a red x. | 335 |
| 13.2 | Speedups for three of our large real-world graphs on the y -axis versus the number of hyper-threads used on the x -axis. | 352 |
| 13.3 | Parallel running time of the ParHAC algorithm in seconds on the y -axis in log-scale versus the size of each graph in terms of the total number of vertices and edges on the x -axis. | 352 |
| 13.4 | Parallel running times of the ParHAC algorithm on the LJ graph in log-scale as a function of the number of threads for varying values of the accuracy parameter, ϵ | 352 |
| 13.5 | ARI score on the <i>digits</i> point dataset as a function of the k used in graph building. The gray vertical line highlights the values for $k = 10$, which is the value used across all datasets for the results in Table 13.2. The red horizontal line is the ARI of SciPy’s average-linkage. | 353 |
| 13.6 | Relative performance of our ParHAC algorithm compared to the SeqHAC, SeqHAC _{\mathcal{E}} , RAC, Affinity, and SCC _{sim} algorithms. We prefix new implementations developed in this paper with the Par prefix. ParHAC, SeqHAC, SeqHAC _{\mathcal{E}} , and RAC are HAC-like algorithms that produce full dendograms, whereas Affinity and SCC _{sim} are MST-based methods that produce a (typically small) set of flat clusterings. The values on top of each bar show the running time of each algorithm in seconds. The times shown for the parallel algorithms are 144 hyper-thread times. We run the ParHAC and SeqHAC algorithm using $\epsilon = 0.1$. We terminated methods that ran for more than 6 hours and mark them with a red x. | 354 |

List of Tables

| | | |
|-----|---|-----|
| 3.1 | These are relevant statistics for the KONECT [199] graphs that we experimented on. | 83 |
| 3.2 | These are best runtimes in seconds for parallel and sequential butterfly counting from PARBUTTERFLY (PB), as well as runtimes from previous work. Note that PGD [7] is parallel, while the rest of the implementations are serial. Also, for the runtimes from our framework, we have noted the ranking used; * refers to side ranking, # refers to approximate complement degeneracy ranking, and \circ refers to approximate degree ranking. The wedge aggregation method used for the parallel runtimes was simple batching, except the cases labeled with \diamond , which used wedge-aware batching. | 83 |
| 3.3 | These are the fractional values $f = (w_s - w_r)/w_s$, where w_s is the number of wedges that must be processed using side ordering and w_r is the number of wedges that must be processed using the labeled ordering. Note that for itwiki, the number of wedges produced by degree ordering is precisely equal to the number of wedges produced by side ordering. | 87 |
| 3.4 | These are runtimes in seconds for sequential butterfly counting from PARBUTTERFLY (using the cache optimization), as well as runtimes from previous work. Note that PGD [7] is parallel, while the rest of the implementations are serial. Also, for the runtimes from our framework, we have noted the ranking used; * refers to side ranking, # refers to approximate complement degeneracy ranking, \diamond refers to degree ranking, and \circ refers to approximate degree ranking. We have also noted the wedge aggregation and butterfly aggregation methods used for the parallel runtimes; * refers to hashing with atomic adds, \blacklozenge refers to histogramming for both aggregation methods, and \square refers to wedge-aware batching. The rest of the parallel runtimes were obtained using simple batching. | 88 |
| 4.1 | Relevant statistics of our input graphs. | 111 |

| | | |
|-----|--|-----|
| 4.2 | Running times (seconds) of the two exact serial implementations and the two exact parallel five-cycle counting implementations without the work scheduling optimization. All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. For the serial algorithms, T_E is our implementation of the ESCAPE algorithm with arboricity orientation, and T_K is our implementation of the serial Kowalik's algorithm. The serial runtimes are measured using the Goodrich-Pszona degeneracy ordering algorithm. For the parallel algorithms, we use superscripts to indicate the orientation that achieved the best running time. ^g refers to Goodrich-Pszona, ^b refers to Barenboim-Elkin, and ^k refers to k -core orientation. Note that degree orientation is never the fastest orientation. For the parallel algorithms, we list the runtimes obtained on a single thread (T_1), 36 cores without hyper-threading (T_{36}), and 36 cores with hyper-threading (T_{36h}). We also tested all implementations on friendster, but they all exceeded the time limit. | 112 |
| 4.3 | These are the number of binary searches each exact algorithm performed for each dataset, and the ratio of the number of binary searches in the ESCAPE algorithm to Kowalik's algorithm. | 113 |
| 4.4 | Single-thread (T_1) and 36-core with hyper-threading (T_{36h}) running times (seconds) of the exact parallel Kowalik and ESCAPE algorithms with the work scheduling optimization, and their parallel speedups. All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. The superscripts indicate the orientation that achieved the best runtime. ^g refers to Goodrich-Pszona, ^b refers to Barenboim-Elkin, and ^o refers to degree orientation. In Table 4.5, we present the data for all orientations. . . | 113 |
| 4.5 | Single-thread (T_1) and 36-core with hyper-threading (T_{36h}) running times (seconds) of the exact parallel Kowalik and ESCAPE algorithms with the work scheduling optimization using all four orientations. All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. The bold values mark the best serial and parallel runtimes for each of Kowalik and ESCAPE, out of the four orientations, which are used in Table 4.4. . | 114 |
| 5.1 | Number of vertices, number of undirected edges (once in each direction), and total k -clique counts for our input graphs. We do not have statistics for certain graphs for large values of k , because algorithm did not terminate in under 5 hours; these entries are represented by a dash. . | 131 |

| | | |
|-----|--|-----|
| 5.2 | Best runtimes in seconds for our parallel (T_{60}) and single-threaded (T_1) k -clique counting algorithm (COUNT-CLIQUEs), as well as the best parallel and sequential runtimes from KCLIST [86]. The fastest runtimes for each experiment are bolded and in green. All runtimes are from tests in the same computing environment, and include time spent pre-processing and counting (but not time spent loading the graph). For our parallel runtimes and KCLIST, we have chosen the fastest orientations and choice between node and edge parallelism per experiment, while for our serial runtimes, we have fixed the orientation given by degree ordering. For the parallel runtimes from COUNT-CLIQUEs, we have noted the orientation used; \circ refers to the Goodrich-Pszona orientation, $*$ refers to the orientation given by k -core, and no superscript refers to the orientation given by degree ordering. For both implementations we have noted whether node or edge parallelism was used; e refers to edge parallelism, and no superscript refers to node parallelism. | 133 |
| 5.3 | The parallel runtimes in seconds for our 4-clique counting algorithm (COUNT-CLIQUEs) with degree ordering and node parallelism on large compressed graphs, using 160 cores with hyper-threading. | 134 |
| 6.1 | Table of notations used in this chapter. | 139 |
| 6.2 | Work and span bounds of algorithms in this chapter. ⁵ | 141 |
| 6.3 | Graph sizes and largest values of k for k -core decomposition. | 156 |
| 7.1 | Summary of graph notation. | 191 |
| 7.2 | The graphs used in our experiments, along with the sizes, maximum degree (dmax), degeneracy (δ), and number of rounds required in peeling, or the bi-core peeling complexity, (ρ) are shown. | 203 |
| 8.1 | These are relevant statistics for the KONECT [199] graphs that we experimented on. Note that we only tested peeling algorithms on graphs for which Sariyüce and Pinar's [282] serial peeling algorithms completed in less than 5.5 hours. As such, there are certain graphs for which we have no available ρ_v and ρ_e data, and these entries are represented by a dash. | 226 |
| 8.2 | These are runtimes in seconds for parallel and single-threaded butterfly peeling from PARBUTTERFLY (PB) and serial butterfly peeling from Sariyüce and Pinar [282]. Note that these runtimes do not include the time taken to count butterflies. For the runtimes from PARBUTTERFLY, we have noted the aggregation method used; $*$ refers to simple batching, $\#$ refers to sorting and \circ refers to histogramming. | 227 |
| 9.1 | Relevant k -clique peeling statistics for the SNAP graphs that we experimented on. We do not have statistics for certain graphs for large values of k , because the corresponding k -clique peeling algorithms did not terminate in under 5 hours; these entries are represented by a dash. | 238 |

| | |
|---|-----|
| 9.2 Best runtimes in seconds for our parallel and single-threaded k -clique peeling algorithm (PEEL-CLIQUEs), as well as the best sequential runtimes from previous work (KCLIST) [86]. The fastest runtimes for each experiment are bolded and in green. All runtimes are from tests in the same computing environment, and include only time spent peeling. For our parallel runtimes, we have chosen the fastest orientations per experiment, while for our serial runtimes, we have fixed the orientation given by degree ordering. For the parallel runtimes from COUNT-CLIQUEs, we have noted the orientation used; \circ refers to the Goodrich-Pszona orientation, and no superscript refers to the orientation given by degree ordering. | 240 |
| 11.1 Sizes of our input graphs, which are from SNAP [207]. | 297 |
| 12.1 Sizes of graph inputs. | 318 |
| 13.1 Graph inputs, including the number of vertices (n), number of directed edges (m), and the average degree (m/n). | 348 |
| 13.2 Adjusted Rand-Index (ARI), Normalized Mutual Information (NMI), Dendrogram Purity, and Dasgupta cost of our new ParHAC implementations (columns 2–3) versus the RAC and SeqHAC implementations (columns 4–5), our Affinity and SCC _{sim} implementations (columns 6–7), and the HAC implementations from SciPy (columns 8–11). Note that both RAC and SeqHAC _{\mathcal{E}} are exact HAC algorithms, and thus compute the same dendrogram. The scores are calculated by evaluating the clustering generated by each cut of the generated dendrogram to the ground-truth labels for each dataset. All graph-based implementations are run over the similarity graph constructed from an approximate k -NN graph with $k = 10$. The Dasgupta cost is computed over the complete similarity graph generated from the all-pairs distance graph, and thus takes into account all pairwise similarities. We display the best quality score for each graph in green and underlined. | 349 |

Chapter 1

Introduction

Large-scale graph processing is one of the major computational challenges of the 21st century, and is a fundamental tool in modern data mining with wide-ranging applications including in social network analysis, fraud and threat detection, recommendation systems, bioinformatics, and machine learning. Graphs have the powerful capability to capture complex interactions and relational metadata, and having principled approaches to efficiently processing these graphs is critical to a computational future. For example, finding certain recurring patterns and dense substructures in biological networks, such as protein interaction networks and gene regulatory networks, can be essential in indicating the functional contributions of these biological entities, such as in the expression of certain diseases. A central problem in graph processing is graph clustering, or community detection, which has a wide range of applications spanning social network analysis [142], recommendation and search systems [87], bioinformatics including in metabolic pathway analysis [183], and machine learning pipelines [174], and has been well-studied under many frameworks. At its core, identifying the underlying substructures of a graph can indicate essential functional groups, such as people with similar interests, news articles on similar topics, or proteins with similar utilities, which can then be synthesized for a variety of applications.

However, as the need to analyze larger and larger data sets increases, designing scalable algorithms that can handle billions of edges while maintaining fast speed and high quality becomes crucial. Current infrastructure for large-scale clustering in production settings relies not only on distributed algorithms with the ability to process trillion-size datasets [30, 309], but also on shared-memory parallel algorithms with the ability to process billion-size datasets [295]. Importantly, large distributed clusters are prohibitively expensive in many use-cases, while commodity multi-core machines are widely available to the average consumer, with much lower per-hour costs and generous machine sizes of up to 24 TB of memory [1]. Moreover, sequential algorithms do not take advantage of the multiple cores available and are prohibitively slow for datasets of these sizes. Thus, it is essential to develop efficient and performant shared-memory parallel algorithms and implementations that can scale to up to datasets of sizes in the hundreds of billions, from both a cost-savings and accessibility point of view.

Achieving parallel implementations that take full advantage of multiple cores is

non-trivial. The problems that we study in this thesis are work-intensive, so highly parallel solutions that significantly increase the asymptotic work complexity, or the total number of operations, lead to poor performance, especially for large datasets and where a limited number of cores is available. Moreover, the state-of-the-art sequential algorithms often have sequential dependencies that limit the amount of parallelism that can be naively introduced, particularly when scaling these solutions to increasing numbers of cores. Some of these problems are also known to be (or we show that they are) P-complete, indicating that polylogarithmic span solutions, or solutions with a polylogarithmic longest chain of sequential dependencies, are highly unlikely, theoretically limiting the scalability of solutions to these problems. Especially in these scenarios, we must additionally take into account practical considerations in designing parallel implementations. Factors including cache performance and memory contention have significant effects on performance (for instance, see [92]), notably in large-scale multicore algorithms that involve frequent memory accesses and updates to shared locations.

This thesis tackles these problems from two angles. *First, we focus on developing algorithms with strong theoretical guarantees, which often translates to the most significant performance improvements in practice, over algorithms without theoretical bounds or with asymptotically worse theoretical guarantees.* By focusing specifically on algorithms with low work and span, our implementations start from a strong theoretical grounding. *Second, we use performance engineering techniques implemented on top of these theoretically efficient algorithms to achieve fast implementations on real-world datasets.* We investigate parallel data structures and techniques that offer tradeoffs between performance and space-usage, as well as heuristics that optimize for cache-efficiency and minimize parallel overheads. The overall goal of this thesis is as follows:

We contend that developing shared-memory parallel clustering algorithms with strong theoretical guarantees and using performance engineering techniques can lead to highly scalable, efficient, and cost-effective implementations on real-world datasets.

We focus in this thesis on shared-memory multicore machines, which are largely accessible to the general public through platforms including Amazon Web Services, Google Cloud Platform, and Microsoft Azure. These machines balance accessibility, cost-effectiveness, computational power, and scalability. In fact, even as real-world graph sizes have increased with the proliferation of large-scale data analysis, the largest publicly available graph today, the WebCommons Hyperlink2012 graph with 3.5 billion vertices and 128 billion edges [233], fits on a reasonably-sized shared-memory machine using standard graph storage formats. We show in this thesis that it is indeed feasible and downright reasonable to perform computationally intensive graph processing algorithms on these graphs on commodity multi-core machines, and that these multi-core computations often outperform their distributed counterparts. For our theoretical analysis, we use the work-span model [177, 84], where informally, the *work* of an algorithm is the total number of operations, and the *span* is the

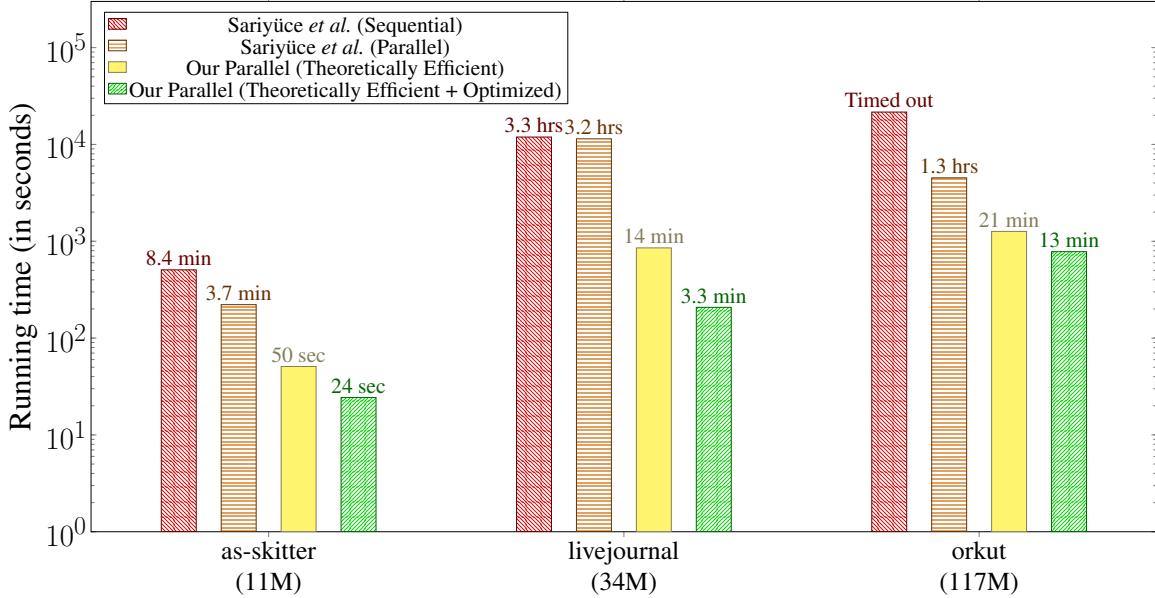


Figure 1.1: Running times in seconds for $(3, 4)$ nucleus decomposition on a 30-core machine with two-way hyper-threading of Sariyüce *et al.*'s [284] sequential implementation, Sariyüce *et al.*'s [283] parallel implementation, our parallel theoretically efficient implementation, and our parallel theoretically efficient implementation with practical optimizations. Note that the number of edges in each graph is labeled underneath the corresponding graph. The running times are given in log-scale.

longest dependency path.¹ Our goal throughout this thesis is to obtain *work-efficient* parallel algorithms in this model, that is, an algorithm with work complexity that asymptotically matches the best-known sequential time complexity for the problem.

We demonstrate our methodology with an example of our work on the nucleus decomposition problem (presented more formally in Chapters 10 and 11), which colloquially, involves finding hierarchical communities of densely connected vertices in a graph through higher-order structures. On a commodity 30-core machine with two-way hyper-threading and 240 GB of main memory, the prior state-of-the-art sequential nucleus decomposition implementation by Sariyüce *et al.* [284] takes 8.5 minutes to run on as-skitter (a real-world graph with 11 million edges), over 3 hours on livejournal (with 34 million edges), and over 6 hours on orkut (with 117 million edges).² These running times are displayed in Figure 1.1. Evidently, the nucleus decomposition problem is computationally intensive; it involves the repeated processing of hierarchies of subgraphs in the overarching graph, and as such, has a high computational workload. Introducing shared-memory parallelism is able to reduce these running times, and the previous state-of-the-art parallel implementation by Sariyüce

¹We describe this model in more detail in Chapter 2.

²Note that these running times are for the $(3, 4)$ nucleus decomposition problem. The precise definition of (r, s) nucleus decomposition is given in Chapter 10.

et al. [283] completes on as-skitter in 3.7 minutes, and on orkut in 1.3 hours.³ However, on the livejournal graph, Sariyüce *et al.*'s parallel algorithm brings the running time of nucleus decomposition from 3.3 hours to 3.2 hours, which is a relatively minor improvement. One of the main limitations of Sariyüce *et al.*'s parallel algorithm is that it is not theoretically efficient and has large span. This is particularly evident in practice on the livejournal graph, where the large theoretical span directly translates to limited performance improvements.

In this thesis, we present a theoretically efficient parallel nucleus decomposition algorithm, and implementing this algorithm, we significantly improve upon Sariyüce *et al.*'s parallel running times, completing in 50 seconds on as-skitter, 14 minutes on livejournal, and 21 minutes on orkut. Our notable speedup on the livejournal graph is because for this graph, our theoretically efficient algorithm performs over 80,000 times fewer the number of sequentially dependent rounds of computation than Sariyüce *et al.*'s parallel algorithm performs. In this sense, our theoretical guarantees translate directly to tangible benefits in practice. Additionally, we implement practical optimizations on top of our theoretically efficient algorithm, taking into account factors including cache behavior and parallel overheads, and we are able to compute the nucleus decomposition in 24 seconds on as-skitter, 3.3 minutes on livejournal, and 13 minutes on orkut. Our combination of techniques allows us to achieve up to a 55x speedup over the previous parallel state-of-the-art for the nucleus decomposition problem, demonstrating the overall effectiveness of considering theoretical and practical factors in conjunction with each other.

Using these approaches more broadly, we have developed theoretically-efficient algorithms with fast parallel implementations for a broad class of computationally intensive problems related to clustering. In this thesis, we study problems including the efficient discovery and enumeration of small subgraphs, which has applications in clustering metrics and graph statistics; decomposition algorithms to discover hierarchical dense substructures based on higher order subgraphs, which form hierarchical clusters in themselves, but can also be used for clustering metrics and in preprocessing tasks; and constructing generalized frameworks for graph and metric clustering with high quality on ground-truth data, including an exploration of the interplay between metric datasets and graph building techniques. Importantly, some of the work in this thesis relating to graph clustering is currently used in production environments at Google for real-world problems [295], and the work contributes to an actively used high-level framework for large-scale shared-memory parallel graph processing algorithms, the Graph Based Benchmark Suite (GBBS) [106]. Across these different topics, the work in this thesis spans results in theory, results in systems, and results bridging theory and systems, and we present in Figure 1.2 an overview of the various topics addressed and the results obtained in my body of work.

Our primary tools for overcoming computational barriers include designing algorithms with the structural properties of real-world graphs in mind, and using approximation algorithms and heuristic solutions to address problems with seem-

³Note that Sariyüce *et al.* [283] provide three different parallel implementations for nucleus decomposition. The running times that we list here represent the fastest of the three implementations for each graph.

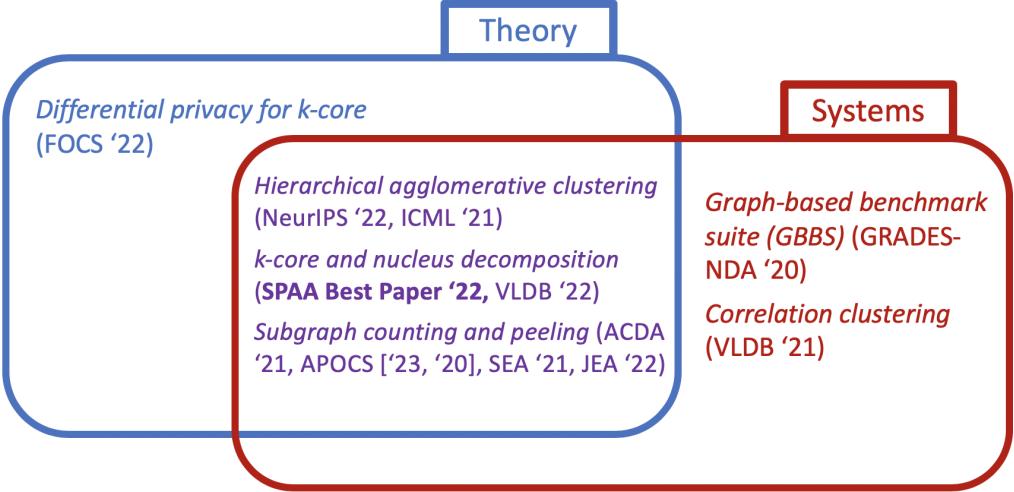


Figure 1.2: An overview of my body of work, including research presented in this thesis. Results for various topics are divided based on whether they contribute to theory, systems, or both.

ingly intractable sequential dependencies. By considering the nature and behavior of datasets used in practical applications, we can inform the development of theoretical guarantees that lead to the most significant performance improvements in practice. In particular, a key property of many real-world graphs from application areas spanning social media, bioinformatics, and traffic, is that they are often sparse [216], and leveraging this property can significantly reduce the amount of computation required in high workload graph processing problems. In this thesis, we develop algorithms to efficiently obtain low out-degree orientations⁴ for sparse graphs, which allows us to reduce the work of classic graph processing problems including subgraph counting, maximal matching, and coloring. We show throughout this thesis that reducing work theoretically often leads to the most significant speed increases in real-world settings. We also show that developing strong approximation guarantees leads to fast algorithms with high quality in practice, which is often tunable to achieve a tradeoff curve between running time and approximation factor.

Moreover, data today is often constantly evolving over time, such as with social networks and financial transaction graphs, so maintaining an orientation in the dynamic setting, which captures incoming edge updates on graphs, is especially important. We explore the batch-dynamic setting in this thesis, which is an extension of the dynamic setting in which graph properties or metrics are maintained over batches of multiple edge updates applied simultaneously, and we overcome sequential barriers using clever amortized analysis techniques.

The remainder of this chapter is organized as follows:

- Section 1.1 describes the contributions of this thesis to the efficient counting

⁴A low out-degree orientation is an orientation of edges in an undirected graph such that the maximum out-degree of any vertex is small.

and enumeration of small subgraphs, including small cycles and cliques. This section also discusses our contributions to developing efficient algorithms for low out-degree orientations, and k -clique counting in the batch-dynamic model of computation.

- Section 1.2 discusses our graph decomposition algorithms and implementations, which build upon our counting and enumeration algorithms to hierarchically discover higher-order dense substructures.
- Section 1.3 introduces our work on graph clustering, focusing specifically on scalable algorithms that demonstrate high quality compared to ground-truth clusters for a variety of real-world classification and community detection tasks.

1.1 Subgraph Counting and Listing

In the first part of this thesis, we design efficient parallel algorithms for a broad class of small subgraph counting and enumeration problems. We explore these problems in both exact and approximate, and both static and batch-dynamic settings, and we additionally provide algorithms for low out-degree orientations that we use as subroutines to reduce the work required in our subgraph counting algorithms.

In Chapters 3, 4, and 5, we study butterfly (four-cycle), five-cycle, and k -clique counting respectively, considering both exact and approximation algorithms and considering various graph orientation schemes to reduce the work. We show that our parallel algorithms are work-efficient with polylogarithmic span, and we implement our algorithms and demonstrate fast performance on large-scale graphs. These subgraphs capture various important properties of real-world graphs; notably, butterflies, or four-cycles, are core substructures signifying dense subregions in bipartite graphs, and cycles in general can represent fraud or money laundering in financial transaction graphs. k -cliques are also a fundamental marker of density in general graphs, and in Chapter 6, we present our work on clique counting in the batch-dynamic setting. We present new parallel batch-dynamic data structures, which we leverage to maintain an approximate k -core decomposition and a low out-degree orientation. We show that this low out-degree orientation can be used in a framework, to give provably-efficient parallel batch-dynamic algorithms for classic graph problems, including clique counting.

1.1.1 Butterfly Counting

Triangles are core substructures in unipartite graphs, and indeed triangle counting is a core metric with widespread applications in areas including social network analysis [246], spam and fraud detection [31], and link classification and recommendation [319]. However, bipartite graphs do not contain triangles; instead, *butterflies*, also known as (2, 2)-bicliques, 4-cycles, or rectangles, are the smallest non-trivial dense subgraph in bipartite graphs.

In Chapter 3, we present a framework, PARBUTTERFLY, that provides different implementations for butterfly counting with strong theoretical guarantees. The main procedure for butterfly counting involves finding *wedges*, or 2-paths, and combining them to count butterflies. In more detail, we find all wedges originating from each vertex, and then aggregate the counts of wedges incident to every distinct pair of vertices forming the endpoints of the wedge. The PARBUTTERFLY framework provides different ways to aggregate wedges in parallel, including sorting, hashing, histogramming, and batching. We further speed up the counting procedure by ranking vertices and only considering wedges formed by a particular ordering of the vertices. PARBUTTERFLY supports different parallel ranking methods, including side-ordering, approximate and exact degree-ordering, and approximate and exact complement-coreness ordering, all of which can be combined with any of the aggregation methods. Furthermore, we present parallel approximate butterfly counting algorithms and implementations within this framework, via graph sparsification based on ideas by Sanei-Mehri *et al.* [277] for the sequential setting. We additionally integrate a cache optimization for butterfly counting by Wang *et al.* [328].

We prove theoretical bounds showing that certain variants of our counting algorithms are work-efficient and take polylogarithmic span. Specifically, PARBUTTERFLY gives a counting algorithm that takes $O(\alpha m)$ expected work, $O(\log m)$ span w.h.p.,⁵ and $O(\min(n^2, \alpha m))$ additional space. Additionally, PARBUTTERFLY gives an approximate counting algorithm that takes $O((1+\alpha'p)m)$ expected work, $O(\log m)$ span w.h.p., and $O(\min(n^2, (1+\alpha'p)m))$ space, where α' is the arboricity of the sparsified graph and p is the sampling probability. We use two sparsification methods, edge sparsification and colorful sparsification, both of which give unbiased estimates of the total butterfly count.

Moreover, we present a comprehensive experimental evaluation of all of the different variants of counting algorithms in PARBUTTERFLY, and we show on a 48-core machine with 2-way hyperthreading, our counting algorithms achieve self-relative speedups of up to 39x and outperform the previous fastest sequential baseline by up to 14x [277]. Compared to PGD [7], the previous state-of-the-art parallel subgraph counting solution that can count butterflies as a special case, PARBUTTERFLY is 350–5169x faster.

1.1.2 Five-cycle Counting

Cycle counting specifically is an important problem in fraud detection [260], and of the 5-vertex subgraphs, five-cycles are significantly more difficult to count because they are the only such pattern that requires first counting all directed three-paths. Indeed, the Efficient Subgraph Counting Algorithmic PackagE (ESCAPE), a software package by Pinar *et al.* that serially counts all 5-vertex subgraphs [258], spends between 25–58% of the total 5-vertex subgraph counting runtime on counting five-cycles alone based on our measurement. We present in Chapter 4 two new parallel five-cycle counting algorithms that not only have strong theoretical guarantees, but are also

⁵We say $O(f(n))$ with high probability (w.h.p.) to denote $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$, where n is the input size.

demonstrably fast in practice. These algorithms are based on two different serial algorithms, namely by Kowalik [195] and from ESCAPE by Pinar *et al.* [258].

Kowalik studied k -cycle counting in graphs for $k \leq 6$ and proposed a five-cycle counting algorithm that runs in $O(md^2) = O(m\alpha^2)$ time for d -degenerate graphs [195], where α is the arboricity of the graph. Pinar *et al.*'s ESCAPE [258] contains contains a five-cycle counting algorithm that, with an important modification that we make, achieves the same asymptotic complexity of $O(m\alpha^2)$. The main procedure in both algorithms, and the essential modification to ESCAPE, is to first compute an appropriate arboricity orientation of the graph in parallel, where the vertices' out-degrees are upper-bounded by $O(\alpha)$. We use parallel orientation algorithms that we developed for parallel k -clique counting [296], which we discuss in more detail in Section 1.1.3. This orientation then enables the efficient counting of directed two-paths and three-paths, which are then appropriately aggregated to form five-cycles. Notably, the counting and aggregation steps can each be efficiently parallelized. The two algorithms differ fundamentally in the ways in which they use the orientations of these path substructures to eliminate double-counting.

We prove theoretical bounds that show that both of our algorithms match the work of the best sequential algorithms, taking $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p.. Additionally, we present two approximate five-cycle counting algorithms based on counting five-cycles in a sparsified graph, and we prove that both approximation algorithms give unbiased estimates on the global five-cycle count. We show that both algorithms take $O(pma^2 + m)$ expected work and $O(\log^2 n)$ span w.h.p. for a sampling probability p .

We present optimized implementations of our algorithms, which use thread-local data structures, fast resetting of arrays, and a new work scheduling strategy to improve load balancing. We provide a comprehensive experimental evaluation of our five-cycle counting algorithms. On a 36-core machine with 2-way hyperthreading, our best exact parallel algorithm achieves between 10–46x self-relative speedup, and between 162–818x speedups over the fastest prior serial five-cycle counting implementation, which is from ESCAPE [258]. We also implement our own serial versions of the two exact algorithms, which are 7–39x faster than ESCAPE's algorithm due to improved theoretical work complexities. Our best parallel algorithm achieves between 10–32x speedups over our best serial algorithm. Moreover, we show the tradeoffs between error and running time of our approximate five-cycle counting algorithms. In particular, we are able to approximate five-cycle counts with 12% error with a 9–189x speedup over exact five-cycle counting on the same graphs.

1.1.3 k -clique Counting and Listing

Finding k -cliques in a graph is a fundamental graph-theoretic problem with a long history of study in both theory and practice. In recent years, k -clique counting and listing have been widely applied in practice due to their many applications, including in learning network embeddings [270], understanding the structure and formation of networks [341, 321], identifying dense subgraphs for community detection [317, 284, 127, 153], and graph partitioning and compression [131].

In Chapter 5, we design a new parallel k -clique counting algorithm ARB-COUNT that matches the work of Chiba-Nishezaki [72] (which is the best known sequential algorithm for k -clique counting for sparse graphs), has polylogarithmic span, and has improved space complexity compared to KCLIST [86], the state-of-the-art parallel k -clique counting algorithm. Our algorithm is able to significantly outperform KCLIST and other competitors, and scale to larger graphs than prior work. ARB-COUNT is based on using low out-degree orientations of the graph to reduce the total work. Assuming that we have a low out-degree ranking of the graph, we show that for a constant k we can count or list all k -cliques in $O(m\alpha^{k-2})$ work, and $O(k \log n + \log^2 n)$ span w.h.p. where α is the arboricity of the graph. Theoretically, ARB-COUNT requires $O(\alpha)$ extra space per processor; in contrast, the KCLIST algorithm requires $O(\alpha^2)$ extra space per processor. Furthermore, KCLIST does not achieve polylogarithmic span.

We also design an approximate k -clique counting algorithm based on counting on a sparsified graph. We show that our approximate algorithm produces unbiased estimates and runs in $O(pma^{k-2} + m)$ work and $O(k \log n + \log^2 n)$ span w.h.p. for a sampling probability of p .

In order to obtain the low out-degree orientations used in our k -clique counting algorithms, we present two new parallel algorithms for efficiently ranking the vertices. We show that a distributed algorithm by Barenboim and Elkin [29] can be implemented in linear work and polylogarithmic span. We also parallelize an external-memory algorithm by Goodrich and Pszona [150] and obtain the same complexity bounds.

We perform a thorough experimental study on a 30-core machine with 2-way hyperthreading and compare to prior work. We show that on a variety of real-world graphs and different k , our k -clique counting algorithm ARB-COUNT achieves 1.31–9.88x speedup over the state-of-the-art parallel KCLIST algorithm [86] and self-relative speedups of 13.23–38.99x. We also compared our k -clique counting algorithm to other parallel k -clique counting implementations including Jain and Seshadhri’s PIVOTER [176], Mhedhbi and Salihoglu’s worst-case optimal join algorithm (WCO) [234], Lai *et al.*’s implementation of a binary join algorithm (BINARYJOIN) [201], and Pinar *et al.*’s ESCAPE [258], and demonstrate speedups of up to several orders of magnitude.

Furthermore, by integrating state-of-the-art parallel graph compression techniques, we can process graphs with tens to hundreds of billions of edges, significantly improving on the capabilities of existing implementations. *As far as we know, we are the first to report 4-clique counts for Hyperlink2012, the largest publicly-available graph, with over two hundred billion undirected edges.*

We study the accuracy-time tradeoff of our sampling algorithm, and show that is able to approximate the clique counts with 5.05% error 5.32–6573.63 times more quickly than running our exact counting algorithm on the same graph. We compare our sampling algorithm to Bressan *et al.*’s serial MOTIVO [60], and demonstrate 92.71–177.29x speedups.

1.1.4 Batch-dynamic k -core Decomposition and Clique Counting

A key challenge in discovering the structure of large-scale networks, and a key concept in achieving scalable and theoretically efficient subgraph counting algorithms, is to detect communities in which vertices have close ties with one another, or are well-connected with respect to each other. The well-connectedness of a vertex or a group of vertices is naturally captured by the concept of a k -core or, more generally, the k -core decomposition. Formally, given an undirected graph G with n vertices and m edges, the k -core of a graph is the maximal subgraph H such that every vertex in H has induced degree at least k . The k -core decomposition is a partition of the graph into layers, such that each vertex v is in layer k if it belongs to a k -core but not a $(k+1)$ -core; this value is known as the *coreness* of v , and these coreness values induce a natural hierarchical clustering. Classic algorithms for k -core decomposition are inherently sequential, and unfortunately, the k -core decomposition is a P-complete problem [17], suggesting that it is unlikely to obtain a parallel algorithm with polylogarithmic depth. In order to obtain a parallel algorithm with polylogarithmic depth, we relax the condition of obtaining an *exact* decomposition to one of obtaining a close *approximate* decomposition.

The approximate k -core decomposition is well-studied as a mechanism to obtaining faster and more scalable algorithms than in the exact setting [65, 144, 310, 124, 75], and approximate k -core is useful in applications where existing methods are already approximate, such as diffusion protocols in epidemiological studies [77, 191, 217, 226], community detection and network centrality measures [110, 125, 162, 235, 327, 346], network visualization and modeling [15, 63, 340, 347], protein interactions [16, 24], and clustering [146, 205]. Moreover, today's large networks are rapidly changing, and recent work has focused on the *dynamic* setting, where edges and vertices can be inserted and deleted, and the k -core decomposition is maintained in real time. Dynamic algorithms for k -core have been developed for both the sequential [209, 279, 349, 337, 209, 310, 214] and parallel [169, 184, 18] settings. However, to the best of our knowledge, *there are no prior existing parallel batch-dynamic k -core algorithms with provable polylogarithmic depth*, which we achieve in Chapter 6.

In Chapter 6, we focus on the *batch-dynamic* setting, where computations are performed over a batch of multiple edge updates applied simultaneously. This setting allows us to leverage parallelization to obtain scalable algorithms. We provide a work-efficient batch-dynamic approximate k -core decomposition algorithm based on a parallel level data structure that we design. We implement our algorithm and show experimentally that it performs favorably compared to the state-of-the-art. Furthermore, we show that our parallel level data structure can be used to obtain work-efficient parallel batch-dynamic algorithms for several other problems, specifically, low out-degree orientation and k -clique coloring.

1.2 Subgraph Decomposition

The second part of this thesis addresses a class of problems relating to the discovery and classification of dense substructures within a graph. The algorithms in this part focus on hierarchical decompositions, revealing structural properties of the underlying graph and exposing different notions or levels of density. We develop theoretically efficient parallel algorithms for these decomposition problems, and show that many of these problems are P-complete, suggesting that solutions that take polylogarithmic span are unlikely. As a result, we also explore approximation algorithms, and we conduct thorough experimental evaluations comparing speed and accuracy trade-offs between our exact and approximate implementations.

For bipartite graphs, we consider the bi-core decomposition in Chapter 7, which is a generalization of the classic k -core decomposition to bipartite graphs. Chapters 8 and 9 presents our theoretically efficient parallel algorithms and implementations for butterfly peeling and k -clique peeling respectively, which classify dense substructures of a graph based on butterfly and k -clique structures respectively, the former of which is of special interest in bipartite graphs as well. Chapters 10 and 11 address the nucleus decomposition problem, which considers higher-order clique structures in its decomposition. We design theoretically efficient parallel algorithms for classifying each r -clique based on the nucleus decomposition and constructing the hierarchy tree for nucleus decomposition respectively, and demonstrate that our algorithms result in fast implementations in practice.

1.2.1 Bi-core Decomposition

In the special case of bipartite graphs, the classic k -core decomposition [213, 229] does not allow the bipartitions to be distinguished from one another, which can be important especially if they exhibit different structures. Notably, since bipartite graphs model the affiliation between two distinct categories of entities, such as in authorship networks [181], group membership networks [286], user-product networks [326], and protein-protein interactions [134], it is often crucial to maintain the distinction between the entities, and allow for different notions of density to apply to one bipartition versus the other. Indeed, developing algorithms to apply specifically to bipartite graphs has become a recent popular direction of research [330, 6, 355, 282].

In Chapter 7, we focus on the bipartite equivalent of the k -clique decomposition, known as the bi-core decomposition, which was introduced by Ahmed *et al.* [6]. Formally, an (α, β) -core (or a bi-core) is the maximal subgraph where the induced degree of each vertex in the first partition is at least α , and the induced degree of each vertex in the second partition is at least β . The bi-core decomposition has applications in a variety of fields, including spam detection on social networks [41], community search on bipartite networks [331], movie viewership analysis [6], and group recommendation [107]. For example, Beutel *et al.* [41] leveraged the bi-core decomposition to detect spammers on user-post social networks, where spammers and fake accounts often form dense subgraphs by interacting with each others' posts, and Wang *et al.* [331]

used the bi-core decomposition as a subroutine for optimizing community search on bipartite networks, by reducing the search space for dense communities.

The previous state-of-the-art sequential algorithm for the bi-core decomposition is by Liu *et al.* [215], who use multiple peeling processes to compute the decomposition in $O(\delta m)$ time, where m is the number of edges and δ denotes the degeneracy of the graph. Liu *et al.* also introduce a parallel algorithm, but their algorithm only parallelizes across different peeling processes and does not parallelize the peeling process itself. As a result, their algorithm has $O(m)$ span, which limits its parallel scalability.

In Chapter 7, we present an efficient parallel bi-core decomposition algorithm, that fully parallelizes the peeling process. Our algorithm achieves $O(\delta m)$ work and $O(\rho \log n)$ span w.h.p., where n is the number of vertices and ρ denotes the bi-core peeling complexity, which we define as the maximum number of rounds of peeling required to remove all vertices from the graph. Notably, our algorithm is work-efficient, and since ρ is upper bounded by n , our parallel algorithm improves upon Liu *et al.*'s span bound when the number of edges is $\Omega(\rho \log n)$. We additionally prove the bi-core decomposition problem to be P-complete, suggesting that there is unlikely to be a polylogarithmic span solution under standard assumptions. Moreover, we present a parallel index structure, that allows for the efficient querying of all vertices x in a given bi-core, in work linear to the size of the bi-core and $O(1)$ span. We also introduce an algorithm to construct our index structure in parallel, which takes $O(m)$ work and $O(\log n)$ span w.h.p.

Finally, we introduce a heuristic optimization that speeds up our algorithm by pruning the number of peeling steps required to obtain the bi-core decomposition. We perform a comprehensive experimental evaluation of our algorithms on real-world graphs with up to hundreds of millions of edges, and show that our parallel bi-core decomposition algorithm achieves up to a 51.4x speedup (with an average of 27.9x) over Liu *et al.*'s sequential algorithm. on a 30-core machine with two-way hyper-threading. Our parallel algorithm also achieves up to a 4.9x speedup (with an average of 2.3x) over Liu *et al.*'s parallel algorithm. For our parallel index structure, our construction algorithm achieves up to a 27.7x speedup (with an average of 18.4x) over Liu *et al.*'s sequential index construction algorithm, and our query algorithm achieves a 116.3x speedup (with an average of 43.8x) over Liu *et al.*'s sequential query algorithm. Overall, we show that our implementations demonstrate good scalability over different numbers of threads and over large-scale graphs.

1.2.2 Butterfly Peeling

As discussed in Section 1.1.1, butterflies, or $(2, 2)$ -bicliques, are fundamental dense building blocks of bipartite graphs, and in particular, they naturally lend themselves to finding dense subgraph structures in bipartite networks. Zou [355] and Sariyüce and Pinar [282] developed peeling algorithms to hierarchically discover dense subgraphs. In Chapter 8, we present in our framework PARBUTTERFLY new parallel algorithms for butterfly peeling, with strong theoretical bounds. Our peeling algorithms iteratively remove the vertices (tip decomposition) or edges (wing decomposition) with

the lowest butterfly count until the graph is empty. Each iteration removes vertices (edges) from the graph in parallel and updates the butterfly counts of neighboring vertices (edges) using the parallel wedge aggregation techniques that we developed for counting, discussed in Section 1.1.1. We use a parallel bucketing data structure by Dhulipala *et al.* [95] and a new parallel Fibonacci heap to efficiently maintain the butterfly counts.

Our parallel Fibonacci heap improves upon the work bounds for vertex-peeling from Sariyüce and Pinar’s sequential algorithms, which take work proportional to the maximum number of per-vertex butterflies. PARBUTTERFLY gives a vertex-peeling algorithm that takes $O(\min(\max-b_v, \rho_v \log n) + \sum_{v \in V} \deg(v)^2)$ expected work, $O(\rho_v \log^2 n)$ span w.h.p., and $O(n^2 + \max-b_v)$ additional space, and an edge-peeling algorithm that takes $O(\min(\max-b_e, \rho_e \log n) + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$ expected work, $O(\rho_e \log^2 m)$ span w.h.p., and $O(m + \max-b_e)$ additional space, where $\max-b_v$ and $\max-b_e$ are the maximum number of per-vertex and per-edge butterflies and ρ_v and ρ_e are the number of vertex and edge peeling iterations required to remove the entire graph. Additionally, given a slightly relaxed work bound, we can improve the space bounds in both algorithms; specifically, we have a vertex-peeling algorithm that takes $O(\rho_v \log n + \sum_{v \in V} \deg(v)^2)$ expected work, $O(\rho_v \log^2 n)$ span w.h.p., and $O(n^2)$ additional space, and we have an edge-peeling algorithm that takes $O(\rho_e \log n + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$ expected work, $O(\rho_e \log^2 m)$ span w.h.p., and $O(m)$ additional space.

We present a comprehensive experimental evaluation of all of the different variants of butterfly peeling algorithms in the PARBUTTERFLY framework. On a 48-core machine with 2-way hyperthreading, our peeling algorithms achieve self-relative speedups of between 1.0–10.7x for vertex peeling and between 2.3–10.4x for edge peeling. Moreover, due to their improved work complexities, our peeling algorithms outperform the fastest sequential baseline [282] by between 1.3–30696x for vertex peeling and between 3.4–7.0x for edge peeling. Our speedups are highly variable because they depend heavily on the peeling complexities and the number of empty buckets processed.

1.2.3 k -clique Densest Subgraph

The k -clique densest subgraph problem is a generalization of the densest subgraph problem, which was first introduced by Tsourakakis [317], that captures higher-order k -clique structures in a graph in order to discover large near-cliques. This problem admits a natural $1/k$ -approximation by peeling vertices in order of their incident k -clique counts. We present in Chapter 9 a new parallel peeling algorithm, ARB-PEEL, that peels all vertices with the lowest k -clique count on each round and uses ARB-COUNT as a subroutine. The expected amortized work of ARB-PEEL is $O(m\alpha^{k-2} + \rho_k(G) \log n)$ and the span is $O(\rho_k(G)k \log n + \log^2 n)$ w.h.p., where $\rho_k(G)$ is the number of rounds needed to completely peel the graph and α is the arboricity of the graph. We also prove that the problem of obtaining the hierarchy given by this process is P-complete for $k > 2$, indicating that a polylogarithmic-span solution is unlikely.

Tsourakakis also shows that naturally extending the Bahmani et al. [26] algorithm for approximate densest subgraph gives an $1/(k(1 + \epsilon))$ -approximation in $O(\log n)$ parallel rounds, although they were not concerned about work. We present an $O(m\alpha^{k-2})$ work and polylogarithmic-span algorithm, ARB-APPROX-PEEL, for obtaining a $1/(k(1+\epsilon))$ -approximation to the k -clique densest subgraph problem. We obtain this work bound using our k -clique counting algorithm as a subroutine. Danisch *et al.* [86] use their k -clique counting algorithm as a subroutine to implement these two approximation algorithms for the k -clique densest subgraph as well, but their implementations do not have provably-efficient bounds.

We implement both ARB-PEEL and ARB-APPROX-PEEL with practical optimizations, and perform an experimental study on a 30-core machine with 2-way hyperthreading. We show that our parallel approximation algorithms for k -clique densest subgraph are able to outperform the parallel KCLIST [86] by up to 29.59x and achieve 1.19–13.76x self-relative speedup. We demonstrate up to 53.53x speedup over Fang *et al.*'s serial COREAPP [127] as well.

1.2.4 Nucleus Decomposition

Sariyüce *et al.* [284] introduced the nucleus decomposition problem, which generalizes the influential notions of k -cores and k -trusses to k -(r, s) nuclei, and can better capture higher-order structures in the graph. Informally, a k -(r, s) nucleus is the maximal induced subgraph such that every r -clique in the subgraph is contained in at least k s -cliques. The goal of the (r, s) nucleus decomposition problem is to identify for each r -clique in the graph, the largest k such that it is in a k -(r, s) nucleus.

Solving the (r, s) nucleus decomposition problem is a significant computational challenge for several reasons. First, simply counting and enumerating s -cliques is a challenging task, even for modest s . Second, storing information for all r -cliques can require a large amount of space, even for relatively small graphs. Third, engineering fast and high-performance solutions to this problem requires taking advantage of parallelism due to the computationally-intensive nature of listing cliques. There are two well-known parallel paradigms for approaching the (r, s) nucleus decomposition problem, a global peeling-based model and a local update model that iterates until convergence [283]. The former is inherently challenging to parallelize due to sequential dependencies and necessary synchronization steps [283], which we address in our work in Chapter 10, and we demonstrate that the latter requires orders of magnitude more work to converge to the same solution and is thus less performant.

Lastly, it is unknown whether existing sequential and parallel algorithms for this problem are theoretically efficient. Notably, existing algorithms perform more work than the fastest theoretical algorithms for k -clique enumeration on sparse graphs [72, 296], and it is open whether one can solve the (r, s) nucleus decomposition problem in the same work as s -clique enumeration.

We address the computational challenges by presenting in Chapter 10 a theoretically efficient parallel algorithm for (r, s) nucleus decomposition that nearly matches the work for s -clique enumeration, along with new techniques that improve the space and cache efficiency of our solutions. The key to our theoretical efficiency is a new

combinatorial lemma bounding the total sum over all k -cliques in the graph of the minimum degree vertex in this clique, which enables us to provide a strong upper bound on the overall work of our algorithm. As a byproduct, we also obtain the most theoretically-efficient serial algorithm for (r, s) nucleus decomposition. We provide several new optimizations for improving the practical efficiency of our algorithm, including a new multi-level hash table structure to space efficiently store data associated with cliques, a technique for efficiently traversing this structure in a cache-friendly manner, and methods for reducing contention and further reducing space usage. Finally, we experimentally study our parallel algorithm on various real-world graphs and (r, s) values, and find that it achieves between 3.31–40.14x self-relative speedup on a 30-core machine with 2-way hyperthreading. The only existing parallel algorithm for nucleus decomposition is by Sariyüce *et al.* [283], but their algorithm requires much more work than the best sequential algorithm. Our algorithm achieves between 1.04–54.96x speedup over the state-of-the-art parallel nucleus decomposition of Sariyüce *et al.*, and our algorithm can scale to larger (r, s) values, due to our improved theoretical efficiency and our proposed optimizations. We are able to compute the (r, s) nucleus decomposition for $r > 3$ and $s > 4$ on several million-scale graphs for the first time.

1.2.5 Nucleus Decomposition Hierarchy

The original formulation of the nucleus decomposition problem by Sariyüce *et al.* [284] involves constructing a hierarchy to capture the higher-order structures in a graph, instead of merely computing the largest k for each r -clique such that it is in a k - (r, s) nucleus, as discussed in Section 1.2.4. More precisely, the goal of the (r, s) nucleus decomposition problem in its entirety is to generate a *nucleus hierarchy* over the nuclei, where for $c' < c$ a c' - (r, s) nucleus A is a descendant of a c - (r, s) nucleus B if A is a subgraph of B . The nucleus decomposition hierarchy is then an unsupervised method for revealing dense substructures of a graph at different resolutions, and is important as it allows for easy exploration of the dense substructures of the graph in structural graph analysis tasks [281].

In Chapter 11, we introduce the first work-efficient parallel algorithm for hierarchy construction in nucleus decomposition. We show that our algorithm is work-efficient, where the key to our theoretical efficiency is our careful construction of subgraphs representing the s -clique-connectivity of r -cliques, that allows us to exploit linear-work graph connectivity instead of more expensive union-finds as used in prior work. Our approach gives as a byproduct the most theoretically-efficient serial algorithm for computing the hierarchy, improving upon the previous best known serial bounds by Sariyüce and Pinar [281].

We also present a practical parallel algorithm that interleaves the hierarchy construction with the computation of the coreness values. Prior work by Sariyüce and Pinar [281] also includes a (serial) interleaved hierarchy algorithm. However, their algorithm requires storing all adjacent r -cliques with different coreness values, which could potentially be proportional to the number of s -cliques in G , and which results in sequential dependencies in their post-processing step to construct the hierarchy.

Our parallel algorithm fully interleaves the hierarchy construction with the coreness computation, and uses only two additional arrays of size proportional to the number of r -cliques in G . Our algorithm uses a concurrent union-find data structure in an innovative way, and our post-processing step to construct the hierarchy tree is fully parallel. Our main insight is a technique to fully extract the connectivity information from adjacent r -cliques with different core numbers while computing the coreness values.

Additionally, we introduce an approximate algorithm for nucleus decomposition and show that it is work-efficient with polylogarithmic span. Our algorithm relaxes the peeling order by allowing all r -cliques within a $((\binom{s}{r}) + \epsilon)$ factor of the current value of k to be peeled in parallel. We show that our algorithm generates coreness estimates that are an $((\binom{s}{r}) + \epsilon)$ -approximation of the true coreness values.

We perform a comprehensive experimental study of our parallel algorithms on real-world graphs using different (r, s) values, for up to $r < s \leq 7$. On a 30-core machine with two-way hyper-threading, our exact algorithm, which generates both the coreness numbers and the hierarchy, achieves up to a 30.96x self-relative parallel speedup, and a 3.76–58.84x speedup over the state-of-the-art sequential algorithm by Sarıyüce and Pinar [281]. In addition, we show that on the same machine our approximate algorithm is up to 3.3x faster than our exact algorithm for computing coreness values, while generating coreness estimates with a multiplicative error of 1–2.92x (with a median error of 1.33x). Our algorithms are able to compute the (r, s) nucleus decomposition hierarchy for $r > 3$ and $s > 4$ on graphs with over a hundred million edges for the first time.

1.3 Graph Clustering

The final part of this thesis focuses on highly scalable graph clustering algorithms that are effective in practice and give good quality clusters compared to ground truth data, considering a broad class of classification tasks. We develop heuristic and approximation algorithms for classic graph clustering objective functions, and additionally demonstrate important relationships between graph clustering algorithms and their counterparts in pointset clustering.

Chapter 12 focuses on the correlation clustering objective, which is a generalization of the classic modularity objective. We present parallel heuristic algorithms that are highly performant in practice, with tunable precision and recall compared to ground-truth data. Chapter 13 presents our parallel algorithm for approximate hierarchical agglomerative clustering, which outputs a hierarchy of clusters where in every step, two vertices or points are merged together to form a cluster. We show hardness results for exact hierarchical agglomerative clustering, and show that our approximate algorithm runs in polylogarithmic span, with high quality on ground-truth data.

1.3.1 Correlation Clustering

A major challenge in graph clustering is to design algorithms that can achieve fast speed at high scale while retaining high quality as evaluated on data sets with ground truth. Many graph clustering algorithms have been proposed to address this challenge, and our goal is to develop a state-of-the-art algorithm from both speed and quality perspectives. In particular, in Chapter 12, we adopt a new LAMBDAACC framework, introduced by Veldt *et al.* [323], which provides a general objective encompassing modularity [149] and correlation clustering [27]. Modularity is a widely-used objective that is formally defined as the fraction of edges within clusters minus the expected fraction of edges within clusters, assuming random distribution of edges. The goal of correlation clustering is to maximize agreements or minimize disagreements, where agreements and disagreements are defined based on edge weights indicating similarity and dissimilarity.

It is NP-hard to approximate modularity within a constant factor [108], so optimizing for modularity, and by extension optimizing for the LAMBDAACC objective, is inherently difficult. The most successful and widely-used modularity clustering implementations focus on heuristic algorithms, notably the popular Louvain method [51]. The Louvain method has been well-studied for use in modularity clustering, with highly optimized heuristics and parallelizations that allow them to scale to large real-world networks [219, 314, 308, 315].

We design, implement, and evaluate a generalized sequential and shared-memory parallel framework for Louvain-based algorithms including modularity and correlation clustering. We optimize the LAMBDAACC objective with state-of-the-art empirical performance, scaling to graphs with billions of edges. We also show that there is an inherent bottleneck to efficiently parallelizing the Louvain method, in that the problem of obtaining a clustering matching that given by the Louvain method on the LAMBDAACC objective, is P-complete. As such, we explore heuristic optimizations and relaxations of the Louvain method, and demonstrate their quality and performance trade-offs for the LAMBDAACC objective.

As part of our comprehensive empirical study, we show that our sequential implementation is orders of magnitude faster than the proof-of-concept implementation of Veldt *et al.* [323]. We note that for both LABMDACC and correlation clustering objective, we are unaware of any existing implementation that would scale to even million-edge graphs and achieve comparable quality. We further show that our parallel implementations obtain up to 28.44x speedups over our sequential baselines on a 30-core machine with 2-way hyperthreading.

Moreover, we show that optimizing for the correlation clustering objective is of particular importance, by studying cluster quality with respect to ground truth data. We observe that optimizing for correlation clustering yields higher quality clusters than the ones obtained by optimizing for the celebrated modularity objective. In addition, we compare our implementation to two other prominent scalable algorithms for community detection: Tectonic [321] and SCD [259] and in both cases obtain favorable results, improving both the performance and quality. Finally, even in the highly competitive and extensively studied area of optimizing for modularity, we ob-

tain an up to 3.5x speedup over a highly optimized parallel shared-memory modularity clustering implementation in NetworKit [308].

1.3.2 Hierarchical Agglomerative Clustering

Hierarchical agglomerative clustering (HAC) [190, 203, 306] is a classic and commonly used clustering algorithm that delivers high-quality results. The premise behind HAC is simple: given n input points, it proceeds in $n - 1$ step where in each step, it merges together the two most similar points. At the conclusion of the process, all input points are merged into one. Alternatively, we can describe HAC on an immutable set of n points, where we maintain a mutable clustering that initially separates each point into its own cluster. In each step, we replace the two most similar clusters by their union, where the notion of similarity is specified by a configurable *linkage function*. The linkage function is usually given as input all pairwise similarities between points from the two clusters, and the most popular functions are *average-linkage* (the arithmetic mean of all similarities), *single-linkage* (the maximum similarity), and *complete-linkage* (the minimum similarity). Among these, average-linkage is of particular importance, as it is known to find high-quality clusters in real-world applications [293, 240, 82, 237].

However, the HAC algorithm is computationally expensive, and a major obstacle to obtaining an efficient and scalable HAC implementation is due to its inherently sequential nature [30, 237], which makes it challenging to parallelize. In Chapter 13, we present **ParHAC**, an efficient parallel algorithm for $(1 + \epsilon)$ -approximate average-linkage HAC with polylogarithmic span and near-linear total work, where the notion of approximation is given by [239]. Our algorithm takes as input a similarity graph consisting of n vertices (representing input points) and m weighted edges (representing nonzero similarities between points), and for fixed ϵ , **ParHAC** runs in polylogarithmic span and $\tilde{O}(m)$ -work⁶. Thus, up to logarithmic factors, **ParHAC** is work-efficient, and achieves very high parallelism (ratio of work to span). Prior to our result, no average-linkage HAC algorithm with sublinear depth was known (even allowing approximation). In addition, we show that allowing approximation is necessary, as we show the problem of exact HAC to be P-complete, suggesting that it is not possible to achieve polylogarithmic span under standard complexity-theory assumptions.

We implement and evaluate **ParHAC** on publicly available real-world datasets and present a comprehensive comparison to other parallel and sequential implementations. Notably, our parallel implementation obtains 50.1x speedup on average compared to the best sequential baseline, while maintaining roughly the same quality.

1.4 Research Presented in This Thesis

The research presented in this thesis consists of the following works, conducted with my co-authors:

- Jessica Shi and Julian Shun. “Parallel Algorithms for Butterfly Computations”. In: *Massive Graph Analytics*, pp. 287-330, 2022. (Chapters 3 and 8)

⁶We define $\tilde{O}(f(x)) := O(f(x)\text{polylog}f(x))$.

- Jessica Shi and Julian Shun. “Parallel Algorithms for Butterfly Computations”. In: *Proceedings of the SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pp. 16-30, 2020. (Earlier version)
- Jessica Shi, Louisa Huang, and Julian Shun. “Parallel Five-Cycle Counting Algorithms”. In: *ACM Journal of Experimental Algorithms (JEA)*, Vol. 27, Article No. 4:1, pp.1-23, 2022. (Chapter 4)
 - Louisa Huang, Jessica Shi, and Julian Shun. “Parallel Five-Cycle Counting Algorithms”. In: *Proceedings of the International Symposium on Experimental Algorithms (SEA)*, pp. 2:1-2:18, 2021. (Earlier version)
 - Jessica Shi, Laxman Dhulipala, and Julian Shun. “Parallel Clique Counting and Peeling Algorithms”. In: *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*, pp. 135-146, 2021. (Chapters 5 and 9)
 - Quanquan Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. “Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems”. In: *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 191-204, 2022. (Chapter 6)
 - Yihao Huang, Claire Wang, Jessica Shi, and Julian Shun. “Efficient Algorithms for Parallel Bi-core Decomposition”. In: *Proceedings of the SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pp. 17-32, 2023. (Chapter 7)
 - Jessica Shi, Laxman Dhulipala, and Julian Shun. “Theoretically and Practically Efficient Parallel Nucleus Decomposition”. In: *Proceedings of the VLDB Endowment*, 15(3), pp. 583-596, 2022. (Chapter 10)
 - Jessica Shi, Laxman Dhulipala, and Julian Shun. “Hierarchical and Approximate Parallel Nucleus Decomposition”. In submission. (Chapter 11)
 - Jessica Shi, Laxman Dhulipala, David Eisenstat, Jakub Łącki, and Vahab Mirrokni. “Scalable Community Detection via Parallel Correlation Clustering”. In: *Proceedings of the VLDB Endowment*, 14(11), pp. 2305–2313, 2021. (Chapter 12)
 - Laxman Dhulipala, David Eisenstat, Jakub Łącki, Vahab Mirrokni, and Jessica Shi. “Hierarchical Agglomerative Graph Clustering in Poly-Logarithmic Depth”. In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 22925-22940, 2022. (Chapter 13)

Other Research. Several other works are not included in this thesis, but are relevant to the topics in this thesis in various ways. [106] presents the Graph Based Benchmark Suite (GBBS), a high-level framework and an accompanying suite of efficient shared-memory graph processing algorithms that scale to large graphs with hundreds of billions of edges on commodity multicore machines. GBBS provides parallel functional

primitives for common graph processing operations and various graph representation formats with options for compression to reduce memory usage. Our work additionally includes a Python-based interface, with support for loading graphs from a variety of common formats and sources. Many of the implementations for the works in this thesis are built using GBBS, and have been incorporated directly into the benchmark suite.

[103] extends upon the algorithms given in [216] in the parallel batch-dynamic setting to obtain differentially private k -core decomposition, low out-degree orientation, and densest subgraph algorithms, as well as a general framework that allows locally adjustable algorithms to be converted into differentially private algorithms. Our framework formalizes a relationship between parallel graph algorithms and differential privacy, and our results improve upon the previous best multiplicative approximation factor for private densest subgraph in near-linear time.

[100] forms the basis of the parallel hierarchical agglomerative graph clustering algorithm in [101]. In the sequential setting, we develop a general framework that encompasses common linkage measures for hierarchical agglomerative clustering (HAC). In many cases our framework gives a nearly-linear time algorithm for exact HAC, and in the popular setting of average-linkage HAC, we obtain a subquadratic algorithm for exact HAC. We additionally provide an ϵ -close approximation algorithm for average-linkage HAC that runs in near-linear time. We validate these results experimentally, and demonstrate that these approaches can speed up the clustering of point datasets by up to 76.5x.

1.5 Thesis Statement

This thesis addresses the challenges of developing practical and theoretically grounded algorithms that can be used widely both in industry and in academia for graph clustering applications. The overall approach of this thesis is to gain a deep theoretical understanding of the underlying bottlenecks and inherent complexity barriers of the problem, which informs the major design considerations of a practical system. Conversely, we also consider the properties of real-world datasets and their behaviors, taking into account systems factors such as cache behavior, memory bandwidth, and parallel overheads, which additionally gives insight into the important concepts of interest in the theoretical setting. The broader vision of this thesis is to tackle computationally intensive problems that are important in industry applications across a wide variety of domains, by considering the cross-cutting effects of techniques from theory, systems, and architecture. Our work demonstrates the importance of considering the ways in which theoretical algorithm development interacts with performance analysis and engineering, and I believe that from a practical standpoint, the algorithms and implementations in this thesis represent a microcosm of the power of interdisciplinary approaches. The code for all of our implementations are publicly available, and some of our implementations are currently used in industry settings for real-world graph processing tasks.

This thesis presents evidence in support of the following statement.

Thesis Statement: *This thesis contends that developing shared-memory parallel clustering algorithms with strong theoretical guarantees and using performance engineering techniques can lead to highly scalable, efficient, and cost-effective implementations on real-world datasets.*

Chapter 2

Preliminaries and Notation

2.1 Graph Notation

We use the following notation for graphs throughout this thesis. Unless otherwise specified, we take graphs $G = (V, E)$ to be simple, unweighted, and undirected, where V is the set of vertices and E is the set of edges. We use $n = |V|$ to denote the number of vertices in G , and $m = |E|$ to denote the number of edges in G . We label vertices by the indices $\{0, \dots, n - 1\}$. For analysis, we assume $m = \Omega(n)$.

For any vertex $v \in V$, we let $N(v)$ denote the neighborhood of v , or the set of neighbors of v , and we let $\deg(v)$ denote the degree of v . If there are multiple graphs or if the graph in reference is ambiguous, we let $N_G(v)$ denote the neighborhood of v in the graph G . Additionally, for any vertex $v \in V$, we let $N^2(v)$ denote the 2-hop neighborhood of v , or the set of all vertices reachable from v by a path of length 2, and similarly, for added clarity when dealing with multiple graphs, we let $N_G^2(v)$ denote the 2-hop neighborhood of v in the graph G . We use $G[U]$ to denote the subgraph of G induced by the vertices in U .

We occasionally use directed graphs in this thesis, and we make clear when we are working with a directed graph rather than an undirected graph. Unless otherwise specified, for a directed graph DG , we let $N(v) = N_{DG}(v)$ denote the out-neighborhood of vertex v in DG , or the set of out-neighbors of v in DG .

2.1.1 Degeneracy and Arboricity

In many of the results in this thesis, we use the closely related notions of k -core, degeneracy, and arboricity. First, we define the notion of k -core.

Definition 2.1 (k -Core). *For a graph $G = (V, E)$ and positive integer k , the k -core of G is the maximal subgraph of G with minimum induced degree k .*

We also define a k -core decomposition using the definition of k -core.

Definition 2.2 (k -Core Decomposition). *A k -core decomposition is a partition of vertices into layers such that a vertex v is in layer k if it belongs to a k -core but not to a $(k + 1)$ -core. The layer k that v is in is called the coreness of v .*

The degeneracy of a graph G is then closely related to the k -core decomposition of G .

Definition 2.3 (Degeneracy). *The **degeneracy** of a graph $G = (V, E)$ is the maximum coreness of any given vertex in G , or equivalently, the smallest k such that every subgraph of G contains a vertex of degree at most k .*

The concept of degeneracy is then closely related to the arboricity of a graph.

Definition 2.4 (Arboricity). *The **arboricity** of a graph $G = (V, E)$, which we denote by α , is the minimum number of spanning forests needed to cover the graph.*

In general, α is upper bounded by $O(\sqrt{m})$ and lower bounded by $\Omega(1)$ [72]. Importantly, the degeneracy of a graph is bounded by $\Theta(\alpha)$ [243]. Also, there is a useful upper bound by Chiba and Nishizeki [72] on the sum of the degree of the minimum degree endpoint over all edges in a graph, which we use in some of the results in this thesis. More explicitly, we note that

$$\sum_{(u,v) \in E} \min(\deg(u), \deg(v)) = O(\alpha m).$$

We also use low out-degree orientations in this thesis, which we define as follows.

Definition 2.5 (a -orientation). *Given a graph $G = (V, E)$, an **a -orientation** of G is a total ordering on the vertices such that when edges in the graph are directed from vertices lower in the ordering to vertices higher in the ordering, the out-degree of each vertex is bounded by a .*

Definition 2.6 (Low Out-degree Orientation). *Given a graph $G = (V, E)$, a **low out-degree orientation** of G is an orientation of its edges such that the out-degree of every vertex is bounded by $O(\alpha)$. Note that an **$O(\alpha)$ -orientation**, or an acyclic orientation where all out-degrees are bounded by $O(\alpha)$, is a low out-degree orientation.*

Shi *et al.* [296] and Besta et al. [40] provide parallel work-efficient algorithms for computing an $O(\alpha)$ -orientation, which take $O(m)$ work and $O(\log^2 n)$ span, and which we use in this thesis. These $O(\alpha)$ -orientation algorithms are introduced in more detail in Chapter 5.

2.1.2 Other Graph Concepts

We use several other standard graph concepts in this thesis. An **k -clique** is a set of vertices such that all $\binom{k}{2}$ edges exist among them. We use throughout this thesis algorithms for k -clique enumeration by Shi *et al.* [296], which take $O(m\alpha^{k-2})$ work and $O(\log^2 n)$ span w.h.p.. These algorithms are introduced in Chapter 5.

A **connected component** in an undirected graph $G = (V, E)$ is a maximal subgraph such that all vertices in the subgraph are reachable from one another. Computing the connected components in a graph can be done in $O(m)$ work and $O(\log n)$ span w.h.p. [143].

2.1.3 Graph Storage

Unless otherwise specified, for our theoretical analysis, we assume that our graphs are represented in adjacency hash tables, where each vertex is associated with a parallel hash table of its neighbors. This allows us to obtain constant time edge update queries, and allows us to perform efficient intersection subroutines.

In our implementations, we store our graphs in *compressed sparse row (CSR)* format, in which each graph consists of an array of vertices and an array of edges. In the array of vertices, each vertex points to the beginning of the list of its neighbors in the array of edges, where neighborhoods are stored contiguously in the array of edges. This format has better cache locality in practice.

2.2 Model of Computation

In our theoretical analysis in this thesis, we use the fundamental *work-span model* with arbitrary forking, which is widely used in analyzing shared-memory parallel algorithms [177, 84], and which has many recent practical uses [303, 105, 311, 334]. A computation in this model can be viewed as a series-parallel DAG where each instruction is a vertex, sequential instructions are composed in series, and the forked processes are composed in parallel. The *work* W of an algorithm is the total number of operations, or the total number of vertices in the DAG, and the *span* S of an algorithm is the longest dependency path, or the longest directed path in the DAG. Brent’s scheduling theorem [58] upper bounds the parallel running time of an algorithm by $W/P + S$ where P is the number of processors. A randomized work-stealing scheduler, such as that in Cilk [53], can be used in practice to obtain a parallel running time of $W/P + O(S)$ in expectation. The goal of our work is to develop *work-efficient* parallel algorithms under this model, or algorithms with a work complexity that asymptotically matches the best-known sequential time complexity for the given problem.

In our theoretical bounds, note that we say $O(f(n))$ *with high probability (w.h.p.)* to denote $O(cf(n))$ with probability at least $1 - 1/n^c$ for any $c \geq 1$, where n is the input size.

2.2.1 Atomic Operations

We assume that this model supports concurrent reads, concurrent writes, compare-and-swaps, atomic adds, atomic max, and fetch-and-adds in $O(1)$ work and span.

More concretely, a *compare-and-swap*($\&x, old, new$) takes as input a memory location x , an old value old , and a new value new . It checks if the value stored in x is equal to old , and if so, atomically updates the value at x to be new and returns *true*. Otherwise, it returns *false*.

An *atomic-add*($\&x, v$) takes as input a memory location x and a value v . It atomically retrieves the value stored in x , adds v , and stores the result in x .

An *atomic-max*($\&x, v$) (or a *write-max*) takes as input a memory location x

and a value v . It atomically retrieves the value stored in x , and stores the maximum of the retrieved value and v back in x .

Finally, a *fetch-and-add*($\&x$) atomically returns the value currently in x and then increments the value in x .

2.3 Parallel Primitives and Data Structures

We use the following parallel primitives and data structures throughout this thesis. Note that we use the notation $[n]$ to refer to the range of integers $[1, \dots, n]$.

2.3.1 Parallel Sequence Operations

First, we describe parallel primitives that take as input a sequence A of length n .

The *reduce* operation additionally takes as input a binary associative function \oplus , and returns the sum of the elements in A with respect to \oplus . We typically take \oplus to be addition, or $+$, and refer to sum-reduction as *reduce-add*. Both reduce and reduce-add take $O(n)$ work and $O(\log n)$ span (assuming \oplus takes $O(1)$ work) [177].

The *prefix sum* operation takes as input an identity element ε and an associative binary operator \oplus , and returns the sequence B of length n where $B[i] = \bigoplus_{j < i} A[j] \oplus \varepsilon$. It also returns the overall sum of the elements in A with respect to \oplus . We also commonly take \oplus to be addition. Prefix sum takes $O(n)$ work and $O(\log n)$ span (assuming \oplus takes $O(1)$ work) [177].

The *suffix-min* operation is a special case of the prefix sum operation, excepted performed on the reverse of A and using min as the operator. Specifically, it returns a sequence B of length n such that $B[i] = \min(A[i+1], A[i+2], \dots, A[n-1])$.

Finally, the *filter* operation takes a predicate function f , and returns the sequence B containing elements $a \in A$ such that $f(a)$ is true, in the same order that these elements appeared in A . Note that filter can be implemented using prefix sum. Filter can be performed in $O(n)$ work and $O(\log n)$ span [177].

2.3.2 Parallel Aggregation

We also use several parallel primitives in our algorithms for aggregating equal keys together.

A parallel *semisort* groups together equal keys such that they appear contiguously but makes no guarantee on total order. The main purpose is simply to aggregate together equal keys. For a sequence of length n , parallel semisort takes $O(n)$ expected work and $O(\log n)$ span w.h.p., assuming a uniformly random hash function that maps keys to integers in the range $[n^{O(1)}]$ [154].

Parallel *radix sort*, or parallel *integer sort* sorts a sequence of n integers. It takes $O(n)$ work and $O(\log n)$ span w.h.p. given that the range of the integers is bounded by $O(n \log^{O(1)} n)$ [265].

Additionally, we use parallel *hash tables*, that support n insertion or deletions in $O(n)$ work and $O(\log^* n)$ span w.h.p., and n membership queris in $O(n)$ work and

$O(1)$ span w.h.p. [148]. Given two parallel hash tables \mathcal{T}_1 and \mathcal{T}_2 of size n_1 and n_2 respectively, the intersection $\mathcal{T}_1 \cap \mathcal{T}_2$ can be computed in $O(\min(n_1, n_2))$ work and $O(\log(n_1 + n_2))$ span w.h.p. For multiple hash tables \mathcal{T}_i for $i \in [m]$, each of size n_i , the intersection $\bigcap_{i \in [m]} \mathcal{T}_i$ can be computed in $O(\min_{i \in [m]}(n_i))$ work and $O(\log(\sum_{i \in [m]} n_i))$ span w.h.p. These subroutines are essential to our clique counting and listing subroutines, allowing us to efficiently find the intersections of adjacency lists of vertices.

Finally, like a semisort, a parallel *histogram* similarly aggregates together equal keys, except using a combination of parallel semisorting and hashing. It also returns the frequencies of each key, and for a sequence of length n , it takes $O(n)$ expected work and $O(\log n)$ span w.h.p. [95].

2.3.3 Parallel Bucketing

We frequently use a parallel *bucketing structure* in our decomposition algorithms, which maintains a mapping from identifiers or objects to buckets. The bucket value of identifiers can change throughout the algorithm, and the structure can efficiently update the buckets containing these identifiers. The structure can also efficiently return the identifiers in the minimum bucket. The common paradigm is to store vertices or cliques as the identifiers, in buckets associated with some metric corresponding to each vertex or clique, such as higher-order clique counts, and then repeatedly remove the bucket of objects with the minimum metric. Removing a set of objects often causes the metrics on the remaining objects to change, leading to update operations on the bucketing structure.

For our work on k -core and bi-core decomposition in Chapters 6 and 7 respectively, and in general for most of our implementations, we use the bucketing structure Julianne designed by Dhulipala *et al.* [95]. This bucketing structure, storing n objects, supports b insertion operations or b update operations in $O(b)$ expected work and $O(\log n)$ span w.h.p.. The total cost of repeatedly extracting the minimum bucket, until the bucketing structure is empty, takes work proportional to the range of the buckets. Also, each extraction operation takes logarithmic span w.h.p. This structure obtains performance improvements in practice by only materializing a constant number of the lowest buckets, reducing the number of times each object's bucket must be updated. In retrieving new buckets, the structure also skips over large ranges of empty buckets containing no objects, allowing for fast retrieval of the minimum non-empty bucket.

However, for theoretical purposes in decomposition algorithms referencing higher-order structures, we use the bucketing structure based on a batch-parallel Fibonacci heap by Shi and Shun [298], which is introduced in more detail in Chapter 8. This batch-parallel Fibonacci heap can be used to implement a bucketing structure storing n objects that supports b bucket insertions in $O(b)$ amortized expected work and $O(\log n)$ span w.h.p., b bucket update operations in $O(b)$ amortized work and $O(\log^2 n)$ span w.h.p., and extracts the minimum bucket in $O(\log n)$ amortized expected work and $O(\log n)$ span w.h.p. Also, our batch-parallel Fibonacci heap takes space linear in the number of objects. Importantly, our bounds for the batch-parallel Fibonacci heap are not dependent on the range of the buckets, only on the number

of objects.

Note that throughout this thesis, we always make it clear which bucketing structure we use.

2.3.4 Other Parallel Data Structures

We use other classic parallel data structures and algorithms in this thesis.

First, *list ranking* takes a linked list as input and returns the distance of each element to the end of the linked list. For a linked list of n elements, list ranking can be solved in $O(n)$ work and $O(\log n)$ span [177]. In our algorithms, we use list ranking to compute a unique identifier for each element in a linked list so that we can write them to an array in parallel.

A *union-find* data structure is used to represent collections of sets and supports the following two operations: *unite*(x, y) joins the sets that contain x and y and *find*(x) returns the identifier of the set containing x . Trees are used to implement union-find data structures, where elements have a parent pointer, and the root of a tree corresponds to the representative of the set. Path compression can be done during unite and find operations to shortcut the pointers of elements traversed to point to the root of the set. We use the concurrent union-find data structure by Jayanti et al. [178] in this thesis.

2.4 Experimental Setup

We summarize in this section the shared-memory multicore machines and overall setup that we use in our experimental evaluations. Our experimental setup has varied, since the algorithms in this thesis were developed over several years and commodity multicore platforms have evolved over time. We have also used machines of varying sizes to accommodate larger graphs. The real-world graphs that we experiment on include bipartite graphs from the Koblenz Network Collection (KONECT) [199], undirected graphs from the Stanford Network Analysis Project (SNAP) [207], DIMACS Shortest Paths challenge road networks [93], and graphs from the Network Repository [269]. We also experiment on symmetrized versions of the Twitter network [200], ClueWeb, a Web graph from CMU’s Lemur project [54], and Hyperlink2012 and Hyperlink2014, hyperlink graphs from the WebDataCommons dataset [233]. We additionally use datasets from the UCI Machine Learning repository [114] and the ANN-Benchmarks collection [21].

All of our programs are written in C++ and are compiled using `g++` and the `-O3` flag, although we have used several versions of `g++` over the years, including versions 7.3.1, 7.4.0, 7.5.0, and 8.2.1, which are specified for each experiment. We also use parallel primitives from PARLAYLIB by Blelloch *et al.* [46], and we have used various parallel schedulers to give the best overall performance for each problem, including the work-stealing scheduler PARLAYLIB by Blelloch *et al.* [46], Cilk Plus’s work-stealing scheduler [53, 206], and OpenMP.

The specifications of the machines that we have used are listed below.

30-core Intel machine. A `c2-standard-60` Google Cloud instance, which consists of 30 cores (with two-way hyper-threading), with 3.1GHz Intel Xeon Scalable (Cascade Lake, 2nd generation; 3.8GHz all-core turbo frequency, 3.9GHz single-core max turbo frequency) processors and 240GB of main memory.

36-core Intel machine. A `c5.18xlarge` AWS EC2 instance, which is a dual-processor system with 18 cores per processor (with two-way hyper-threading, 3.00GHz Intel Xeon Platinum 8124M processors, 3.4GHz all-core turbo frequency, 3.5GHz single-core max turbo frequency), and 144GiB of main memory.

48-core Intel machine. A `m5d.24xlarge` AWS EC2 instance, which consists of 48 cores (with two-way hyper-threading), with Intel Xeon Platinum 8175 processors (3.1GHz all-core turbo frequency), 384GiB of main memory, and 4×900 NVMe SSD.

48-core Intel machine. A `m1-megamem-96` Google Cloud instance, which consists of 48 cores (with two-way hyper-threading), with 2.0GHz Intel Xeon Scalable (Sky-lake, 1st generation; 2.7GHz all-core turbo frequency, 3.5GHz single-core max turbo frequency) processors and 1433.6GB of main memory.

72-core Intel machine. A 72-core Dell PowerEdge R930 (with two-way hyper-threading), with 4×2.4 GHz Intel 18-core E7-8867 v4 Xeon processors and 1TB of main memory.

80-core Intel machine. A `m1-ultramem-160` Google Cloud instance, which consists of 80 cores (with two-way hyper-threading), with 2.2GHz Intel Xeon E7 (Broadwell E7; 2.6GHz all-core turbo frequency, 3.3GHz single-core max turbo frequency) processors and 3844GB of main memory.

Part I

Subgraph Counting and Listing

Introduction

In the absence of ground-truth data, one of the key methods to determining the quality of graph clusterers is by investigating the makeup of small subgraphs within these clusters in relation to the overarching graph [336, 317, 32, 321]. Additionally, graphlet kernels and subgraph-based feature extraction for graphs are important steps in many machine learning pipelines for graph classification and community detection [167, 294, 270, 156]. Discovering specific subgraphs, such as k -cliques, can also contribute directly to community detection and graph partitioning tasks, as subroutines to more complex algorithms [32, 321]. Outside of clustering-related applications, subgraph counting in general has widespread applications in network analytics across various domains including bioinformatics and social network analysis [166, 62, 129]. We also note that cycle counting specifically has further important applications, including spam and fraud detection [31], and link classification and recommendation [319].

However, subgraph counting is a difficult problem particularly for subgraphs of larger sizes. The number of possible subgraphs grows exponentially with the subgraph size, so as such, discovering and counting these subgraphs naively can be computationally expensive. For instance, five-cycle counting is particularly computationally intensive and takes up between 25–58% of the total running time of the state-of-the-art serial Efficient Subgraph Counting Algorithmic PackagE (ESCAPE) [258] for general five-vertex subgraph counting. ESCAPE is unable to complete five-cycle counting on a graph with 200 million edges (com-orkut [207]) in 5.5 hours on an 18-core machine with 144 GiB of main memory. Additionally, for certain graphs, even state-of-the-art subgraph counting implementations for specific subgraphs, notably cliques, encounter memory limitations. Jain and Seshadhri’s PIVOTER [176], Danisch et al.’s kCLIST [86], Mhedhbi and Salihoglu’s worst-case optimal join algorithm [234], and Lai *et al.*’s binary join algorithm [201] all run out of memory when counting four-cliques on graphs with billions to hundreds of billions of edges (ClueWeb [85], Hyperlink2014 [233], Hyperlink2012 [233]) on a machine with 80 cores and 3844 GiB of main memory.

We overcome these difficulties in this part of the thesis by exploiting common structural characteristics of real-world graphs. In particular, real-world graphs are often sparse and exhibit low arboricity. We develop in Chapters 5 and 6 algorithms to efficiently obtain acyclic low out-degree orientations of these graphs, which we can then use to prune the search space of directed subgraphs. Even with the use of low out-degree orderings, though, we note that there are theoretical and practical barriers to efficiently counting large subgraphs in parallel in practice. We address these barriers in

various ways. We explore the tradeoffs between space usage and performance in both our theoretical guarantees and our implementations, and we introduce approximation algorithms with unbiased estimators that offer much more significant speedups. We also explore different data structures and orientation algorithms, which may not be work-efficient but which offer better performance with lower parallel overheads in practice. We apply these techniques to butterfly (four-cycle) counting in Chapter 3, five-cycle counting in Chapter 4, k -clique counting and listing in Chapter 5, and batch-dynamic k -clique counting in Chapter 6. Our implementations are able to complete five-cycle counting on a graph with 200 million edges (com-orkut [207]) in under 3 minutes, and four-clique counting on graphs with billions to hundreds of billions of edges (ClueWeb [85], Hyperlink2014 [233], Hyperlink2012 [233]) in 2 hours, 4 hours, and 45 hours respectively, on the same corresponding machines as previously mentioned.

The results in this part of the thesis have appeared in the following publications.

- Jessica Shi and Julian Shun. “Parallel Algorithms for Butterfly Computations”. In: *Massive Graph Analytics*, pp. 287-330, 2022. (Chapter 3)
- Jessica Shi and Julian Shun. “Parallel Algorithms for Butterfly Computations”. In: *Proceedings of the SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pp. 16-30, 2020. (Earlier version)
- Jessica Shi, Louisa Huang, and Julian Shun. “Parallel Five-Cycle Counting Algorithms”. In: *ACM Journal of Experimental Algorithmics (JEA)*, Vol. 27, Article No. 4:1, pp.1-23, 2022. (Chapter 4)
- Louisa Huang, Jessica Shi, and Julian Shun. “Parallel Five-Cycle Counting Algorithms”. In: *Proceedings of the International Symposium on Experimental Algorithms (SEA)*, pp. 2:1-2:18, 2021. (Earlier version)
- Jessica Shi, Laxman Dhulipala, and Julian Shun. “Parallel Clique Counting and Peeling Algorithms”. In: *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*, pp. 135-146, 2021. (Chapter 5)
- Quanquan Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. “Parallel Batch-Dynamic Algorithms for k -Core Decomposition and Related Graph Problems”. In: *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 191-204, 2022. (Chapter 6)

Chapter 3

Butterfly Counting

3.1 Introduction

Triangles are a fundamental graph motif in unipartite graphs, and indeed, triangle counting is a key metric that is widely applicable in areas including social network analysis [246], spam and fraud detection [31], and link classification and recommendation [319]. However, many real-world graphs are bipartite and model the affiliations between two groups. For example, bipartite graphs are used to represent peer-to-peer exchange networks (linking peers to the data they request), group membership networks (e.g., linking actors to movies they acted in), recommendation systems (linking users to items they rated), factor graphs for error-correcting codes, and hypergraphs [56, 204]. Bipartite graphs contain no triangles; the smallest non-trivial subgraph is a *butterfly* (also known as rectangles), which is a $(2, 2)$ -biclique (containing two vertices on each side and all four possible edges among them), and thus having efficient algorithms for counting butterflies is crucial for applications on bipartite graphs [325, 12, 277]. Notably, butterfly counting has applications in link spam detection [147] and document clustering [94]. An example bipartite graph and its butterflies is shown in Figure 3.1.

There has been recent work on designing efficient sequential algorithms for butterfly counting [72, 325, 354, 277]. However, given the high computational requirements of butterfly computations, it is natural to study whether we can obtain performance improvements using parallel machines. This chapter presents a framework for butterfly computations, called PARBUTTERFLY, that enables us to obtain new parallel algorithms for butterfly counting. PARBUTTERFLY is a modular framework that enables us to easily experiment with many variations of our algorithms. We not only show that our algorithms are efficient in practice, but also prove strong theoretical bounds on their work and span. Given that all real-world bipartite graphs fit on a multicore machine, we design parallel algorithms for this setting.

For butterfly counting, the main procedure involves finding wedges (2-paths) and combining them to count butterflies. See Figure 3.1 for an example of wedges. In particular, we want to find all wedges originating from each vertex, and then aggregate the counts of wedges incident to every distinct pair of vertices forming the endpoints

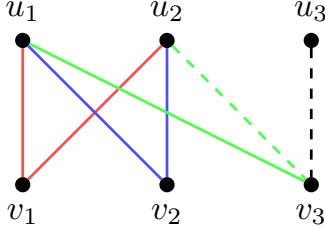


Figure 3.1: The butterflies in this graph are $\{u_1, v_1, u_2, v_2\}$, $\{u_1, v_1, u_2, v_3\}$, and $\{u_1, v_2, u_2, v_3\}$. The red, blue, and green edges each produce a wedge ($\{u_1, v_1, u_2\}$, $\{u_1, v_2, u_2\}$, and $\{u_1, v_3, u_2\}$). Note that these wedges all share the same endpoints, namely u_1 and u_2 . Thus, any combination of two of these wedges forms a butterfly. For example, $\{u_1, v_1, u_2\}$ and $\{u_1, v_2, u_2\}$ combine to form a butterfly $\{u_1, v_1, u_2, v_2\}$. However, the dashed edges produce another wedge, $\{u_2, v_3, u_3\}$, which has different endpoints, namely u_2 and u_3 . This wedge cannot be combined with any of the previous wedges to form a butterfly.

of the wedge. With these counts, we can obtain global, per-vertex, and per-edge butterfly counts. The PARBUTTERFLY framework provides different ways to aggregate wedges in parallel, including sorting, hashing, histogramming, and batching. Also, we can speed up butterfly counting by ranking vertices and only considering wedges formed by a particular ordering of the vertices. PARBUTTERFLY supports different parallel ranking methods, including side-ordering, approximate and exact degree-ordering, and approximate and exact complement-coreness ordering. These orderings can be used with any of the aggregation methods. To further speed up computations on large graphs, PARBUTTERFLY also supports parallel approximate butterfly counting via graph sparsification based on ideas by Sanei-Mehri *et al.* [277] for the sequential setting. Furthermore, we integrate into PARBUTTERFLY a recently proposed cache optimization for butterfly counting by Wang *et al.* [328].

We prove theoretical bounds showing that some variants of our counting algorithms are highly parallel and match the work of the best sequential algorithm. For a graph $G(V, E)$ with m edges and arboricity α , PARBUTTERFLY gives a counting algorithm that takes $O(\alpha m)$ expected work, $O(\log m)$ span with high probability (w.h.p.), and $O(\min(n^2, \alpha m))$ additional space.

We present a comprehensive experimental evaluation of all of the different variants of counting algorithms in the PARBUTTERFLY framework. On a 48-core machine, our counting algorithms achieve self-relative speedups of up to 38.5x and outperform the fastest sequential baseline by up to 13.6x. Compared to PGD [7], a state-of-the-art parallel subgraph counting solution that can be used for butterfly counting as a special case, we are 349.6–5169x faster. We find that although the sorting, hashing, and histogramming aggregation approaches achieve better theoretical complexity, batching usually performs the best in practice due to lower overheads.

In summary, the contributions of this chapter are as follows.

- (1) New parallel algorithms for butterfly counting.

- (2) A framework PARBUTTERFLY with different ranking and wedge aggregation schemes that can be used for parallel butterfly counting.
- (3) Strong theoretical bounds on algorithms obtained using PARBUTTERFLY.
- (4) A comprehensive experimental evaluation on a 48-core machine demonstrating high parallel scalability and fast running times compared to the best sequential baselines, as well as significant speedups over the state-of-the-art parallel subgraph counting solution.

The PARBUTTERFLY code can be found at <https://github.com/jeshi96/parbutterfly>.

3.2 Preliminaries

In this chapter, we take every bipartite graph $G = (U, V, E)$ to be simple and undirected.

A **butterfly** is a set of four vertices $u_1, u_2 \in U$ and $v_1, v_2 \in V$ with edges $(u_1, v_1), (u_1, v_2), (u_2, v_1), (u_2, v_2) \in E$. A **wedge** is a set of three vertices $u_1, u_2 \in U$ and $v \in V$, with edges $(u_1, v), (u_2, v) \in E$. We call the vertices u_1, u_2 **endpoints** and the vertex v the **center**. Symmetrically, a wedge can also consist of vertices $v_1, v_2 \in V$ and $u \in U$, with edges $(v_1, u), (v_2, u) \in E$. We call the vertices v_1, v_2 endpoints and the vertex u the center. We can decompose a butterfly into two wedges that share the same endpoints but have distinct centers.

We store our graphs in compressed sparse row (CSR) format, which requires $O(m + n)$ space. We initially maintain separate offset and edge arrays for each vertex partition U and V , and we assume that all arrays are stored consecutively in memory.

3.3 PARBUTTERFLY Framework

In this section, we describe the PARBUTTERFLY framework and its components. Section 3.3.1 describes the procedures for counting butterflies. Section 3.4 goes into more detail on the parallel algorithms that can be plugged into the framework, as well as their analysis.

3.3.1 Counting Framework

Figure 3.2 shows the high-level structure of the PARBUTTERFLY framework. Step 1 assigns a global ordering to the vertices, which helps reduce the overall work of the algorithm. Step 2 retrieves all the wedges in the graph, but only where the second and third vertices of the wedge have higher rank than the first. Step 3 counts for every pair of vertices the number of wedges that share those vertices as endpoints. Step 4 uses the wedge counts to obtain global, per-vertex, or per-edge butterfly counts. For each step, there are several options with respect to implementation, each of which can be independently chosen and used together. Figure 3.3 shows an example of executing

PARBUTTERFLY Framework for Counting

- (1) *Rank vertices*: Assign a global ordering, `rank`, to the vertices.
- (2) *Retrieve wedges*: Retrieve a list W of wedges (x, y, z) where $\text{rank}(y) > \text{rank}(x)$ and $\text{rank}(z) > \text{rank}(x)$.
- (3) *Count wedges*: For every pair of vertices (x_1, x_2) , how many distinct wedges share x_1 and x_2 as endpoints.
- (4) *Count butterflies*: Use the wedge counts to obtain the global butterfly count, per-vertex butterfly counts, or per-edge butterfly counts.

Figure 3.2: The PARBUTTERFLY framework for counting.

| Step 1 | Step 2 | Step 3 | Step 4 |
|----------|---|---|--|
| 1. v_3 | | $\left\{ ((v_3, v_1), u_1), ((v_3, v_2), u_1) \right\} \Rightarrow$ | $+ \binom{2}{2}$ butterflies on v_1, v_3 |
| 2. u_1 | $v_3 \left\{ ((v_3, v_1), u_1), ((v_3, v_2), u_1) \right\}$ | $\left\{ ((v_3, v_1), u_2), ((v_3, v_2), u_2) \right\} \Rightarrow$ | $+ (2-1)$ butterflies on u_1, u_2 |
| 3. u_2 | $\left\{ ((v_3, v_1), u_2), ((v_3, v_2), u_2) \right\}$ | $\left\{ ((v_3, v_2), u_1), ((v_3, v_2), u_2) \right\} \Rightarrow$ | $+ \binom{2}{2}$ butterflies on v_2, v_3 |
| 4. v_1 | | $\left\{ ((u_1, u_2), v_1), ((u_1, u_2), v_2) \right\} \Rightarrow$ | $+ (2-1)$ butterflies on u_1, u_2 |
| 5. v_2 | $u_1 \left\{ ((u_1, u_2), v_1), ((u_1, u_2), v_2) \right\}$ | $\left\{ ((u_1, u_2), v_1), ((u_1, u_2), v_2) \right\} \Rightarrow$ | $+ \binom{2}{2}$ butterflies on u_1, u_2 |
| 6. u_3 | | | $+ (2-1)$ butterflies on v_1, v_2 |

Figure 3.3: We execute butterfly counting per vertex on the graph in Figure 3.1. In Step 1, we rank vertices in decreasing order of degree. In Step 2, for each vertex v in order, we retrieve all wedges where v is an endpoint and where the other two vertices have higher rank (the wedges are represented as $((x, z), y)$ where x and z are endpoints and y is the center). In Step 3, we aggregate wedges by their endpoints, and this produces the butterfly counts for Step 4. Note that if we have w wedges that share the same endpoint, this produces $\binom{w}{2}$ butterflies for each of the two endpoints and $w - 1$ butterflies for each of the centers of the w wedges.

each of the steps. The options within each step of PARBUTTERFLY are described in the rest of this section.

3.3.1.1 Ranking

The ordering of vertices when we retrieve wedges is significant since it affects the number of wedges that we process. As we discuss in Section 3.4.1, Sanei-Mehri *et*

al. [277] order all vertices from one bipartition of the graph first, depending on which bipartition produces the least number of wedges, giving them practical speedups in their serial implementation. We refer to this ordering as *side order*. Chiba and Nishizeki [72] achieve a lower work complexity for counting by ordering vertices in decreasing order of degree, which we refer to as *degree order*.

For practical speedups, we also introduce *approximate degree order*, which orders vertices in decreasing order of the logarithm of their degree (*log-degree*). Since the ordering of vertices in many real-world graphs have good locality, approximate degree order preserves the locality among vertices with equal log-degree. We show in Section 3.4.4 that the work of butterfly counting using approximate degree order is the same as that of using degree order.

Degeneracy order, also known as the ordering given by vertex coreness, is a well-studied ordering of vertices given by repeatedly finding and removing vertices of smallest degree [290, 229]. This ordering can be obtained serially in linear time using a k -core decomposition algorithm [229], and in parallel in linear work by repeatedly removing (peeling) all vertices with the smallest degree from the graph in parallel [95]. The span of peeling is proportional to the number of peeling rounds needed to reduce the graph to an empty graph. We define **complement degeneracy order** to be the ordering given by repeatedly removing vertices of largest degree. This mirrors the idea of decreasing order of degree, but encapsulates more structural information about the graph.

However, using complement degeneracy order is not efficient. The span of finding complement degeneracy order is limited by the number of rounds needed to reduce a graph to an empty graph, where each round deletes all maximum degree vertices of the graph. As such, we define **approximate complement degeneracy order**, which repeatedly removes vertices of largest log-degree. This reduces the number of rounds needed and closely approximates the number of wedges that must be processed using complement degeneracy order. We implement both of these using the parallel bucketing structure of Dhulipala et al. [95].

We show in Section 3.4.5 that using complement degeneracy order and approximate complement degeneracy order give the same work-efficient bounds as using degree order. We show in Section 3.5 that empirically, the same number or fewer wedges must be processed (compared to both side and degree order) if we consider vertices in complement degeneracy order or approximate complement degeneracy order.

In total, the options for ranking are side order, degree order, approximate degree order, complement degeneracy order, and approximate complement degeneracy order.

3.3.1.2 Wedge Aggregation

We obtain wedge counts by aggregating wedges by endpoints. PARBUTTERFLY implements fully-parallel methods for aggregation including sorting, hashing, and histogramming, as well as a partially-parallel batching method.

We can aggregate the wedges by semisorting key-value pairs where the key is the two endpoints and the value is the center. Then, all elements with the same key are

grouped together, and the size of each group is the number of wedges shared by the two endpoints. We implemented this approach using parallel sample sort from the Problem Based Benchmark Suite (PBBS) [48, 300] due to its better cache-efficiency over parallel semisort.

We can also use a parallel hash table to store key-value pairs where the key is two endpoints and the value is a count. We insert the endpoints of all wedges into the table with value 1, and sum the values on duplicate keys. The value associated with each key then represents the number of wedges that the two endpoints share. We use a parallel hash table based on linear probing with an atomic addition combining function [299].

Another option is to insert the key-value pairs into a parallel histogramming structure which counts the number of occurrences of each distinct key. The parallel histogramming structure that we use is implemented using a combination of semisorting and hashing [95].

Finally, in our partially-parallel batching method we process a batch of vertices in parallel and find the wedges incident on these vertices. Each vertex aggregates its wedges serially, using an array large enough to contain all possible second endpoints. The *simple* setting in our framework fixes the number of vertices in a batch as a constant based on the space available, while the *wedge-aware* setting determines the number of vertices dynamically based on the number of wedges that each vertex processes.

In total, the options for combining wedges are sorting, hashing, histogramming, simple batching, and wedge-aware batching.

3.3.1.3 Butterfly Aggregation

There are two main methods to translate wedge counts into butterfly counts, per-vertex or per-edge.¹ One method is to make use of atomic adds, and add the obtained butterfly count for the given vertex/edge directly into an array, allowing us to obtain butterfly counts without explicit re-aggregation.

The second method is to reuse the aggregation method chosen for the wedge counting step and use sorting, hashing, or histogramming to combine the butterfly counts per-vertex or per-edge.²

3.3.1.4 Other Options

There are a few other options for butterfly counting in PARBUTTERFLY. First, butterfly counts can be computed per vertex, per edge, or in total. For wedge aggregation methods apart from batching, since the number of wedges can be quadratic in the size of the original graph, it may not be possible to fit all wedges in memory at once; a parameter in our framework takes into account the number of wedges that can be handled in memory and processes subsets of wedges using the chosen aggregation

¹For total counts, butterfly counts can simply be computed and summed in parallel directly.

²Note that this is not feasible for partially-parallel batching, so in that case, the only option is to use atomic adds.

method until they are all processed. Similarly, for wedge aggregation by batching, a parameter takes into account the available space and appropriately determines the number of vertices per batch.

PARBUTTERFLY also implements both edge and colorful sparsification as described by Sanei-Mehri *et al.* [277] to obtain approximate total butterfly counts. For approximate counting, the sub-sampled graph is simply passed to the framework shown in Figure 3.2 using any of the aggregation and ranking choices, and the final result is scaled appropriately. Note that this can only be used for total counts.

Finally, Wang *et al.* [328] independently describe an algorithm for butterfly counting using degree ordering, as done in Chiba and Nishizeki [72], and also propose a cache optimization for wedge retrieval. Their cache optimization involves retrieving precisely the wedges given by Chiba and Nishizeki’s algorithm, but instead of retrieving wedges by iterating through the lower ranked endpoint (for every v , retrieve wedges (v, w, u) where w, u have higher rank than v), they retrieve wedges by iterating through the higher ranked endpoint (for every u , retrieve wedges (v, w, u) where w, u have higher rank than v). Inspired by their work, we have augmented PARBUTTERFLY to include this cache optimization for all of our orderings.

3.4 PARBUTTERFLY Algorithms

We describe here our parallel algorithms for butterfly counting in more detail. Our theoretically-efficient parallel algorithms are based on the work-efficient sequential butterfly listing algorithm, introduced by Chiba and Nishizeki [72].

Note that Wang *et al.* [325] proposed the first algorithm for butterfly counting, but their algorithm is not work-efficient. They also give a simple parallelization of their counting algorithm that is similarly not work-efficient. Moreover, Sanei-Mehri *et al.* [277] give a sequential butterfly counting algorithm, but their algorithm is also not work-efficient.

3.4.1 Preprocessing

The main subroutine in butterfly counting involves processing a subset of wedges of the graph; previous work differ in the way in which they choose wedges to process. As mentioned in Section 3.3.1.1, Chiba and Nishizeki [72] choose wedges by first ordering vertices by decreasing order of degree and then for each vertex in order, extracting all wedges with said vertex as an endpoint and deleting the processed vertex from the graph. Note that the ordering of vertices does not affect the correctness of the algorithm – in fact, Sanei-Mehri *et al.* [277] use this precise algorithm but with all vertices from one bipartition of the graph ordered before all vertices from the other bipartition. Importantly, Chiba and Nishizeki’s [72] original decreasing degree ordering gives the work-efficient bounds $O(am)$ on butterfly counting.

Throughout this section, we use decreasing degree ordering to obtain the same work-efficient bounds in our parallel algorithms. However, note that using approximate degree ordering, complement degeneracy ordering, and approximate comple-

Algorithm 3.1 – Preprocessing

```
1: procedure PREPROCESS( $G = (U, V, E), f$ )
2:    $X \leftarrow \text{SORT}(U \cup V, f)$             $\triangleright$  Sort vertices in increasing order of rank according to
      function  $f$ 
3:   Let  $x$ 's rank  $R[x]$  be its index in  $X$ 
4:    $E' \leftarrow \{(R[u], R[v]) \mid (u, v) \in E\}$             $\triangleright$  Rename all vertices to their rank
5:    $G' = (X, E')$ 
6:   parfor  $x \in X$  do
7:      $N_{G'}(x) \leftarrow \text{SORT}(\{y \mid (x, y) \in E'\})$   $\triangleright$  Sort neighbors by decreasing order of rank
8:     Store  $\deg_x(x)$  and  $\deg_y(x)$  for all  $(x, y) \in E'$ 
9:   return  $G'$ 
```

ment degeneracy ordering also gives us these work-efficient bounds; we prove the work-efficiency of these orderings in Sections 3.4.4 and 3.4.5. Furthermore, our exact and approximate counting algorithms work for any ordering; only the theoretical analysis depends on the ordering.

We use **rank** to denote the index of a vertex in some ordering, in cases where the ordering that we are using is clear or need not be specified. We define a modified degree, $\deg_y(x)$, to be the number of neighbors $z \in N(x)$ such that $\text{rank}(z) > \text{rank}(y)$. We also define a modified neighborhood, $N_y(x)$, to be the set of neighbors $z \in N(x)$ such that $\text{rank}(z) > \text{rank}(y)$.

We give a preprocessing algorithm, PREPROCESS (Algorithm 3.1), which takes as input a bipartite graph and a ranking function f , and renames vertices by their rank in the ordering. The output is a general graph (we discard bipartite information in our renaming). PREPROCESS also sorts neighbors by decreasing order of rank.

PREPROCESS begins by sorting vertices in increasing order of rank. Assuming that f returns an integer in the range $[0, n - 1]$, which is true in all of the orderings provided in PARBUTTERFLY, this can be done in $O(n)$ expected work and $O(\log n)$ span w.h.p. with parallel integer sort [265]. Renaming our graph based on vertex rank takes $O(m)$ work and $O(1)$ span (to retrieve the relevant ranks). Finally, sorting the neighbors of our renamed graph and the modified degrees takes $O(m)$ expected work and $O(\log m)$ span w.h.p. (since these ranks are in the range $[0, n - 1]$). All steps can be done in linear space. The following lemma summarizes the complexity of preprocessing.

Lemma 3.1. *Preprocessing can be implemented in $O(m)$ expected work, $O(\log m)$ span w.h.p., and $O(m)$ space.*

3.4.2 Counting Algorithms

In this section, we describe and analyze our parallel algorithms for butterfly counting.

The following equations describe the number of butterflies per vertex and per edge. Sanei-Mehri *et al.* [277] derived and proved the per-vertex equation, as based on Wang *et al.*'s [325] equation for the total number of butterflies. We give a short proof of the per-edge equation.

Algorithm 3.2 – Parallel wedge retrieval

```

1: procedure GET-WEDGES( $G = (V, E), f$ )
2:   parfor  $x_1 \in V$  do
3:     parfor  $i \leftarrow 0$  to  $\deg_{x_1}(x_1)$  do
4:        $y \leftarrow N(x_1)[i]$                                  $\triangleright y = i^{\text{th}}$  neighbor of  $x_1$ 
5:       parfor  $j \leftarrow 0$  to  $\deg_{x_1}(y)$  do
6:          $x_2 \leftarrow N(y)[j]$                                  $\triangleright x_2 = j^{\text{th}}$  neighbor of  $y$ 
7:          $f((x_1, x_2), y)$      $\triangleright (x_1, x_2)$  are the endpoints,  $y$  is the center of the wedge
8:   return  $W$ 

```

Lemma 3.2. *For a bipartite graph $G = (U, V, E)$, the number of butterflies containing a vertex u is given by*

$$\sum_{u' \in N^2(u)} \binom{|N(u) \cap N(u')|}{2}. \quad (3.1)$$

The number of butterflies containing an edge $(u, v) \in E$ is given by

$$\sum_{u' \in N(v) \setminus \{u\}} (|N(u) \cap N(u')| - 1). \quad (3.2)$$

Proof. The proof for the number of butterflies per vertex is given by Sanei-Mehri *et al.* [277]. For the number of butterflies per edge, we note that given an edge $(u, v) \in E$, each butterfly that (u, v) is contained within has additional vertices $u' \in U, v' \in V$ and additional edges $(u', v), (u, v'), (u', v') \in E$. Thus, iterating over all $u' \in N(v)$ (where $u' \neq u$), it suffices to count the number of vertices $v' \neq v$ such that v' is adjacent to u and to u' . In other words, it suffices to count $v' \in N(u) \cap N(u') \setminus \{v\}$. This gives us precisely $\sum_{u' \in N(v) \setminus \{u\}} (|N(u) \cap N(u')| - 1)$ as the number of butterflies containing (u, v) . \square

Note that in both equations given by Lemma 3.2, we iterate over wedges with endpoints u and u' to obtain our desired counts (Step 4 of Figure 3.2). We now describe how to retrieve the wedges (Step 2 of Figure 3.2).

3.4.2.1 Wedge Retrieval

There is a subtle point to make in retrieving all wedges. Once we have retrieved all wedges with endpoint x , Equation (3.1) dictates the number of butterflies that x contributes to the second endpoints of these wedges, and Equation (3.2) dictates the number of butterflies that x contributes to the centers of these wedges. As such, given the wedges with endpoint x , we can count not only the number of butterflies on x , but also the number of butterflies that x contributes to other vertices of our graph. Thus, after processing these wedges, there is no need to reconsider wedges containing x (importantly, there is no need to consider wedges with center x).

From Chiba and Nishizeki's [72] work, to minimize the total number of wedges that we process, we must retrieve all wedges considering endpoints x in decreasing

order of degree, and then delete said vertex from the graph (i.e., do not consider any other wedge containing x).

We introduce here a parallel wedge retrieval algorithm, GET-WEDGES (Algorithm 3.2), that takes $O(\alpha m)$ work and $O(\log m)$ span. We assume that GET-WEDGES takes as input a preprocessed (ranked) graph and a function to apply on each wedge (either for storage or for processing). (The algorithm iterates through all vertices x_1 and retrieves all wedges with endpoint x_1 such that the center and second endpoint both have rank greater than x_1 (Lines 3–7). This is equivalent to Chiba and Nishizeki’s algorithm which deletes vertices from the graph, but the advantage is that since we do not modify the graph, all wedges can be processed in parallel. We process exactly the set of wedges that Chiba and Nishizeki process, and they prove that there are $O(\alpha m)$ such wedges.

Since the adjacency lists are sorted in decreasing order of rank, we can obtain the end index of the loops on Line 5 using an exponential search in $O(\deg_{x_1}(y))$ work and $O(\log(\deg_{x_1}(y)))$ span. Then, iterating over all wedges takes $O(\alpha m)$ work and $O(1)$ span. In total, we have $O(\alpha m)$ work and $O(\log m)$ span.

After retrieving wedges, we have to group together the wedges sharing the same endpoints, and compute the size of each group. We define a subroutine GET-FREQ that takes as input a function that retrieves wedges, and produces a hash table containing endpoint pairs with their corresponding wedge frequencies. This can be implemented using an additive parallel hash table, that increments the count per endpoint pair. Note that parallel semisorting or histogramming could also be used, as discussed in Section 3.3.1. For an input of length n , GET-FREQ takes $O(n)$ expected work and $O(\log n)$ span w.h.p. using any of the three aggregation methods as proven in prior work [154, 148, 95]. However, semisorting and histogramming require $O(n)$ space, while hashing requires space proportional to the number of unique endpoint pairs. In the case of aggregating all wedges, this is $O(\min(n^2, \alpha m))$ space.

The following lemma summarizes the complexity of wedge retrieval and aggregation.

Lemma 3.3. *Iterating over all wedges can be implemented in $O(\alpha m)$ expected work, $O(\log m)$ span w.h.p., and $O(\min(n^2, \alpha m))$ space.*

Note that this is a better worst-case work bound than the work bound of $O(\sum_{v \in V} \deg(v)^2)$ using side order. In the worst-case $O(\alpha m) = O(m^{1.5})$ while $O(\sum_{v \in V} \deg(v)^2) = O(mn)$. We have that $mn = \Omega(m^{1.5})$, since $n = \Omega(m^{0.5})$.

3.4.2.2 Per Vertex

We now describe the full butterfly counting per vertex algorithm, which is given as COUNT-V in Algorithm 3.3. As described previously, we implement preprocessing in Line 10, and we implement wedge retrieval and aggregation by endpoint pairs in Line 3.

We note that following Line 3, by counting the frequency of wedges by endpoints, for each fixed vertex x_1 we have obtained in R a list of all possible endpoints $(x_1, x_2) \in X \times X$ with the size of their intersection $|N(x_1) \cap N(x_2)|$. Thus, by Lemma 3.2, for

Algorithm 3.3 – Parallel work-efficient butterfly counting per vertex

```

1: procedure COUNT-V-WEDGES(GET-WEDGES-FUNC)
2:   Initialize  $B$  to be an additive parallel hash table that stores butterfly counts per
   vertex
3:    $R \leftarrow \text{GET-FREQ}(\text{GET-WEDGES-FUNC})$             $\triangleright$  Aggregate wedges by wedge endpoints
4:   parfor  $((x_1, x_2), d)$  in  $R$  do
5:     Insert  $(x_1, \binom{d}{2})$  and  $(x_2, \binom{d}{2})$  in  $B$             $\triangleright$  Store butterfly counts per endpoint
6:      $f : ((x_1, x_2), y) \rightarrow \text{Insert } (y, R(x_1, x_2) - 1) \text{ in } B$ 
7:    $f : ((x_1, x_2), y) \rightarrow \text{Insert } (y, R(x_1, x_2) - 1) \text{ in } B$             $\triangleright$  Function to store butterfly counts per center
8:   GET-WEDGES-FUNC( $f$ )       $\triangleright$  Iterate over wedges to store butterfly counts per center
9:   return  $B$ 

10: procedure COUNT-V( $G = (U, V, E)$ )
11:    $G' = (X, E') \leftarrow \text{PREPROCESS}(G)$ 
12:   return COUNT-V-WEDGES(GET-WEDGES( $G'$ ))

```

Algorithm 3.4 – Parallel work-efficient butterfly counting per edge

```

1: procedure COUNT-E-WEDGES(GET-WEDGES-FUNC)
2:   Initialize  $B$  to be an additive parallel hash table that stores butterfly counts per edge
3:    $R \leftarrow \text{GET-FREQ}(\text{GET-WEDGES-FUNC})$             $\triangleright$  Aggregate wedges by wedge endpoints
4:    $f : ((x_1, x_2), y) \rightarrow \text{Insert } ((x_1, y), R(x_1, x_2) - 1) \text{ and } ((x_2, y), R(x_1, x_2) - 1) \text{ in } B$ 
5:    $f : ((x_1, x_2), y) \rightarrow \text{Insert } ((x_1, y), R(x_1, x_2) - 1) \text{ and } ((x_2, y), R(x_1, x_2) - 1) \text{ in } B$             $\triangleright$  Function to store butterfly counts per edge
6:   GET-WEDGES-FUNC( $f$ )       $\triangleright$  Iterate over wedges to store butterfly counts per edge
7:   return  $B$ 

8: procedure COUNT-E( $G = (U, V, E)$ )
9:    $G' = (X, E') \leftarrow \text{PREPROCESS}(G)$ 
10:  return COUNT-E-WEDGES(GET-WEDGES( $G'$ )))

```

each endpoint x_2 , x_1 contributes $\binom{|N(x_1) \cap N(x_2)|}{2}$ butterflies, and for each center y (as given in W), x_1 contributes $|N(x_1) \cap N(x_2)| - 1$ butterflies. As such, we compute the per-vertex counts by iterating through R to add the requisite count to each endpoint (Line 5) and iterating through all wedges using GET-WEDGES to add the requisite count to each center (Lines 6–8).

Extracting the butterfly counts from our wedges takes $O(\alpha m)$ work (since we are iterating through all wedges) and $O(1)$ span, and as discussed earlier, GET-FREQ takes $O(\alpha m)$ expected work, $O(\log m)$ span w.h.p., and $O(\min(n^2, \alpha m))$ space. The total complexity of butterfly counting per vertex is given as follows.

Theorem 3.1. *Butterfly counting per vertex can be performed in $O(\alpha m)$ expected work, $O(\log m)$ span w.h.p., and $O(\min(n^2, \alpha m))$ space.*

3.4.2.3 Per Edge

We now describe the full butterfly counting per edge algorithm, which is given as COUNT-E in Algorithm 3.4. We implement preprocessing and wedge retrieval as described previously, in Line 9 and Line 3, respectively.

As we discussed in Section 3.4.2.2, following Step 3 for each fixed vertex x_1 we have in R a list of all possible endpoints $(x_1, x_2) \in X \times X$ with the size of their intersection $|N(x_1) \cap N(x_2)|$. Thus, by Lemma 3.2, we compute per-edge counts by iterating through all of our wedge counts and adding $|N(x_1) \cap N(x_2)| - 1$ to our butterfly counts for the edges contained in the wedges with endpoints x_1 and x_2 . As such, we use GET-WEDGES to iterate through all wedges, look up in R the corresponding count $|N(x_1) \cap N(x_2)| - 1$ on the endpoints, and add this count to the corresponding edges.

Extracting the butterfly counts from our wedges takes $O(\alpha m)$ work (since we are essentially iterating through W) and $O(1)$ span. GET-FREQ takes $O(\alpha m)$ expected work, $O(\log m)$ span w.h.p., and $O(\min(n^2, \alpha m))$ space. The total complexity of butterfly counting per edge is given as follows.

Theorem 3.2. *Butterfly counting per edge can be performed in $O(\alpha m)$ expected work, $O(\log m)$ span w.h.p., and $O(\min(n^2, \alpha m))$ space.*

3.4.3 Approximate Counting

Sanei-Mehri *et al.* [277] describe for computing approximate total butterfly counts based on sampling and graph sparsification. Their sparsification methods are shown to have better performance, and so we focus on parallelizing these methods. The methods are based on creating a sparsified graph, running an exact counting algorithm on the sparsified graph, and scaling up the count returned to obtain an unbiased estimate of the total butterfly count.

The **edge sparsification** method sparsifies the graph by keeping each edge independently with probability p . The butterfly count of the sparsified graph is divided by p^4 to obtain an unbiased estimate (since each butterfly remains in the sparsified graph with probability p^4). Our parallel algorithm simply applies a filter over the adjacency lists of the graph, keeping an edge with probability p . This takes $O(m)$ work, $O(\log m)$ span, and $O(m)$ space.

The **colorful sparsification** method sparsifies the graph by assigning a random color in $[1, \dots, \lceil 1/p \rceil]$ to each vertex and keeping an edge if the colors of its two endpoints match. Sanei-Mehri *et al.* [277] show that each butterfly is kept with probability p^3 , and so the butterfly count on the sparsified graph is divided by p^3 to obtain an unbiased estimate. Our parallel algorithm uses a hash function to map each vertex to a color, and then applies a filter over the adjacency lists of the graph, keeping an edge if its two endpoints have the same color. This takes $O(m)$ work, $O(\log m)$ span, and $O(m)$ space.

The variance bounds of our estimates are the same as shown by Sanei-Mehri *et al.* [277], and we refer the reader to their paper for details. The expected number of edges in both methods is pm , and by plugging this into the bounds for exact butterfly counting, and including the cost of sparsification, we obtain the following theorem.

Theorem 3.3. *Approximate butterfly counting with sampling rate p can be performed in $O((1 + \alpha' p)m)$ expected work, $O(\log m)$ span w.h.p., and $O(\min(n^2, (1 + \alpha' p)m))$ space, where α' is the arboricity of the sparsified graph.*

3.4.4 Approximate Degree Ordering

We show now that using approximate degree ordering in our preprocessing step also gives work-efficient bounds for butterfly counting. The proof for this closely follows Chiba and Nishizeki's [72] proof for degree ordering; notably, Chiba and Nishizeki prove that $O(\sum_{(u,v) \in E} \min(\deg(u), \deg(v))) = O(\alpha m)$.

Theorem 3.4. *Butterfly counting per vertex and per edge using approximate degree ordering is work-efficient.*

Proof. The total work of our counting algorithms, as discussed in Sections 3.4.2.2 and 3.4.2.3, is given precisely by the number of wedges that we must process, or for a preprocessed graph $G' = (X, E')$, $O(\sum_{x \in X} \sum_{y \in N_x(x)} \deg_x(y))$.

We must have that $\deg_x(y) \leq \deg(y) \leq 2 \cdot \deg(x)$; otherwise, y would appear before x in approximate degree order. Moreover, in our double summation, each edge appears precisely once by virtue of our ordering; thus, our bound becomes $O(\sum_{(x,y) \in E} (2 \cdot \min(\deg(x), \deg(y)))) = O(\alpha m)$, as desired. \square

3.4.5 Complement Degeneracy Ordering

We also show that using complement degeneracy ordering and approximate complement degeneracy ordering in our preprocessing step similarly gives work-efficient bounds for butterfly counting. As before, the proof for this closely follows from Chiba and Nishizeki's [72] proof for degree ordering.

Theorem 3.5. *Butterfly counting per vertex and per edge using complement degeneracy ordering is work-efficient.*

Proof. The total work of our counting algorithms, as discussed in Sections 3.4.2.2 and 3.4.2.3, is given precisely by the number of wedges that we must process, or for a preprocessed graph $G' = (X, E')$, $O(\sum_{x \in X} \sum_{y \in N_x(x)} \deg_x(y))$.

It is clear that $\deg_x(y) \leq \deg(y)$ by construction. We would like to show that $\deg_x(y) \leq \deg(x)$ as well.

Consider the sequential complement k -core algorithm. For every round r , let $\deg^r(x)$ denote the degree of x considering only the induced subgraph on unpeeled vertices. When we peel a vertex x in round r , we have for all neighbors y of x , $\deg^r(y) \leq \deg^r(y)$. By our ordering construction, we have $\deg_x(y) = \deg^r(y)$, and trivially, $\deg^r(x) \leq \deg(x)$. Thus, $\deg_x(y) \leq \deg(x)$, as desired.

Thus, the number of wedges that we must process is bounded by $O(\sum_{x \in X} \sum_{y \in N_x(x)} \min(\deg(x), \deg(y)))$. Each edge appears precisely once in this double summation by virtue of our ordering. Therefore, our bound becomes $O(\sum_{(x,y) \in E} \min(\deg(x), \deg(y))) = O(\alpha m)$, as desired. \square

Theorem 3.6. *Butterfly counting per vertex and per edge using approximate complement degeneracy ordering is work-efficient.*

Proof. This follows from the proof of Theorem 3.5, except that in each round r , when we peel a vertex u , we have for all neighbors y of x , $\deg^r(y) \leq 2 \cdot \deg^r(x)$. Thus, $\deg_x(y) \leq 2 \cdot \deg(x)$, and so the number of wedges that we must process is bounded by $O(\sum_{(x,y) \in E} (2 \cdot \min(\deg(x), \deg(y)))) = O(\alpha m)$. \square

3.5 Experiments

3.5.1 Environment

We run our experiments on an m5d.24xlarge AWS EC2 instance, which consists of 48 cores (with two-way hyper-threading), with 3.1 GHz Intel Xeon Platinum 8175 processors and 384 GiB of main memory. We use Cilk Plus’s work-stealing scheduler [53, 206] and we compile our programs with g++ (version 7.3.1) using the -O3 flag. We test our algorithms on a variety of real-world bipartite graphs from the Koblenz Network Collection (KONECT) [199]. We remove self-loops and duplicate edges from the graph. Table 3.1 describes the properties of these graphs, including sizes, number of butterflies, and peeling complexities.

We compare our algorithms against Sanei-Mehri *et al.*’s [277] work, which is the state-of-the-art sequential butterfly counting implementation.

Notationally, when discussing wedge and butterfly aggregation methods, we use the prefix “A” to refer to using atomic adds for butterfly aggregation, and we take a lack of prefix to mean that the wedge aggregation method was used for butterfly aggregation. “BatchS” is the simple version of batching and “BatchWA” is the wedge-aware version of batching that dynamically assigns tasks to workers so they have a roughly equal number of wedges to process.

3.5.2 Results

3.5.2.1 Butterfly Counting

Figures 3.4, 3.5, and 3.6 show the runtimes over different aggregation methods for counting per vertex, per edge, and in total, respectively, for the seven datasets in Table 3.1 with sequential counting times exceeding 1 second. The times are normalized to the fastest combination of aggregation and ranking methods for each dataset. We find that simple batching and wedge-aware batching give the best runtimes for butterfly counting in general. Among the work-efficient aggregation methods, hashing and histogramming with atomic adds are often faster than sorting, particularly for larger graphs due to increased parallelism and locality, respectively. Our fastest parallel runtimes for each dataset for total, per-vertex, and per-edge counts are shown in Table 3.2.

We also implemented sequential algorithms for butterfly counting in PARBUTTERFLY that do not incur any of the parallelism overheads. Table 3.2 includes the runtimes for our sequential counting implementations, as well as runtimes for implementations from previous works, all of which we tested on the same machine. The

| Dataset | Abbreviation | $ U $ | $ V $ | $ E $ | # butterflies |
|----------------------|---------------|------------|------------|-------------|--------------------|
| DBLP | dblp | 4,000,150 | 1,425,813 | 8,649,016 | 21,040,464 |
| Github | github | 120,867 | 56,519 | 440,237 | 50,894,505 |
| Wikimedia edits (it) | itwiki | 2,225,180 | 137,693 | 12,644,802 | 298,492,670,057 |
| Discogs label-style | discogs | 270,771 | 1,754,823 | 5,302,276 | 3,261,758,502 |
| Discogs artist-style | discogs_style | 383 | 1,617,943 | 5,740,842 | 77,383,418,076 |
| LiveJournal | livejournal | 7,489,073 | 3,201,203 | 112,307,385 | 3,297,158,439,527 |
| Wikimedia edits (en) | enwiki | 21,416,395 | 3,819,691 | 122,075,170 | 2,036,443,879,822 |
| Delicious user-item | delicious | 33,778,221 | 833,081 | 101,798,957 | 56,892,252,403 |
| Orkut | orkut | 8,730,857 | 2,783,196 | 327,037,487 | 22,131,701,213,295 |
| Web trackers | web | 27,665,730 | 12,756,244 | 140,613,762 | 20,067,567,209,850 |

Table 3.1: These are relevant statistics for the KONECT [199] graphs that we experimented on.

| Dataset | Total Counts | | | | | | Per-Vertex Counts | | | | | | Per-Edge Counts | | | | | |
|-------------|--------------------------|---------|----------------|----------------|------------------|----------------|--------------------------|-----------|----------------|--------------------------|-----------|----------------|--------------------------|----|----|--------------------------|--------------------------|--|
| | Sanei-Mehri et al. [277] | | | ESCAPE [258] | | | Sariyüce and Pinar [282] | | | Sariyüce and Pinar [282] | | | Sariyüce and Pinar [282] | | | Sariyüce and Pinar [282] | | |
| | PB | PB | T ₁ | T ₁ | T _{48h} | T ₁ | PB | PB | T ₁ | PB | PB | T ₁ | T _{48h} | PB | PB | T ₁ | Sariyüce and Pinar [282] | |
| itwiki | 0.10* | 1.38* | 1.63 | 1798.43 | 4.97 | 0.13* | 1.43* | 6.06 | 0.37* | 3.24° | 19314.87 | | | | | | | |
| discogs | 0.90#◊ | 1.36° | 4.12 | 234.48 | 2.08 | 0.93#◊ | 1.53° | 96.09 | 0.59° | 5.01* | 1089.04 | | | | | | | |
| livejournal | 3.83* | 35.41* | 37.80 | > 5.5 hrs | 139.06 | 5.65* | 36.22* | 158.79 | 10.26° | 105.65* | > 5.5 hrs | | | | | | | |
| enwiki | 8.29° | 68.73* | 69.10 | > 5.5 hrs | 151.63 | 11.75° | 75.10* | 608.53 | 16.73° | 167.69* | > 5.5 hrs | | | | | | | |
| delicious | 13.52° | 165.03* | 162.00 | > 5.5 hrs | 286.86 | 18.36° | 182.00* | 1027.12 | 23.58° | 321.02° | > 5.5 hrs | | | | | | | |
| orkut | 35.07* | 423.02* | 403.46 | > 5.5 hrs | 1321.20 | 66.19* | 439.02* | 2841.27 | 131.07*◊ | 1256.83* | > 5.5 hrs | | | | | | | |
| web | 12.18° | 115.53° | 4340 | > 5.5 hrs | 172.77 | 15.89° | 195.43° | > 5.5 hrs | 17.40# | 218.15° | > 5.5 hrs | | | | | | | |

Table 3.2: These are best runtimes in seconds for parallel and sequential butterfly counting from PARBUTTERFLY (PB), as well as runtimes from previous work. Note that PGD [7] is parallel, while the rest of the implementations are serial. Also, for the runtimes from our framework, we have noted the ranking used; * refers to side ranking, # refers to approximate complement degeneracy ranking, and ° refers to approximate degree ranking. The wedge aggregation method used for the parallel runtimes was simple batching, except the cases labeled with ◊, which used wedge-aware batching.

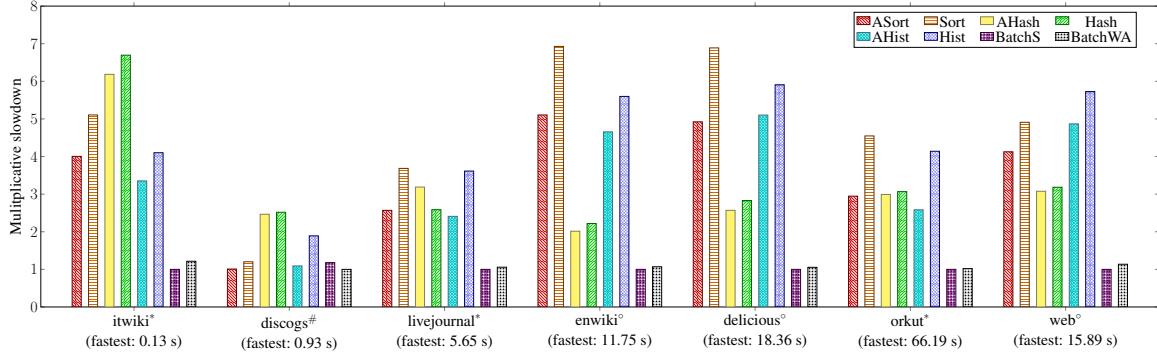


Figure 3.4: These are the parallel runtimes for butterfly counting per vertex, considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking, and ° refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses.

code from Sanei-Mehri *et al.* and Sariyüce and Pinar [282] are serial implementations for global and local butterfly counting, respectively. PGD [7] is a parallel framework for counting subgraphs of up to size 4 and ESCAPE is a serial framework for counting subgraphs of up to size 5. We timed only the portion of the codes that counted butterflies. Our configurations achieve parallel speedups between 6.3–13.6x over the best sequential implementations for large enough graphs.³ We also improve upon PGD by 349.6–5169x due to having a work-efficient algorithm.

Figures 3.7 and 3.8 show our self-relative speedups on livejournal for per-vertex and per-edge counting, respectively. Across all rankings, on livejournal, we achieve self-relative speedups between 10.4–30.9x for per-vertex counting, between 9.2–38.5x for per-edge counting, and between 7.1–38.4x for in total counting.

3.5.2.2 Ranking

Figure 3.9 shows the runtimes for butterfly counting per vertex for different rankings using the simple batching method. The times are normalized to the time for the fastest ranking for each dataset. Side ordering outperforms the other rankings for itwiki, livejournal, and orkut, while approximate complement degeneracy, approximate degree, and degree orderings outperform side ordering for discogs, enwiki, delicious, and web.

Note that different rankings change the number of wedges that we must process; in particular, we found that complement degeneracy and approximate complement degeneracy minimizes the number of wedges that we process across all of the real-world graphs considered. However, complement degeneracy is not a feasible ordering in practice, since the time for ranking often exceeds the time for the actual counting. Moreover, side ordering often outperforms the other rankings due to better locality,

³By “large enough,” we mean graphs for which the sequential counting algorithms take more than 2 seconds to complete.

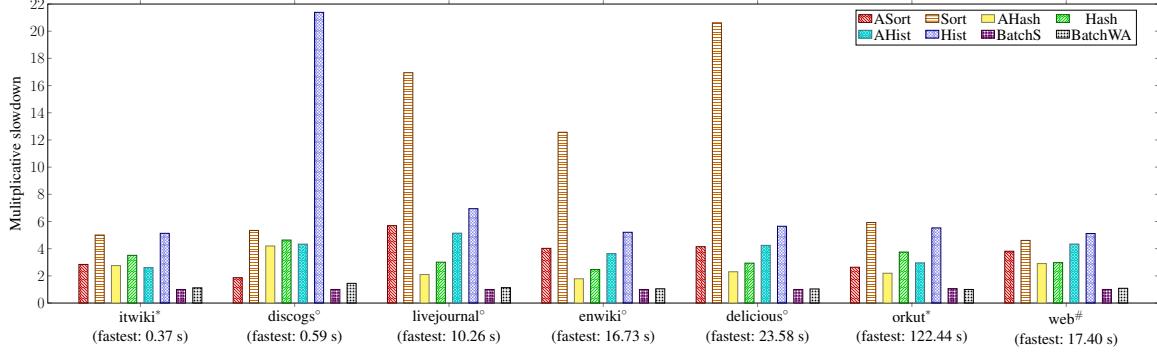


Figure 3.5: These are the parallel runtimes for butterfly counting per edge, considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking, and ° refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses.

especially if the number of wedges processed by the other rankings does not greatly exceed the number of wedges given by side ordering. We found that the approximate complement degeneracy, degree, and approximate degree orders perform similarly, and these orderings are all efficient to compute.

As such, we devise a metric f that in general determines whether side ordering outperforms other rankings. If we let w_s be the number of wedges processed by using side ordering and w_r be the number of wedges processed by using another ranking, our metric is $(w_s - w_r)/w_s$. If this metric is below 0.1, then side ordering will outperform or perform just as well as other rankings. Table 3.3 shows this metric across all of the rankings in PARBUTTERFLY. Note that this metric is fairly similar across these other rankings, and is particularly high for web, explaining the significant speedup obtained by using approximate degree ordering over side ordering for web. The f metric can be computed at runtime, before actually ranking the graph, to decide which ordering to use.

3.5.2.3 Approximate Counting

Figure 3.10 shows runtimes for both colorful sparsification and edge sparsification on orkut, as well as the corresponding single-threaded times. We see that over a variety of probabilities p we achieve self-relative speedups between 4.9–21.4x.

3.5.3 Cache Optimization for Butterfly Counting

We find that using Wang *et al.*'s [328] cache optimization for total, per-vertex, and per-edge parallel butterfly counting gives speedups of up to 1.7x of our parallel butterfly counting algorithms without the cache optimization, considering the best aggregation and ranking methods for each case. We present the butterfly counting experiments with the cache optimization enabled in this section. The trends are similar to

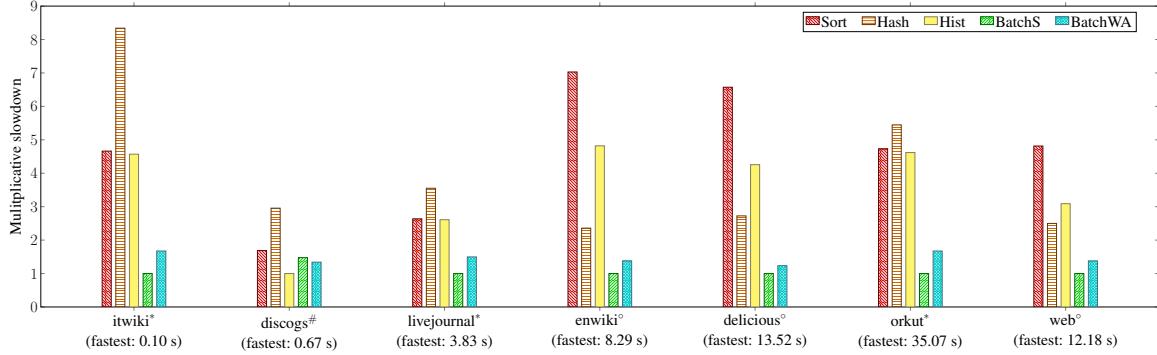


Figure 3.6: These are the parallel runtimes for butterfly counting in total, considering different wedge aggregation methods (butterfly aggregation does not apply). We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking, and ° refers to approximate degree ranking. All times are scaled by the fastest parallel time, as parentheses.

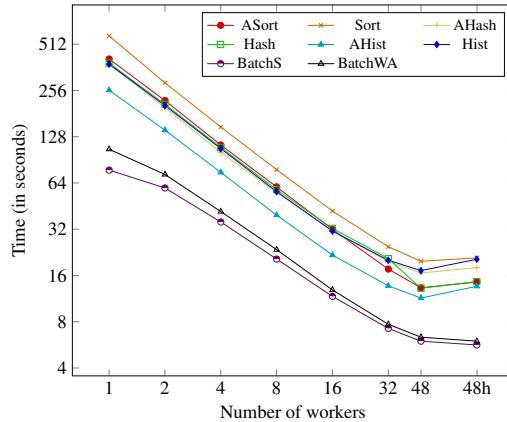


Figure 3.7: These are the runtimes for butterfly counting per vertex on livejournal using side ranking, over different numbers of threads. The self-relative speedups are between 13.7–28.3x.

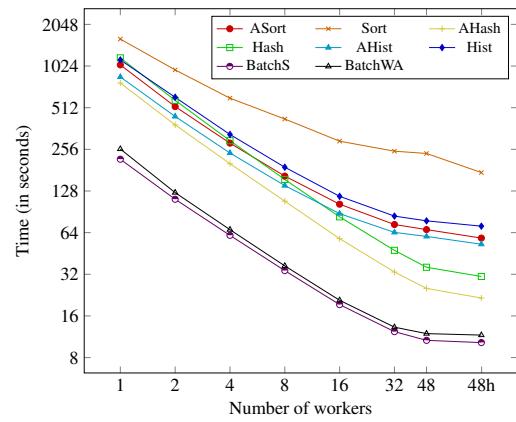


Figure 3.8: These are the runtimes for butterfly counting per edge on livejournal using approximate degree ranking, over different numbers of threads. The self-relative speedups are between 15.9–38.0x.

the results without the cache optimization enabled. Note that the cache optimization does not always improve the performance of butterfly counting; for certain graphs, the best butterfly counting time is obtained without using the optimization.

3.5.4 Butterfly Counting

Figures 3.11, 3.12, and 3.13 show the runtimes over different aggregation methods for counting per vertex, per edge, and in total, respectively, for the seven datasets

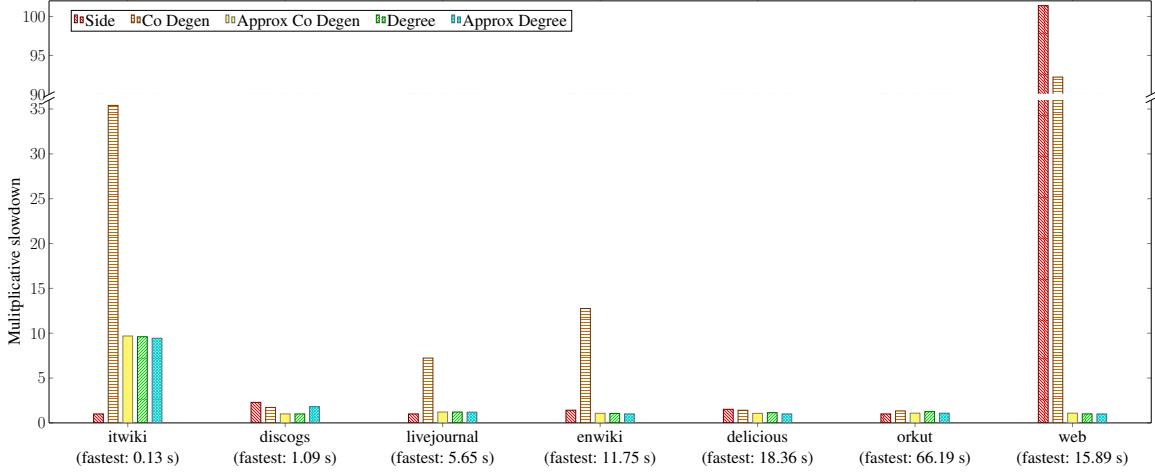


Figure 3.9: These are the runtimes for butterfly counting per vertex, considering different rankings. We use simple batching as our wedge aggregation method. All times are scaled by the fastest runtime, as indicated in parentheses. Moreover, the time taken to rank each graph is included in the runtimes.

| Dataset | Approx | | Approx | |
|-------------|-----------------------|------------|-----------------------|------------|
| | Complement Degeneracy | Degeneracy | Complement Degeneracy | Degeneracy |
| itwikipedia | 0.021 | | 0.020 | 0 |
| discogs | 0.97 | | 0.97 | 0.97 |
| livejournal | 0.035 | | 0.033 | 0.011 |
| enwiki | 0.47 | | 0.47 | 0.45 |
| delicious | 0.60 | | 0.60 | 0.59 |
| orkut | 0.070 | | 0.064 | 0.042 |
| web | 0.95 | | 0.95 | 0.95 |

Table 3.3: These are the fractional values $f = (w_s - w_r)/w_s$, where w_s is the number of wedges that must be processed using side ordering and w_r is the number of wedges that must be processed using the labeled ordering. Note that for itwikipedia, the number of wedges produced by degree ordering is precisely equal to the number of wedges produced by side ordering.

in Figure 3.1 with sequential counting times exceeding 1 second. The times are normalized to the fastest combination of aggregation and ranking methods for each dataset. Considering different wedge and butterfly aggregation methods, we again find that simple batching and wedge-aware batching give the best runtimes for butterfly counting in general. Of the work-efficient aggregation methods, hashing with atomic adds is often the fastest, particularly for larger graphs due to increased parallelism.

Also, Figure 3.4 shows the runtimes for the sequential counting implementations using the cache optimization, as well as runtimes for implementations from previous works, all of which were tested on the same machine. Our configurations achieve parallel speedups of between 5.7–18.0x over the best sequential implementations for

| Dataset | Total Counts | | | | Per-Vertex Counts | | | | Per-Edge Counts | | | |
|-------------|--------------------------|-------------|-------------|-----------------|--------------------------|-----------------|--------------------------|-------------|-----------------|----------|-----------|---|
| | Sanei-Mehri et al. [277] | | PGD [7] | | SARIYİÇİ and PINAR [282] | | SARIYİÇİ and PINAR [282] | | | | | |
| | PB T_{48h} | PB T_1 | PB T_1 | PB T_{48h} | PB T_1 | PB T_{48h} | PB T_1 | PB T_1 | | | | |
| itwwiki | 0.10* | 1.22° | 1.63 | 1798.43 | 4.97 | 0.14* | 1.46* | 6.06 | 0.41*□ | 4.77* | 19314.87 | |
| discogs | 0.81#♦ | 1.20◊ | 4.12 | 234.48 | 2.08 | 0.83°* | 0.90# | 96.09 | 0.52° | 2.97° | 1089.04 | |
| livejournal | 3.83* | 35.12* | 37.80 | > 5.5 hrs | 139.06 | 5.58* | 38.91* | 158.79 | 8.91◊ | 106.07* | > 5.5 hrs | |
| enwiki | 5.59° | 59.18° | 69.10 | > 5.5 hrs | 151.63 | 8.31° | 54.71° | 608.53 | 11.58° | 152.92° | > 5.5 hrs | |
| delicious | 8.05° | 82.43° | 162.00 | > 5.5 hrs | 286.86 | 11.93° | 96.18° | 1027.12 | 15.98° | 210.70° | > 5.5 hrs | |
| orkut | 22.47# | 414.02* | 403.46 | > 5.5 hrs | 1321.20 | 41.57° | 557.10* | 2841.27 | 89.52° | 1141.05* | > 5.5 hrs | ∞ |
| web | 8.08° | 61.68° | 4340 | > 5.5 hrs | 172.77 | 13.63◊ | 77.82° | > 5.5 hrs | 14.08◊ | 154.35° | > 5.5 hrs | |

Table 3.4: These are runtimes in seconds for sequential butterfly counting from PARBUTTERFLY (using the cache optimization), as well as runtimes from previous work. Note that PGD [7] is parallel, while the rest of the implementations are serial. Also, for the runtimes from our framework, we have noted the ranking used; * refers to side ranking, # refers to approximate complement degeneracy ranking, ◊ refers to degree ranking, and ° refers to approximate degree ranking. We have also noted the wedge aggregation and butterfly aggregation methods used for the parallel runtimes; * refers to hashing with atomic adds, ♦ refers to histogramming for both aggregation methods, and □ refers to wedge-aware batching. The rest of the parallel runtimes were obtained using simple batching.

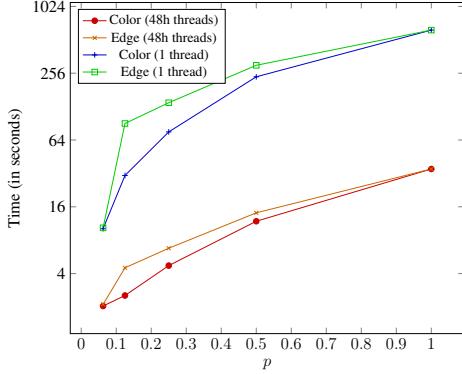


Figure 3.10: These are the runtimes for colorful sparsification and edge sparsification over different probabilities p . We considered both the runtimes on 48 cores hyper-threaded and on a single thread. We ran these algorithms on orkut, using simple batch aggregation and side ranking.

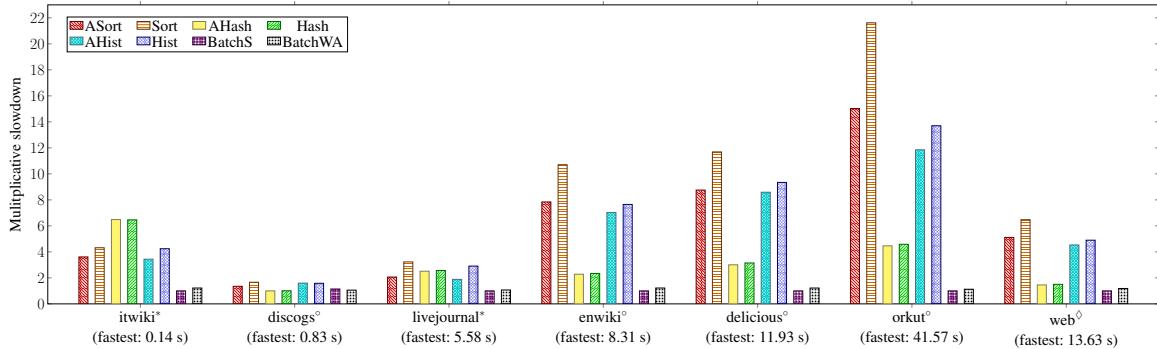


Figure 3.11: These are the parallel runtimes for butterfly counting per vertex (using the cache optimization), considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, \diamond refers to degree ranking, and \circ refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses.

large enough graphs.⁴ We improve upon PGD [7] by 289.5–5170x.

Figures 3.14 and 3.15 show our self-relative speedups on livejournal for per-vertex and per-edge counting, respectively, using the cache optimization. Across all rankings, on livejournal, we achieve self-relative speedups between 14.1–37.4x for per-vertex counting and between 9.8–38.9x for per-edge counting.

3.5.4.1 Ranking

Figure 3.16 shows the runtimes for butterfly counting per vertex for different rankings using the simple batching method with the cache optimization. The times are

⁴By “large enough,” we mean graphs for which the sequential counting algorithms take more than 2 seconds to complete.

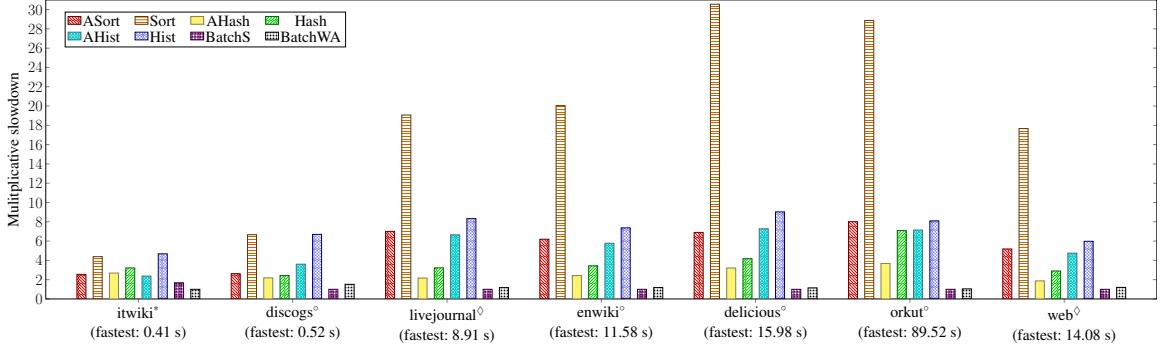


Figure 3.12: These are the parallel runtimes for butterfly counting per edge (using the cache optimization), considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, \diamond refers to degree ranking, and \circ refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses.

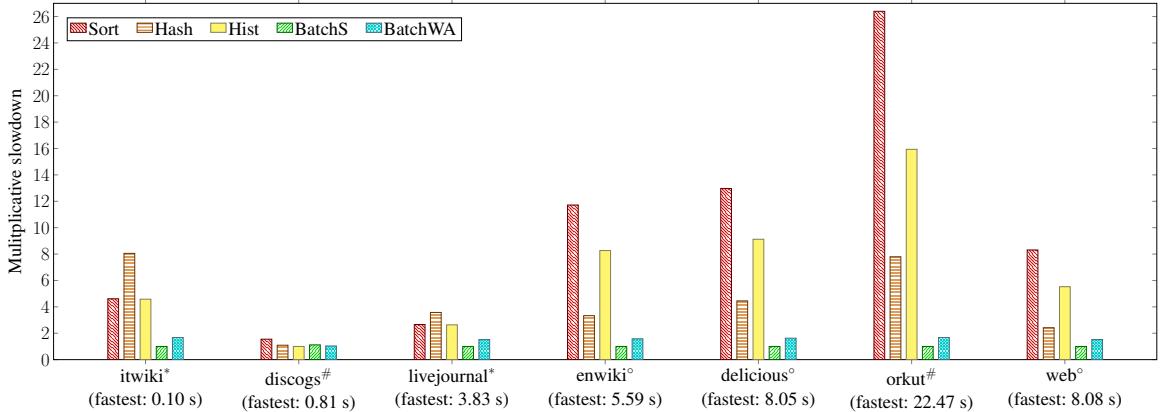


Figure 3.13: These are the parallel runtimes for butterfly counting in total (using the cache optimization), considering different wedge aggregation methods (butterfly aggregation does not apply). We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking, and \diamond refers to approximate degree ranking. All times are scaled by the fastest parallel time, as parentheses.

normalized to the time for the fastest ranking for each dataset. Side ordering outperforms the other rankings for itwiki and livejournal, while approximate complement degeneracy, approximate degree, and degree orderings outperform side ordering for discogs, enwiki, delicious, orkut, and web.

3.5.4.2 Approximate Counting

Figure 3.10 shows runtimes with the cache optimization for both colorful sparsification and edge sparsification on orkut, as well as the corresponding single-threaded times.

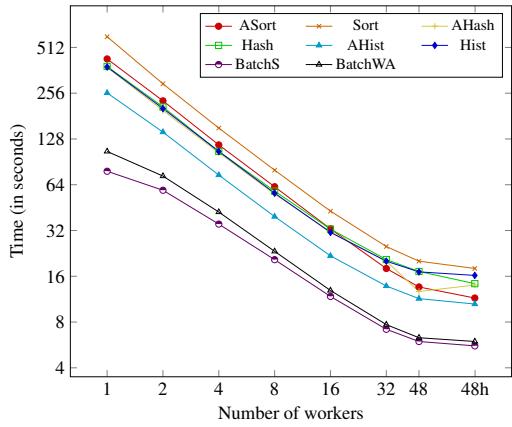


Figure 3.14: These are the runtimes for butterfly counting per vertex on livejournal using side ranking and using the cache optimization, over different numbers of threads. The self-relative speedups are between 14.1–37.4x.

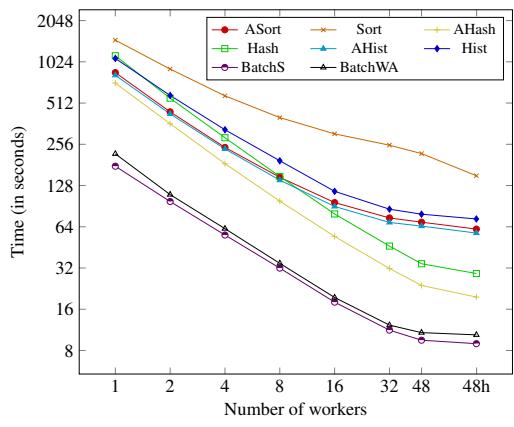


Figure 3.15: These are the runtimes for butterfly counting per edge on livejournal using approximate degree ranking and using the cache optimization, over different numbers of threads. The self-relative speedups are between 9.8–38.9x.

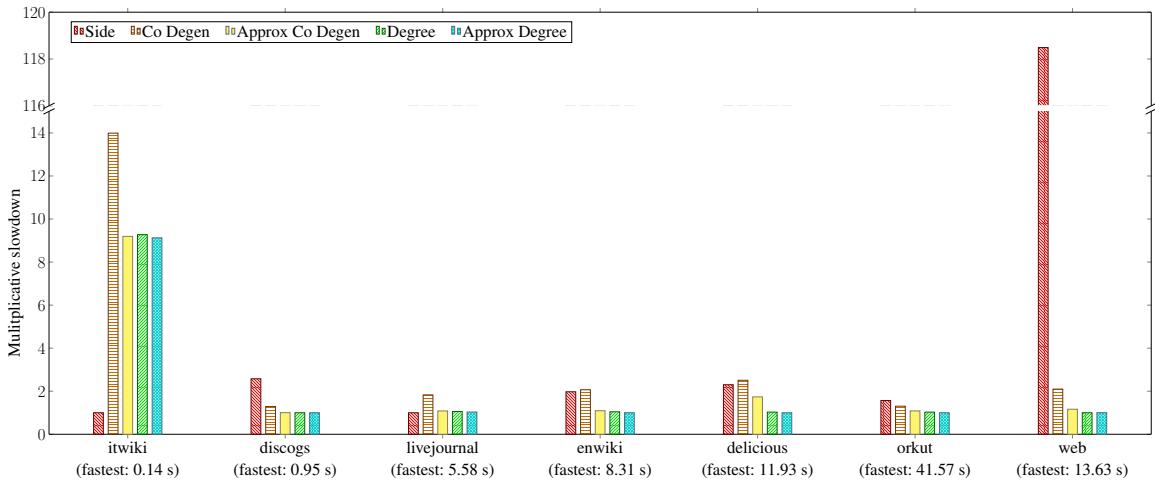


Figure 3.16: These are the parallel runtimes for butterfly counting per vertex (using the cache optimization), considering different rankings. We use simple batching as our wedge aggregation method. All times are scaled by the fastest parallel time, as indicated in parentheses. Moreover, the time taken to rank each graph is included in the runtimes.

We see that over a variety of probabilities p we achieve self-relative speedups between 5.4–18.9x.

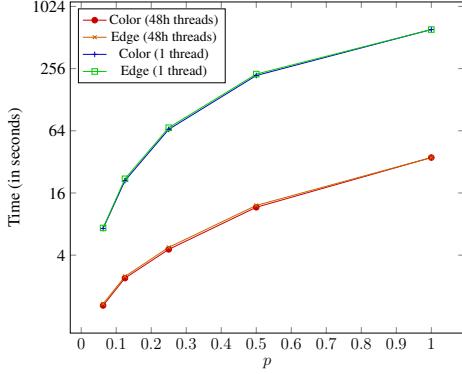


Figure 3.17: These are the runtimes for colorful sparsification and edge sparsification over different probabilities p (using the cache optimization). We considered both the runtimes on 48 cores hyperthreaded and on a single thread. We ran these algorithms on orkut, using simple batch aggregation and side ranking.

3.6 Related Work

There have been several sequential algorithms designed for butterfly counting. Wang *et al.* [325] propose the first algorithm for butterfly counting over vertices in $O(\sum_{v \in V} \deg(v)^2)$ work, and Sanei-Mehri *et al.* [277] introduce a practical speedup by choosing the vertex partition with fewer wedges to iterate over. Sanei-Mehri *et al.* [277] also introduce approximate counting algorithms based on sampling and graph sparsification. Later, Zhu *et al.* [354] present a sequential algorithm for counting over vertices based on ordering vertices (although they do not specify which order) in $O(\sum_{v \in V} \deg(v)^2)$ work. They extend their algorithm to the external-memory setting and also design sampling algorithms. Chiba and Nishizeki's [72] original work on counting 4-cycles in general graphs applies directly to butterfly counting in bipartite graphs and has a better work complexity. Chiba and Nishizeki [72] use a ranking algorithm that counts the total number of 4-cycles in a graph in $O(\alpha m)$ work, where α is the arboricity of the graph. While they only give a total count in their work, their algorithm can easily be extended to obtain counts per-vertex and per-edge in the same time complexity. Butterfly counting using degree ordering was also described by Xia [339]. Sariyüce and Pinar [282] introduce algorithms for butterfly counting over edges, which similarly takes $O(\sum_{v \in V} \deg(v)^2)$ work.

In terms of prior work on parallelizing these algorithms, Wang *et al.* [325] implement a distributed algorithm using MPI that partitions the vertices across processors, and each processor sequentially counts the number of butterflies for vertices in its partition. They also implement a MapReduce algorithm, but show that it is less efficient than their MPI-based algorithm. The largest graph they report parallel times for is the *deli* graph with 140 million edges and 1.8×10^{10} butterflies; this is the delicious tag-item graph in KONECT [199]. On this graph, they take 110 seconds on 16 nodes, whereas on the same graph we take 5.17 seconds on 16 cores.

Very recently, and independently of our work, Wang *et al.* [328] describe an algorithm for butterfly counting using degree ordering, as done in Chiba and Nishizeki [72],

and also propose a cache optimization for wedge retrieval. Their parallel algorithm is our parallel algorithm with simple batching for wedge aggregation, except they manually schedule the threads, while we use the Cilk scheduler. They use their algorithm to speed up approximate butterfly counting, and also propose an external-memory variant.

There has been recent work on algorithms for finding subgraphs of size 4 or 5 [165, 119, 258, 7, 91], which can be used for butterfly counting as a special case. Marcus and Shavitt [228] design a sequential algorithm for finding subgraphs of up to size 4. Hocevar and Demsar [165] present a sequential algorithm for counting subgraphs of up to size 5. Pinar *et al.* [258] also present an algorithm for counting subgraphs of up to size 5 based on degree ordering as done in Chiba and Nishizeki [72]. Elenberg *et al.* [119] present a distributed algorithm for counting subgraphs of size 4. Ahmed *et al.* [7] present the PGD shared-memory framework for counting subgraphs of up to size 4. The work of their algorithm for counting 4-cycles is $O(\sum_{(u,v) \in E} (\deg(v) + \sum_{u' \in N(v)} \deg(u')))$, which is higher than that of our algorithms. Aberger et al. [4] design the EmptyHeaded framework for parallel subgraph finding based on worst-case optimal join algorithms [247]. For butterfly counting, their approach takes quadratic work. We were unable to obtain runtimes for EmptyHeaded because it ran out of memory in our environment. Dave *et al.* [91] present a parallel method for counting subgraphs of up to size 5 local to each edge. For counting 4-cycles, their algorithm is the same as PGD, which we compare with. There have also been various methods for approximating subgraph counts via sampling [182, 8, 9, 333, 175, 59, 271, 230]. Finally, there has also been significant work for the past decade on parallel triangle counting algorithms (e.g., [304, 23, 312, 19, 253, 254, 80, 251, 313, 320, 225, 193, 152, 151, 168, 98, 159, 348] and papers from the annual GraphChallenge [276], among many others).

3.7 Discussion

We have designed in this chapter a framework PARBUTTERFLY that provides efficient parallel algorithms for butterfly counting (global, per-vertex, and per-edge). We have also shown strong theoretical bounds in terms of work and span for these algorithms. The PARBUTTERFLY framework is built with modular components that can be combined for practical efficiency. PARBUTTERFLY outperforms the best existing parallel butterfly counting implementations, and we outperform the fastest sequential baseline by up to 13.6x for butterfly counting.

Chapter 4

Five-cycle Counting

4.1 Introduction

While there has been significant recent work on counting subgraphs of size three or four [118, 119, 7], counting subgraphs of size five or more is a difficult task even on the most modern hardware due to the massive number of such subgraphs in large graphs. As the subgraph sizes grow, the number of possible subgraphs grows exponentially. We consider specifically the efficient counting of five-cycles. This pattern is particularly important for fraud detection [260]. Compared to other connected five-vertex patterns, five-cycles are much more difficult to count because they are the only such pattern that requires first counting all directed three-paths. Notably, the Efficient Subgraph Counting Algorithmic PackagE (ESCAPE), a software package by Pinar et al. that serially counts all five-vertex subgraphs in large graphs [258], spends between 25–58% of the total runtime on counting five-cycles alone based on our measurement.

While there has been prior work on developing and implementing serial five-cycle counting algorithms [195, 258, 165], there has been no prior work on designing and implementing theoretically-efficient and scalable parallel five-cycle counting algorithms. We focus on designing multicore solutions, as nearly all publicly-available graphs (which have up to hundreds of billions of edges [233]) can fit on a commodity multicore machine [97, 106].

We present two new parallel five-cycle counting algorithms that not only have strong theoretical guarantees, but are also demonstrably fast in practice. These algorithms are based on two different serial algorithms, namely by Kowalik [195] and from ESCAPE by Pinar et al. [258]. Kowalik studied k -cycle counting in graphs for $k \leq 6$ and proposed a five-cycle counting algorithm that runs in $O(md^2) = O(m\alpha^2)$ time for d -degenerate graphs [195], where m is the number of edges in the graph and α is the arboricity of the graph.¹ The ESCAPE implementation contains a five-cycle counting algorithm that, with an important modification that we make, achieves the same asymptotic complexity of $O(m\alpha^2)$ [258]. The arboricity of a graph is a measure of its sparsity, and having running times parameterized by α is desirable since most

¹A graph is d -degenerate if every subgraph has a vertex of degree at most d , and a graph has arboricity α if the minimum number of spanning forests needed to cover all of the edges of the graph is α .

real-world graphs have low arboricity [97].

The main procedure in both algorithms and the essential modification to the ESCAPE algorithm is to first compute an appropriate arboricity orientation of the graph in parallel, where the vertices' out-degrees are upper-bounded by $O(\alpha)$. This orientation then enables the efficient counting of directed two-paths and three-paths, which are then appropriately aggregated to form five-cycles. Notably, the counting and aggregation steps can each be efficiently parallelized. The two algorithms differ fundamentally in the ways in which they use the orientations of these path substructures to eliminate double-counting. We prove theoretical bounds that show that both of our algorithms match the work of the best sequential algorithms, taking $O(m\alpha^2)$ work and $O(\log^2 n)$ span with high probability (w.h.p.). Additionally, we present two approximate five-cycle counting algorithms based on counting five-cycles in a sparsified graph, and we prove that both approximation algorithms give unbiased estimates on the global five-cycle count. We show that both algorithms take $O(pma^2 + m)$ expected work and $O(\log^2 n)$ span w.h.p. for a sampling probability p .

We present optimized implementations of our algorithms, which use thread-local data structures, fast resetting of arrays, and a new work scheduling strategy to improve load balancing. We provide a comprehensive experimental evaluation of our five-cycle counting algorithms. On a 36-core machine with 2-way hyperthreading, our best exact parallel algorithm achieves between 10–46x self-relative speedup, and between 162–818x speedups over the fastest prior serial five-cycle counting implementation, which is from ESCAPE [258]. We also implement our own serial versions of the two exact algorithms, which are 7–38.91x faster than ESCAPE's algorithm due to improved theoretical work complexities. Our best parallel algorithm achieves between 10–32x speedups over our best serial algorithm. Moreover, we show the tradeoffs between error and running time of our approximate five-cycle counting algorithms. In particular, we are able to approximate five-cycle counts with 11.77% error with a 9.10–188.94x speedup over exact five-cycle counting on the same graphs. Our parallel five-cycle counting code is available at <https://github.com/ParAlg/gbbs/tree/master/benchmarks/CycleCounting>.

4.2 Background and Related Work

The difficulty of cycle counting has attracted considerable research effort over the years. Counting the number of k -cycles with k as an input parameter is NP-complete since it includes the problem of finding a Hamiltonian cycle. However, efficient algorithms have been developed to count k -cycles for $k \leq 5$. Notably, Alon et al. [14] developed algorithms for efficiently finding a k -cycle for general k , but these translate to efficient k -cycle counting algorithms only for planar graphs where $k \leq 5$. For $k = 3, 4$, Chiba and Nishizeki [72] proposed algorithms that take $O(m\alpha)$ time. More recently, Bera et al. [34] analyzed the subgraph counting problem for $k = 5$ and gave an algorithm that takes $O(m\alpha^3)$ time, and the five-cycle counting part of the algorithm takes $O(m\alpha^3)$ time. However, it is shown in the same study that this result is unlikely to be extended to $k > 5$, due to the Triangle Detection Conjecture, which

puts a lower bound of $\Omega(m^{1+\gamma})$ time with $\gamma > 0$ on any triangle detection algorithm on an input graph with m edges [2]. If the conjecture holds, a reduction of the triangle detection problem to the six-cycle counting problem implies that there cannot be a $o(f(\alpha)m^{1+\gamma})$ time algorithm for six-cycle counting.

Until recently, because of the high computational power required, exact five-vertex subgraph counting was often deemed impractical on graphs with more than a few million edges. Most effort has focused on obtaining approximate counts or approximate graphlet frequency distributions [338, 44, 263]. Hocevar and Demsar [165] developed Orca to count subgraphs of up to size five and tested them on graphs with tens of thousands of vertices. Pinar et al. [258] developed ESCAPE, which is the first package that aims to perform exact counting of all five-vertex subgraphs on moderately large graphs. However, ESCAPE does not exploit parallelism and is not optimized for cycle-counting. Kowalik [195] gave a serial algorithm for five-cycle counting that takes $O(m\alpha^2)$, the best known theoretical bound for five-cycle counting, but does not provide an implementation. In Section 4.4, we describe Kowalik’s and Pinar et al.’s five-cycle counting algorithms in more detail.

While there has not been prior work on parallel five-cycle counting algorithms, parallel cycle counting algorithms for smaller cycles have been studied over the years. Specifically, for the case of three-cycles, or triangles, there has been a significant amount of attention over the past two decades (e.g., [289, 319, 251, 39], among many others).

Moreover, fast sequential algorithms for four-cycles have been studied extensively. For bipartite graphs, four-cycles, also known as butterflies, are the smallest non-trivial subgraphs. Chiba and Nishizeki’s work [72] described a four-cycle counting algorithm that takes $O(m\alpha)$ work by using a degree ordering of the graph. Subsequently, butterfly counting algorithms using degree ordering and other orderings have also been designed [339, 325, 277, 354, 282].

There have been fewer studies on parallel four-cycle counting algorithms. The Parametrized Graphlet Decomposition package by Ahmed et al. [7] provides efficient parallel implementations of exact counting of subgraphs of up to size four, including four-cycles. Wang et al. [325] implement a distributed algorithm using MPI that partitions the vertices across processors, where each processor sequentially counts the number of butterflies for vertices in its partition. Shi and Shun [298] presented a framework for parallel butterfly counting with several algorithms achieving $O(m\alpha)$ expected work and $O(\log m)$ span with high probability. Wang et al. [328] describe a similar parallel butterfly counting algorithm, with an additional cache optimization in their implementation.

4.3 Preliminaries

Graph Notation. When discussing directed graphs, $N^\rightarrow(v)$ denotes v ’s out-neighbors and $N^\leftarrow(v)$ denote the in-neighbors.

Furthermore, we use $N_v(u)$ ($N_v^\rightarrow(u)$ for directed graphs) to represent the neighbors of vertex u that are after v given a non-increasing degree ordering. When vertices are

relabelled by non-increasing degree order, we can easily obtain $N_v(u) = N(u) \cap \{w \in V \mid w > v\}$.

Estimators. We define a (ε, δ) -estimator of a number X , to be a random variable Y , where $\Pr[|Y - X| \geq \varepsilon X] \leq \delta$ for any $\varepsilon, \delta \in [0, 1]$.

4.4 Five-cycle Counting Algorithms

In this section, we present two new parallel algorithms for counting five-cycles. The first algorithm is based on the serial algorithm by Kowalik [195]. Kowalik shows that the algorithm achieves a time complexity of $O(m)$ on planar graphs, in which $\alpha = O(1)$, or $O(md^2) = O(m\alpha^2)$ on d -degenerate graphs. The second algorithm is based on the serial algorithm by Pinar et al. in their ESCAPE framework for counting all 5-vertex subgraphs in a graph [258]. We show that both of the parallel algorithms that we design are provably work-efficient with polylogarithmic span.

Additionally, we present two techniques that, combined with our parallel exact five-cycle counting algorithms, allow us to generate unbiased estimates of the global five-cycle count in parallel using graph sparsification.

4.4.1 Preprocessing: Graph Orientation

Similar to many previous subgraph counting algorithms [34, 296], a key step in our algorithms is a preprocessing step that orients the graph G , creating a directed acyclic graph G^\rightarrow where the out-degrees of vertices are upper-bounded. We use *l -orientation* to refer to an orientation where each vertex's out-degree is bounded by l . Furthermore, orientations in our context are always induced from a total ordering of the vertices, where directed edges point from vertices lower in the ordering to vertices higher in the ordering. As such, the problem of orienting the graph is reduced to the problem of finding an appropriate ordering of the vertices.

Degree Orientation. The core idea of orienting an undirected input graph based on ordering the vertices by non-increasing degree to perform subgraph counting or listing is attributed to Chiba and Nishizeki [72]. Using degree ordering, they proposed efficient triangle and four-cycle counting algorithms based on this key result:

Lemma 4.1 ([72]). *For a graph $G = (V, E)$, $\sum_{(u,v) \in E} \min\{\deg(u), \deg(v)\} \leq 2\alpha m$.*

This result allows us to bound the number of wedges in graph G by $2m\alpha$, where a wedge is defined as a triple (v, w, u) where $(v, w), (w, u) \in E$, $\deg(v) \geq \deg(w)$ and $\deg(v) \geq \deg(u)$. In Kowalik's five-cycle algorithm, wedges are the building blocks of five-cycles, and we can show that $O(\alpha)$ work is done for each wedge. Combined with the $O(m\alpha)$ bound on the number of wedges, this gives us the $O(m\alpha^2)$ running time bound.

Arboricity Orientation. An *arboricity orientation* of a graph is one where the vertices’ out-degrees are upper-bounded by $O(\alpha)$. An arboricity-oriented graph has slightly different theoretical properties compared to a degree-oriented graph, but literature has shown that in some algorithms arboricity orientation can achieve the same practical efficiency as degree orientation [296]. We note that in Kowalik’s five-cycle counting algorithm as well as our parallelization of the algorithm, both a degree ordering and an arboricity ordering are required to achieve work-efficiency.

One way to obtain an arboricity orientation is by computing the degeneracy ordering using a standard k -core decomposition algorithm [213, 229]. The algorithm repeatedly removes the vertex with the lowest degree from the graph. When we direct edges using this orientation, we obtain a DAG where each vertex’s out-degree is bounded by the degeneracy. While this algorithm can be parallelized to be work-efficient, it does not attain polylogarithmic span; notably, the problem is P-complete [17].

Since the parallel algorithm for exact degeneracy ordering has sub-optimal span, we use approximate algorithms with polylogarithmic span. We test two such algorithms: Goodrich-Pszona and Barenboim-Elkin. Both algorithms work by peeling low-degree vertices in batches. Goodrich and Pszona originally designed the algorithm in the external-memory model [150], while Barenboim and Elkin designed the algorithm for a distributed model [29]. Shi et al. adapted both algorithms for shared memory and showed that both compute an $O(\alpha)$ -orientation in $O(m)$ work and $O(\log^2 n)$ span (one of which is deterministic and the other of which is randomized) [296]. A different algorithm with the same (deterministic) work and span bounds was described by Besta et al. [40].

4.4.2 Kowalik’s Algorithm

We present in Algorithm 4.1 our parallelization of Kowalik’s serial five-cycle counting algorithm [195]. In this algorithm, vertices are sorted and processed in non-increasing degree order. Each vertex is processed by counting all five-cycles with the vertex itself as the lowest-ranked (i.e., highest-degree) vertex. After processing all vertices, each five-cycle is counted exactly once and the counts are summed.

Recall that we use $N_v(u)$ ($N_v^\rightarrow(u)$ for directed graphs) to represent the neighbors of vertex u that are after v in the non-increasing degree ordering. Since the vertices are relabeled by non-increasing degree order on line 3, we can easily obtain $N_v(u) = N(u) \cap \{w \in V \mid w > v\}$.

We now focus on the iteration v of the outer for-loop. An example is shown in Figure 4.1. We note that for each v , we consider only vertices ranked higher than v to complete five-cycles containing v . Lines 7–8 count in a parallel hash table U_v all wedges, where v is one of the endpoints and v is the lowest-ranked vertex in the wedge. Then, lines 11–12 store in a parallel hash table $T_{v,u}$ all wedges where v is one of the endpoints, v is the lowest-ranked vertex in the wedge, and u is the center. Both hash tables are indexed on w , the other endpoint of the wedge. We label each subfigure in Figure 4.1 with the corresponding parallel hash table U_v , where $v = 0$ for subfigures (a)–(e), and $v = 1$ for subfigure (f). However, note that we preemptively

Algorithm 4.1 – Kowalik’s Five-Cycle Counting Algorithm Parallelized

```

1: procedure COUNT-FIVE-CYCLES( $G = (V, E)$ )
2:    $\#_c \leftarrow 0$ 
3:   Relabel vertices of  $G$  such that  $\deg(0) \geq \deg(1) \geq \dots \geq \deg(n - 1)$ 
4:   Orient  $G$  using arboricity orientation to produce  $G^\rightarrow$ 
5:   parfor  $v \leftarrow 0$  to  $n - 1$  do
6:     Initialize an empty parallel hash table  $U_v$ 
7:     parfor  $u \in N_v(v)$  do
8:       parfor  $w \in N_v(u)$  do  $U_v[w] \leftarrow U_v[w] + 1$ 
9:     parfor  $u \in N_v(v)$  do
10:    Initialize an empty parallel hash table  $T_{v,u}$ 
11:    parfor  $w \in N_v(u)$  do
12:       $T_{v,u}[w] \leftarrow 1$ 
13:      parfor  $w \in N_v(u)$  do
14:        parfor  $x \in N_v^\rightarrow(w)$  do
15:          if  $x \neq u$  then
16:            if  $w \in N^\rightarrow(v)$  or  $v \in N^\rightarrow(w)$  then
17:               $\#_c \leftarrow \#_c + U_v[x] - T_{v,u}[x] - 1$ 
18:            else
19:               $\#_c \leftarrow \#_c + U_v[x] - T_{v,u}[x]$ 
20:   return  $\#_c$ 

```

subtract each entry in $T_{v,u}$ from U_v in our labels, for fixed u ; specifically, $u = 1$ for subfigures (a) and (b), $u = 2$ for subfigures (c) and (f), $u = 3$ for subfigure (d), and $u = 4$ for subfigure (e).

On each iteration of the loop in line 13, the algorithm counts all five-cycles that contain the wedge $v — u — w$. To accomplish this, the algorithm iterates through each neighbor x of w in G^\rightarrow , and considers the number of wedges that x shares with v , which is stored in $U_v[x]$. Note that three vertices of the cycle are given (v , u , and w), so the algorithm must ensure that the two vertices used to complete the cycle do not include these existing vertices. Each subfigure in Figure 4.1 considers a different $\{u, v, w, x\}$, where $\{u, v, w\}$ are given by the blue edges, and x is given by the red edge. Line 15 ensures that $x \neq u$ in the cycle; note that $x \neq w$ because the graph is assumed to not contain self-loops, and $x \neq v$ by definition of N_v^\rightarrow . Lines 16–19 check if v and w are neighbors; if so, then the number of wedges ending in x includes the wedge $v — w — x$, which does not properly complete a five-cycle. In this case, there is one fewer five-cycle completed by the wedges ending in x , and so we subtract one on line 17. In Figure 4.1, this case holds when there is a grey edge, which is precisely the edge connecting v and w . Finally, note that if there exists the wedge $v — u — x$, then this similarly does not properly complete a five-cycle, so we subtract $T_{v,u}[x]$, which stores precisely this wedge. We assume that indexing an entry that does not exist in a hash table returns a value of 0. In Figure 4.1, this is already noted in our preemptive subtraction of $T_{v,u}$ from U_v in our labels.

As every thread operates on the variable $\#_c$, we use atomic add for all of these

Example graph G G^\rightarrow under degree orientation

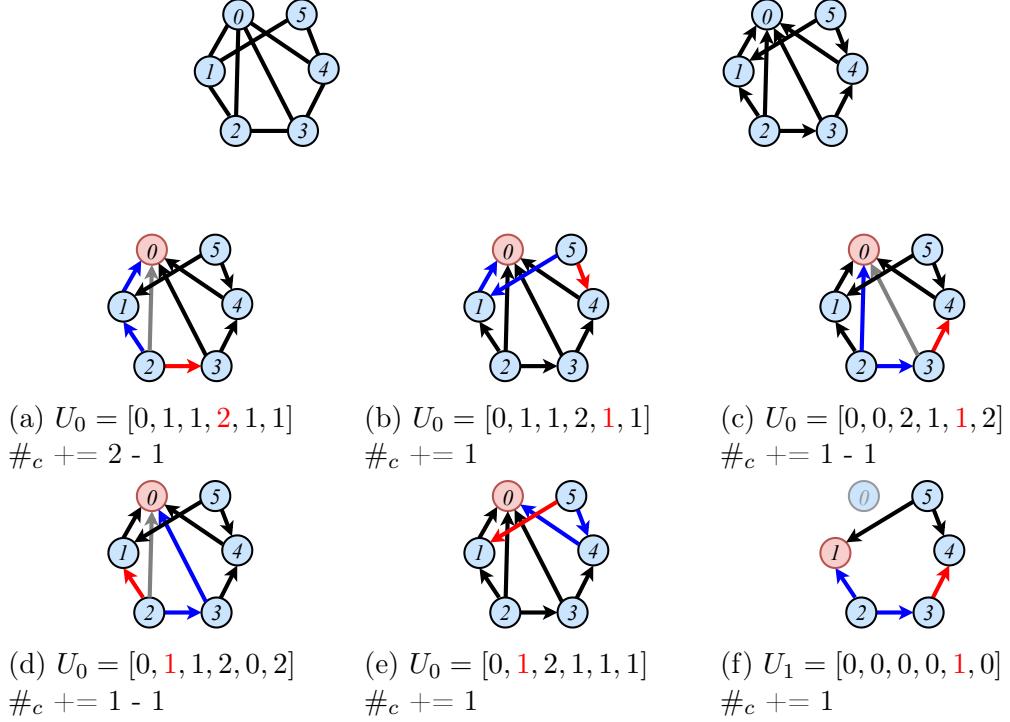


Figure 4.1: This figure outlines steps in our parallelization of Kowalik’s five-cycle counting algorithm where $\#_c$ is updated (Algorithm 4.1). Each subfigure considers a different $\{u, v, w, x\}$ from lines 13–14, and the corresponding U_v is displayed for each subfigure. For simplicity, the U_i hash tables are depicted as arrays, with the appropriate wedge counts stored at the index on the corresponding endpoint, and the updates to the parallel hash tables $T_{v,u}$ in lines 11–12 of Algorithm 4.1 are shown as subtracted directly from the corresponding U_v for a fixed u from line 9.

The vertices have already been relabeled by non-increasing degree and the entries in each U_v have already been computed (lines 10–12). The vertex v that we are considering on line 5 is colored in red. The edges colored in blue form wedges $v — u — w$, and the direction of those edges is irrelevant. The red edges represent the out-edge $w \rightarrow x$ on line 14. When w and v are neighbors (the edge is colored grey), the condition checked on line 16 returns true, and the subsequent line in each algorithm is executed (sub-figures (a), (c), and (d)). Otherwise, line 19 is executed (sub-figures (b), (e), and (f)). The final value of $\#_c$ is 4.

operations, which takes $O(1)$ work. In practice, we use thread-local variables to keep the count and sum them in the end to avoid heavy contention. We now show that the parallel algorithm is work-efficient and has polylogarithmic span.

Theorem 4.1. *Algorithm 4.1 can be performed in $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p., and $O(m\alpha)$ space on a graph with m edges and arboricity α .*

Proof. For line 3, we sort n integers in the range $[0, n - 1]$, which can be done in $O(n)$ work and $O(\log n)$ span w.h.p. using parallel integer sorting [265]. As discussed

Algorithm 4.2 – Five-Cycle Counting in ESCAPE Parallelized

```

1: procedure COUNT-FIVE-CYCLES( $G = (V, E)$ )
2:    $\#_c \leftarrow 0$ 
3:   Orient  $G$  using arboricity orientation to produce  $G^\rightarrow$ 
4:   parfor  $v \leftarrow 0$  to  $n - 1$  do
5:     Initialize an empty parallel hash table  $U_v$ 
6:     parfor  $w \in N^\leftarrow(v)$  do
7:       parfor  $u \in N(w)$  do
8:          $U_v[u] \leftarrow U_v[u] + 1$ 
9:       parfor  $w \in N^\rightarrow(v)$  do
10:        parfor  $u \in N^\rightarrow(w)$  do
11:           $U_v[u] \leftarrow U_v[u] + 1$ 
12:        parfor  $u \in N^\leftarrow(v)$  do
13:          parfor  $w \in N^\leftarrow(u)$  do
14:            parfor  $x \in N^\rightarrow(w)$  do
15:              if  $x \neq v$  and  $x \neq u$  then
16:                 $\#_c \leftarrow \#_c + U_v[x]$ 
17:              if  $w \in N(v)$  then
18:                 $\#_c \leftarrow \#_c - 1$ 
19:              if  $x \in N(u)$  then
20:                 $\#_c \leftarrow \#_c - 1$ 
21:   return  $\#_c$ 

```

in Section 4.4.1, line 4 can be implemented in $O(m)$ work and $O(\log^2 n)$ span [296]. As a result, the for-loops on lines 7 and 9 iterating over $u \in N_v(v)$ take at most $\min(\deg(u), \deg(v))$ iterations, and by Lemma 4.1, the total number of times we iterate through $w \in N_v(u)$ on each of lines 8, 11, and 13 is at most $2m\alpha$.

Since parallel hash tables can perform a batch of k operations in $O(k)$ work and $O(\log^* k)$ span w.h.p., the time complexities of lines 8 and 12 are given by $O(m\alpha)$ work and $O(\log^* n)$ span w.h.p. Then, the for-loop of line 14 has at most $O(\alpha)$ iterations because of the $O(\alpha)$ -orientation of the graph. In total, lines 15–19 are executed at most $O(\alpha) \cdot 2m\alpha = O(m\alpha^2)$ times, and again due to the parallel hash tables, the time complexity is given by $O(m\alpha^2)$ work and $O(\log^* n)$ span w.h.p. In all, the total time complexity is given by $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p.

Finally, this algorithm uses $O(m\alpha)$ space. Based on Lemma 4.1, the total number of keys stored over all U_v 's is upper-bounded by $O(m\alpha)$, as is the number of keys stored over all $T_{v,u}$'s (over all pairs (v, u)). The parallel hash table's space usage is linear in the number of keys [148]. Hence, the total space usage is $O(m\alpha)$. \square

4.4.3 ESCAPE Algorithm

Another serial five-cycle counting algorithm is given by Pinar et al. as part of ESCAPE, which counts all 5-vertex subgraphs in a graph serially [258].

The first step of the ESCAPE five-cycle counting algorithm is to orient the graph.

Example graph G G^\rightarrow under degree orientation

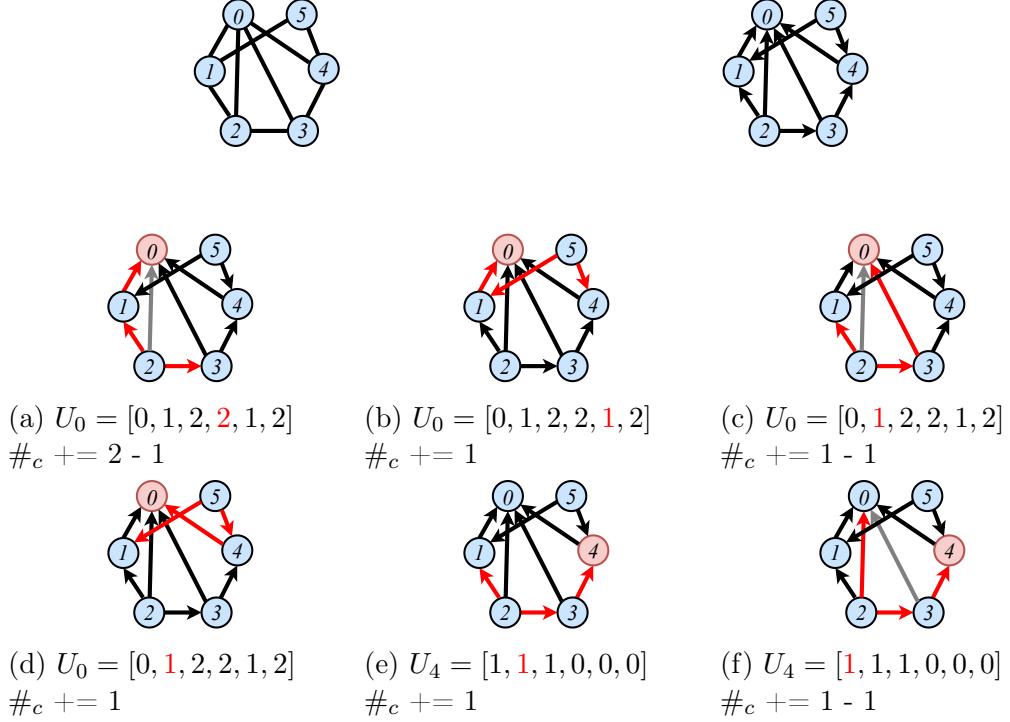


Figure 4.2: This figure outlines steps in the ESCAPE five-cycle counting algorithm where $\#_c$ is updated (Algorithm 4.2). Each subfigure considers a different $\{u, v, w, x\}$ from lines 12–15, and the corresponding U_v is displayed for each subfigure. For simplicity, the U_i hash tables are depicted as arrays, with the appropriate wedge counts stored at the index on the corresponding endpoint.

Note that the entries in each U_v have already been computed (lines 6–11). The vertex v that we are considering on line 4 is colored in red. The red edges represent the directed 3-paths $v \leftarrow u \leftarrow w \rightarrow x$ found on lines 12–15. Lines 17 and 19 check whether v and w or u and x are neighbors, respectively. When either of the conditions holds, the relevant edge is colored grey. Each grey edge subtracts one from the five-cycle count. Note that in sub-figures (a) and (c), the condition that v is adjacent to w from line 17 holds, and in sub-figure (f), the condition that u is adjacent to x from line 19 holds. In sub-figures (b), (d), and (e), neither conditions hold, and therefore 1 is not subtracted from the final count. The final value of $\#_c$ is 4.

The ESCAPE framework uses degree orientation and achieves a time complexity of $O(m^2)$. We note that, if instead an arboricity orientation is used, the five-cycle counting algorithm achieves an improved time complexity of $O(m\alpha^2)$. We include this modification in our parallelization of the ESCAPE five-cycle counting algorithm to achieve work-efficient bounds. The proof of the serial time complexity with the arboricity orientation follows directly from the proof of our parallel algorithm.

We present in Algorithm 4.2 our parallelization of the algorithm from ESCAPE, and an example is shown in Figure 4.2. We use $u \prec v$ to indicate that u precedes v .

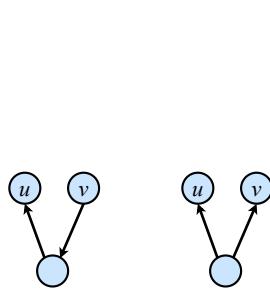


Figure 4.3: An inout-wedge (left) and an out-wedge (right).

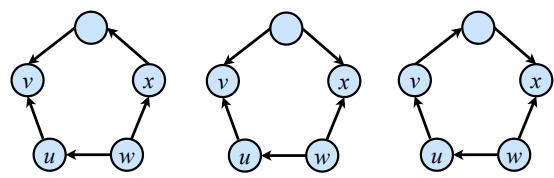


Figure 4.4: All three possible acyclic orientations of directed five-cycles, assuming a graph orientation induced by a total ordering of the vertices. All three forms have the component $v \leftarrow u \leftarrow w \rightarrow x$, which is a 3-path between v and x . They are completed by an inout-wedge from x to v , an out-wedge between v and x , and an inout-wedge from v to x , respectively.

in the ordering that produced the orientation, and so an edge from u to v exists in the directed graph G^\rightarrow if and only if $u \prec v$.

After orienting the graph using an arboricity orientation (line 3), for each vertex v (line 4), the algorithm counts all out-wedges and inout-wedges (see Figure 4.3). We denote the number of out-wedges with endpoints v and u by $W_{++}(v, u)$, and the number of inout-wedges with endpoints v and u , starting with a directed edge out of v , by $W_{+-}(v, u)$. For each v , the algorithm computes $W_{++}(u, v) + W_{+-}(u, v)$ on lines 6–8 and $W_{+-}(v, u)$ on lines 9–11, and stores these counts in a parallel hash table U_v . Note that lines 6–8 computes the total sum $W_{++}(u, v) + W_{+-}(u, v)$ rather than the constituent parts, so we iterate over all directed neighbors $w \in N^\leftarrow(v)$, and then all undirected neighbors $u \in N(w)$.

Figure 4.4 shows all possible orientations of acyclically directed five-cycles. We iterate over the 3-path shown in Figure 4.4 from vertex v to vertex x (lines 12–15), each of which can be completed by either an inout-wedge or an out-wedge with endpoints v and x , assuming $x \neq v$ and $x \neq u$. Now, any orientation of a five-cycle has one of the three configurations shown in Figure 4.4, where exactly one of the vertices can be assigned to be v . Thus, every 3-path between a pair (v, x) contributes $W_{+-}(v, x) + W_{++}(v, x) + W_{+-}(x, v)$ (which is stored in U from lines 6–11) to the five-cycle count. However, this over-counts five-cycles since the wedge and the 3-path may overlap. Lines 16–20 deal with the over-counting when adding the number of wedges to the total count.

In more detail, Line 16 first adds $U_v[x]$ to the count (again, assume that indexing an entry that does not exist in a hash table returns a value of 0). Line 17 checks if w is adjacent to v ; if so, depending on the direction of the edge between w and v , there is either an out-wedge or an inout-wedge on v, w , and x , that does not complete a five-cycle with the 3-path. Line 18 subtracts the five-cycle counted for this case. Similarly, line 19 checks if x is adjacent to u , and if so, there is either an out-wedge or an inout-wedge on v, u , and x , that does not complete a five-cycle; line 20 corrects this.

Similar to the parallelization of Kowalik’s algorithm, in theory we use atomic adds for all of the increments on the $\#_c$ variable, and in practice we use thread-local variables.

Theorem 4.2. *Algorithm 4.2 can be performed in $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p., and $O(m\alpha)$ space on a graph with m edges and arboricity α .*

Proof. As discussed in Section 4.4.1, line 3 can be implemented in $O(m)$ work and $O(\log^2 n)$ span [296]. Lines 6–11 go through all inout-wedges and out-wedges where v is an endpoint. Because of the arboricity orientation, there are at most $m\alpha$ inout-wedges and out-wedges. Each wedge is counted at most twice, and so lines 6–11 incur $O(m\alpha)$ hash table operations, which takes $O(m\alpha)$ work and $O(\log^* n)$ span w.h.p.

There are $O(m\alpha^2)$ 3-paths (i.e., $v \leftarrow u \leftarrow w \rightarrow x$) and each is encountered exactly once in the triply-nested for-loop (lines 12–20). Again, by using an arboricity orientation, the algorithm executes lines 15–20 for at most $O(m\alpha^2)$ times, which due to the hash table operations, takes $O(m\alpha^2)$ work and $O(\log^* n)$ span w.h.p.

Overall, the algorithm takes $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p.

The parallel hash tables and the space to store the accumulated cycle counts account for all of the additional space usage. Since each wedge results in at most two additional keys in the hash tables, the number of keys in all of the hash tables U_i is upper-bounded by twice the total number of out-wedges and inout-wedges. For the arboricity-oriented graph, there are $O(m\alpha)$ out-wedges and $O(m\alpha)$ inout-wedges, and so the number of keys across all hash tables is bounded by $O(m\alpha)$. Thus, the algorithm takes $O(m\alpha)$ space. \square

4.4.4 Approximate Counting

To further speed up five-cycle counting, we present here two techniques, *edge sparsification* and *colorful sparsification*, that allow us to compute approximate five-cycle counts in parallel. These techniques are based on prior work on approximate triangle, butterfly (bi-clique), and k -clique counting [251, 298, 296]. Both methods involve constructing a sparsified graph and running an exact parallel five-cycle counting algorithm on the sparsified graph. We prove that scaling the count returned gives an unbiased estimate of the total five-cycle count.

4.4.4.1 Edge Sparsification

In the *edge sparsification* method, we sparsify our input graph G by choosing to preserve each edge of G uniformly at random with probability p , in parallel. We call the resulting graph with only preserved edges G' . Then, we run our parallel five-cycle counting algorithm (Algorithm 4.1 or Algorithm 4.2) on G' , which outputs a count C . We output $Y = Cp^{-5}$ as the estimate for the global five-cycle count.

We prove the following theorem about the properties of this estimator.

Theorem 4.3. *Let X be the true five-cycle count in G , and let C be the five-cycle count in the sparsified graph G' using probability p . Let $Y = Cp^{-5}$. Then, $\mathbb{E}[Y] = X$*

and $\text{Var}[Y] = p^{-10}(X(p^5 - p^{10}) + \sum_{z=1}^3 s_z(p^{10-z} - p^{10}))$ where s_z is the number of pairs of five-cycles that share z edges.

Proof. Let X_i be the indicator variable denoting whether the i 'th five-cycle in G is preserved in G' . For a five-cycle to be preserved, all edges in the cycle must be preserved. This occurs with probability p^5 , since each edge is preserved with probability p . Thus, since the number of five-cycles in G' is given by $C = \sum_i X_i$, we have $\mathbb{E}[C] = \sum_i \mathbb{E}[X_i] = Xp^5$. Moreover, $\mathbb{E}[Y] = \mathbb{E}[Cp^{-5}] = Xp^5 \cdot p^{-5} = X$, as desired.

Now, by definition, the variance is given by $\text{Var}[Y] = \text{Var}[Cp^{-5}] = p^{-10}\text{Var}[\sum_i X_i] = p^{-10}(\sum_i \text{Var}[X_i] + \sum_{i \neq j} \text{Cov}[X_i, X_j])$. By definition, for all i , $\text{Var}[X_i] = \mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2 = \mathbb{E}[X_i] - \mathbb{E}[X_i]^2 = p^5 - p^{10}$. Thus, $\text{Var}[Y] = p^{-10}(X(p^5 - p^{10}) + \sum_{i \neq j} \text{Cov}[X_i, X_j])$.

Now, for $i \neq j$, $\text{Cov}[X_i, X_j] = \mathbb{E}[X_i X_j] - \mathbb{E}[X_i]\mathbb{E}[X_j]$. This term depends on the number of edges that cycles i and j share. The covariance is 0 if they share no edges, since X_i and X_j are independent in this case. For pairs of cycles sharing z edges where $z = 1, 2$, or 3 , we have $\mathbb{E}[X_i X_j] = \Pr[X_i = 1] \cdot \Pr[X_j = 1 | X_i = 1] = p^5 \cdot p^{5-z} = p^{10-z}$. The latter term is because the z shared edges are guaranteed to be preserved by the condition, so it remains to compute the probability of preserving the remaining $5 - z$ edges. Moreover, $\mathbb{E}[X_i]\mathbb{E}[X_j] = p^{10}$, so in total, $\text{Cov}[X_i, X_j] = p^{10-z} - p^{10}$. If we let s_z denote the number of pairs of five-cycles that share z edges, then $\sum_{i \neq j} \text{Cov}[X_i, X_j] = \sum_{z=1}^3 s_z(p^{10-z} - p^{10})$. In total, we have $\text{Var}[Y] = p^{-10}(X(p^5 - p^{10}) + \sum_{z=1}^3 s_z(p^{10-z} - p^{10}))$, as desired. \square

Corollary 4.1. *Let Δ denote the maximum degree of any vertex in G , and let X be the true five-cycle count in G . Let Y be the approximate five-cycle count computed using probability p . If $p \geq \max(208\Delta^3/X, (208\Delta^2/X)^{1/2}, (208\Delta/X)^{1/3}, (208/X)^{1/5})$, then $\text{Var}[Y] \leq X^2/8$.*

Proof. First, from Theorem 4.3, $\text{Var}[Y] = p^{-10}(X(p^5 - p^{10}) + \sum_{z=1}^3 s_z(p^{10-z} - p^{10})) \leq Xp^{-5} + s_3p^{-3} + s_2p^{-2} + s_1p^{-1}$ where s_z is the number of pairs of five-cycles that share z edges. Note that $s_z \leq X \binom{5}{z} \Delta^{4-z}$. This follows from fixing each five-cycle, and upper bounding the number of possible ways to extend the five-cycle to a new five-cycle such that the extension shares z edges with the original five-cycle and requires $5 - z$ additional edges. Substituting for s_z and the value of p in the corollary statement, we have $\text{Var}[Y] \leq Xp^{-5} + 10X\Delta p^{-3} + 10X\Delta^2 p^{-2} + 5X\Delta^3 p^{-1} \leq X(208/X)^{(1/5) \cdot (-5)} + 10X\Delta(208\Delta/X)^{(1/3) \cdot (-3)} + 10X\Delta^2(208\Delta^2/X)^{(1/2) \cdot (-2)} + 5X\Delta^3(208\Delta^3/X)^{-1} = X^2/8$. \square

Now, let Z be the mean of β independent instances of Y , where we define β later. Then, by Markov's inequality, $\Pr[|Z - X| \geq \varepsilon X] = \Pr[(Z - X)^2 \geq \varepsilon^2 X^2] \leq \frac{\mathbb{E}[(Z - X)^2]}{\varepsilon^2 X^2} = \frac{\text{Var}[Z]}{\varepsilon^2 X^2} = \frac{\text{Var}[Y]}{\varepsilon^2 X^2 \beta}$. Using Corollary 4.1, if $p \geq \max(208\Delta^3/X, (208\Delta^2/X)^{1/2}, (208\Delta/X)^{1/3}, (208/X)^{1/5})$, then $\text{Var}[Y] \leq X^2/8$. Thus, $\Pr[|Z - X| \geq \varepsilon X] \leq \frac{X^2/8}{\varepsilon^2 X^2 \beta} = \frac{1}{8\varepsilon^2 \beta}$, and if we let $\beta = \frac{1}{8\varepsilon^2 \delta}$, then Z is a (ε, δ) -estimator of X . Notably, if we increase the number of samples β by a factor of b , then we can either decrease δ by a factor of b or ε by a factor of b^2 . The opposite relation holds if we decrease β by a factor of b ; namely, we can either increase δ by a factor of b or increase ε by a factor of b^2 .

We further prove the following theorem about the complexity of our sampling algorithm based on edge sparsification.

Theorem 4.4. *Our parallel five-cycle sampling algorithm using edge sparsification with probability p gives an unbiased estimate of the global five-cycle count and takes $O(pma^2 + m)$ expected work and $O(\log^2 n)$ span w.h.p.*

Proof. First, Theorem 4.3 shows that our algorithm gives an unbiased estimate of the global count. We now discuss the work and span of our algorithm. Choosing each edge to preserve uniformly at random takes $O(m)$ work and $O(1)$ span. Creating a subgraph containing preserved edges can be done using a parallel filter and a parallel prefix sum in $O(m)$ work and $O(\log m)$ span. Since each edge is preserved with probability p , the subgraph has pm edges in expectation. The arboricity of the subgraph is upper bounded by the arboricity of the original graph, α . Thus, by Theorems 4.1 and 4.2, running our parallel five-cycle counting algorithm on the subgraph takes $O(pma^2)$ expected work and $O(\log^2 n)$ span w.h.p. In total, we have $O(pma^2 + m)$ expected work and $O(\log^2 n)$ span w.h.p. \square

4.4.4.2 Colorful Sparsification

In the *colorful sparsification* method, we sparsify our input graph G by coloring each vertex of G uniformly at random with one of c colors, and preserving edges only if both endpoints are of the same color. Note that we can perform this sparsification in parallel. Let $p = 1/c$. We call the resulting graph with only preserved edges G' . Then, we run our parallel five-cycle counting algorithm (Algorithm 4.1 or Algorithm 4.2) on G' , which outputs a count C . We output $Y = Cp^{-4}$ as the estimate for the global five-cycle count.

We prove the following theorem about the properties of this estimator.

Theorem 4.5. *Let X be the true five-cycle count in G , and let C be the five-cycle count in the sparsified graph G' using c colors. Let $p = 1/c$, and let $Y = Cp^{-4}$. Then, $\mathbb{E}[Y] = X$ and $\text{Var}[Y] = p^{-8}(X(p^4 - p^8) + \sum_{z=2}^4 t_z(p^{9-z} - p^8))$ where t_z is the number of pairs of five-cycles that share z vertices.*

Proof. Let X_i be the indicator variable denoting whether the i 'th five-cycle in G is preserved in G' . For a five-cycle to be preserved, all edges in the cycle must be preserved, or in other words, all vertices in the cycle must be colored using the same color. This occurs with probability p^4 , since after fixing the color of one vertex v in the cycle, the remaining 4 vertices must have the same color as v . Thus, since the number of five-cycles in G' is given by $C = \sum_i X_i$, we have $\mathbb{E}[C] = \sum_i \mathbb{E}[X_i] = Xp^4$. Moreover, $\mathbb{E}[Y] = \mathbb{E}[Cp^{-4}] = Xp^4 \cdot p^{-4} = X$, as desired.

Now, by definition, the variance is given by $\text{Var}[Y] = \text{Var}[Cp^{-4}] = p^{-8}\text{Var}[\sum_i X_i] = p^{-8}(\sum_i \text{Var}[X_i] + \sum_{i \neq j} \text{Cov}[X_i, X_j])$. By definition, for all i , $\text{Var}[X_i] = \mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2 = \mathbb{E}[X_i] - \mathbb{E}[X_i]^2 = p^4 - p^8$. Thus, $\text{Var}[Y] = p^{-8}(X(p^4 - p^8) + \sum_{i \neq j} \text{Cov}[X_i, X_j])$.

Now, for $i \neq j$, $\text{Cov}[X_i, X_j] = \mathbb{E}[X_i X_j] - \mathbb{E}[X_i] \mathbb{E}[X_j]$. This term depends on the number of vertices that cycles i and j share. The covariance is 0 if they share

no vertices, and the covariance is also 0 if they share one vertex, since the event that the remaining vertices of each cycle have the same color as the shared vertex is independent between the two cycles. For pairs of cycles sharing z vertices where $z = 2, 3$, or 4 , we note that $\mathbb{E}[X_i X_j] = \Pr[X_i = 1] \cdot \Pr[X_j = 1 | X_i = 1] = p^4 \cdot p^{5-z} = p^{9-z}$. The latter term is because the z shared vertices are guaranteed to be the same color by the condition, and so it remains to compute the probability that the remaining $5 - z$ vertices are also of the same color. Moreover, $\mathbb{E}[X_i] \mathbb{E}[X_j] = p^8$, and so in total, $\text{Cov}[X_i, X_j] = p^{9-z} - p^8$. If we let t_z denote the number of pairs of five-cycles that share z vertices, then $\sum_{i \neq j} \text{Cov}[X_i, X_j] = \sum_{z=2}^4 t_z (p^{9-z} - p^8)$. In total, we have $\mathbb{V}\text{ar}[Y] = p^{-8}(X(p^4 - p^8) + \sum_{z=2}^4 t_z (p^{9-z} - p^8))$, as desired. \square

Corollary 4.2. *Let Δ denote the maximum degree of any vertex in G , and let X be the true five-cycle count in G . Let Y be the approximate five-cycle count computed using c colors, where $p = 1/c$. If $p \geq \max(208\Delta^3/X, (208\Delta^2/X)^{1/2}, (208\Delta/X)^{1/3}, (208/X)^{1/4})$, then $\mathbb{V}\text{ar}[Y] \leq X^2/8$.*

Proof. First, from Theorem 4.5, $\mathbb{V}\text{ar}[Y] = p^{-8}(X(p^4 - p^8) + \sum_{z=2}^4 t_z (p^{9-z} - p^8)) \leq Xp^{-4} + t_4 p^{-3} + t_3 p^{-2} + t_2 p^{-1}$ where t_z is the number of pairs of five-cycles that share z vertices. Note that $t_z \leq X \binom{5}{z} \Delta^{5-z}$. This follows from fixing each five-cycle, and upper bounding the number of possible ways to extend the five-cycle to a new five-cycle such that the extension shares z vertices with the original five-cycle and requires $5 - z$ additional vertices. Substituting for t_z and the value of p in the corollary statement, we have $\mathbb{V}\text{ar}[Y] \leq Xp^{-4} + 5X\Delta p^{-3} + 10X\Delta^2 p^{-2} + 10X\Delta^3 p^{-1} \leq X(208/X)^{(1/4)\cdot(-4)} + 5X\Delta(208\Delta/X)^{(1/3)\cdot(-3)} + 10X\Delta^2(208\Delta^2/X)^{(1/2)\cdot(-2)} + 10X\Delta^3(208\Delta^3/X)^{-1} = X^2/8$. \square

Again, let Z be the mean of β independent instances of Y , where we define β later. Then, by Markov's inequality, $\Pr[|Z - X| \geq \varepsilon X] = \Pr[(Z - X)^2 \geq \varepsilon^2 X^2] \leq \frac{\mathbb{E}[(Z - X)^2]}{\varepsilon^2 X^2} = \frac{\mathbb{V}\text{ar}[Z]}{\varepsilon^2 X^2} = \frac{\mathbb{V}\text{ar}[Y]}{\varepsilon^2 X^2 \beta}$. Using Corollary 4.2, if $p \geq \max(208\Delta^3/X, (208\Delta^2/X)^{1/2}, (208\Delta/X)^{1/3}, (208/X)^{1/4})$, then $\mathbb{V}\text{ar}[Y] \leq X^2/8$. Thus, $\Pr[|Z - X| \geq \varepsilon X] \leq \frac{X^2/8}{\varepsilon^2 X^2 \beta} = \frac{1}{8\varepsilon^2 \beta}$, and if we let $\beta = \frac{1}{8\varepsilon^2 \delta}$, then Z is a (ε, δ) -estimator of X .

We further prove the following theorem about the complexity of our sampling algorithm based on colorful sparsification.

Theorem 4.6. *Our parallel five-cycle sampling algorithm using colorful sparsification with c colors where $p = 1/c$ gives an unbiased estimate of the global five-cycle count and takes $O(pma^2 + m)$ expected work and $O(\log^2 n)$ span w.h.p.*

Proof. First, Theorem 4.5 shows that our algorithm gives an unbiased estimate of the global count. We now discuss the work and span of our algorithm. Coloring each vertex takes $O(n)$ work and $O(1)$ span. Creating a subgraph containing edges where endpoints are the same color can be done using a parallel filter and a parallel prefix sum in $O(m)$ work and $O(\log m)$ span. Since each edge is preserved with probability $p = 1/c$, the subgraph has pm edges in expectation. The rest of the proof is the same as the proof of Theorem 4.4. \square

4.4.5 Implementation

We implement the serial and parallel versions of Kowalik’s algorithm and Pinar et al.’s algorithm, as well as our approximate algorithms, using the Graph Based Benchmark Suite framework (GBBS) [97, 106]. In GBBS, graphs are represented in compressed sparse row (CSR) format, which does not allow us to check edge existence in $O(1)$ work; instead, we sort the neighbor lists in the preprocessing step and use binary search to locate neighbors.

In our approximate counting algorithms, the sparsification is done by simply using a hash function to sample the edges or assign a color to each vertex, and then applying a filter and prefix sum to create the subgraph in CSR format. The rest of this section describes the optimizations for five-cycle counting either on the original graph (for exact counting) or on the sparsified graph (for approximate counting).

In our parallel implementations, we only parallelize the outer for-loop for each algorithm since there is sufficient parallelism provided by the outer for-loop alone. For Kowalik’s algorithm, instead of using parallel integer sort, we use a cache-efficient implementation of parallel sample sort [48] from GBBS to sort the vertices by degree. We also use vertex-indexed size- n arrays instead of parallel hash tables for U_i and $T_{i,j}$.

Thread-local Data Structures. As we parallelize the outer for-loop, the arrays U_i in both algorithms must be allocated per iteration. We optimize this allocation by using the `parallel_for_alloc` construct in GBBS, which allocates one array per thread and reuses this space over iterations. Each iteration uses the array as a local array, and so this incurs no synchronization overhead. With this optimization, the algorithm only requires $O(Pn)$ space, where P is the number of processors.

Fast Reset. Additionally, the thread-local arrays must be reset after each iteration of the outer for-loop. Depending on the structure of the graph, the array can be sparse, and naively resetting the entire array incurs $O(n^2)$ extra work, which is costly. We use a separate thread-local array to record the non-zero entries and reset only those entries after an iteration of the outer for-loop. This optimization at most doubles the space requirement for the algorithm, but drastically improves the running time by avoiding unnecessary writes.

Work Scheduling. The naive parallelization of the five-cycle counting algorithms blocks a fixed number of vertices together and processes them in series. For our experiments, we use a block size of 16, which we found to give the best performance in this setting. However, due to the nature of the algorithm, the amount of work per vertex is not uniform. This is particularly true for Kowalik’s algorithm, which processes vertices in non-increasing degree order and deletes a vertex after processing it. The number of five-cycles that can be counted under a given vertex v in the outermost loop falls off rapidly with the vertex’s degree rank. In our work scheduling optimization, we block vertices together into groups that require similar amounts of work by estimating the work required for each vertex. We use the sum

of the degrees of a vertex’s neighbors as the estimator. That is, for each vertex v , we estimate the amount of work done on the vertex to be $\sum_{w \in N(v)} \deg(w)$.

4.5 Experiments

Environment. We run our experiments on a `c5.18xlarge` AWS EC2 instance, which is a dual-processor system with 18 cores per processor (2-way hyper-threading, 3.00GHz Intel Xeon(R) Platinum 8124M processors), and 144 GiB of main memory. We use Cilk Plus for parallelism [206, 53]. We use the `g++` compiler (version 8.2.1) with the `-O3` flag.

We test the performance of our parallel exact and approximate five-cycle counting algorithms. Our parallel implementations use all of the optimizations described in Section 4.4.5, except that we test the performance with and without the work scheduling optimization. We compare the performance of the exact parallel implementations against our implementations of Kowalik’s algorithm and the ESCAPE algorithm. We also tested the performance of the exact serial five-cycle counting algorithm in the ESCAPE package, the fastest known implementation of exact five-cycle counting. This algorithm is embedded inside the ESCAPE code for counting all five-vertex patterns, and so we obtained timings by running only the five-cycle counting portion of the code. We found our exact serial ESCAPE implementation to be 1.1–2.95x faster than the one provided in the ESCAPE package, and hence present only our running times in the tables.

We also test the effect of using different parallel arboricity ordering algorithms. Besides Goodrich-Pszona, Barenboim-Elkin, and k -core, we also tested non-decreasing degree ordering as an approximation of degeneracy ordering. Intuitively, it limits the out-degree of the graph by directing edges from lower-degree vertices to higher-degree neighbors. Also, note that we use the parallel Gooodrich-Pszona and Barenboim-Elkin implementations by Shi et al. [296], and the parallel k -core implementation from Julienne [95], which is work-efficient but does not have polylogarithmic span. Finally, we test the accuracy of our approximate algorithms, and their speedups over the exact algorithms.

We perform these tests on a number of real-world graphs from the Stanford Network Analysis Platform [207]. Table 4.1 describes the properties of these graphs. All graphs are simple, unweighted, and undirected.

Exact Serial Five-cycle Counting. Table 4.2 lists the running times of the two exact serial five-cycle counting algorithms. Our serial Kowalik implementation always outperforms our serial ESCAPE implementation, and the difference in running times between the ESCAPE algorithm and Kowalik’s algorithm grows as the graph size grows. The serial Kowalik algorithm achieves between 6.37–14.77x speedup over our serial ESCAPE implementation, and between 7–38.91x speedup over the original ESCAPE implementation.

| Dataset | $ V $ | $ E $ | # 5-cycles |
|-----------------------------|--------------------|--------------------|--------------------|
| email-Eu-Core (email) | 1005 | 32128 | 245,585,096 |
| com-DBLP (dblp) | 425957 | 2.10×10^6 | 3,440,276,253 |
| com-YouTube (youtube) | 1.16×10^6 | 5.98×10^6 | 34,643,647,544 |
| com-LiveJournal (lj) | 4.03×10^6 | 6.94×10^7 | 6,668,633,603,006 |
| com-Orkut (orkut) | 3.27×10^6 | 2.34×10^8 | 42,499,585,526,270 |
| com-Friendster (friendster) | 1.25×10^8 | 3.61×10^9 | 96,281,214,210,322 |

Table 4.1: Relevant statistics of our input graphs.

Exact Parallel Five-cycle Counting. Table 4.2 shows the best performance with the exact Kowalik and ESCAPE algorithms with 1 thread, 36 cores without hyper-threading, and 72 hyper-threads, without the work scheduling optimization. We see that the algorithms achieve decent speedup without the work scheduling optimization. The parallel speedup plateaus from 36 to 72 hyper-threads, especially for the parallelization of Kowalik’s algorithm. The speedup for the Kowalik algorithm is usually lower since, due to its degree ordering, it does not distribute work evenly across vertices, but rather concentrates the work on high-degree vertices. Our naive parallel algorithm groups a fixed number of vertices together regardless of whether they are high- or low-degree, resulting in unbalanced work distribution across workers.

From both the serial and parallel running times, we observe that the ESCAPE algorithm, with all of the same optimizations as the parallel Kowalik’s algorithm, generally has about a 10x slowdown compared to Kowalik’s algorithm. We attribute this difference to the discrepancy in the number of edge queries the two algorithms must perform. Since we store graphs in CSR format, each edge query requires a binary search. In Kowalik’s algorithm, an edge query is performed for every (v, w) -pair, and it can be performed just before the for-loop with x , so there only needs to be $O(m\alpha)$ binary searches. In the ESCAPE algorithm, (x, u) needs to be queried $O(m\alpha^2)$ times. Table 4.3 shows that the ESCAPE algorithm does significantly more binary searches than Kowalik’s algorithm.

Compared to the state-of-the-art serial five-cycle counting implementation provided in the ESCAPE package, without the work scheduling optimization, our parallel Kowalik implementation achieves a speedup of 33.78–229.79x, and our parallel ESCAPE implementation achieves a speedup of 5.73–23.16x.

Work Scheduling Optimization. We present the best running times of the exact parallel Kowalik and ESCAPE algorithms using work scheduling in Table 4.4, and Table 4.5 shows the running times for different orientations. Compared to Table 4.2, we see that the work scheduling optimization is effective on both parallel algorithms. It allows exact five-cycle counting to be performed on the friendster graph in under 2.5 hours using the parallel Kowalik algorithm. Figure 4.5 shows the relative running time of the parallel Kowalik algorithm with 72 hyper-threads with different arboricity orientation subroutines, including Goodrich-Pszona, Barenboim-Elkin, degree ordering, and k -core orientation, with and without the work scheduling op-

| Dataset | Exact Serial Algorithm | | Exact Parallel Kowalik Algorithm | | | Exact Parallel ESCAPE Algorithm | | | | |
|---------|------------------------|---------|----------------------------------|---------------------|---------------------|---------------------------------|----------------------|----------------------|----------------------|------|
| | Running times (s) | | Running times (s) | | | Speedup | | | | |
| | T_E | T_K | T_1 | T_{36} | T_{36h} | T_1/T_{36h} | T_1 | T_{36} | T_{36h} | |
| email | 0.36 | 0.026 | 0.027 ^b | 0.0027 ^g | 0.0029 ^b | 9.3 | 0.376 ^b | 0.017 ^g | 0.0177 ^g | 21.2 |
| dblp | 2.93 | 0.46 | 0.48 ^g | 0.046 ^b | 0.046 ^g | 10.4 | 3.24 ^g | 0.34 ^k | 0.277 ^k | 11.7 |
| youtube | 40.70 | 4.73 | 4.80 ^b | 1.73 ^g | 1.69 ^g | 2.8 | 43.94 ^g | 14.5 ^g | 9.96 ^g | 4.4 |
| lj | 2579.34 | 174.60 | 174.60 ^b | 29.0 ^g | 25.72 ^g | 6.8 | 2582.30 ^g | 426.38 ^g | 308.41 ^g | 8.4 |
| orkut | 38K | 2878.38 | 2867.07 ^b | 504.61 ^b | 487.4 ^b | 5.9 | TL | 8192.33 ^g | 6384.24 ^g | - |

Table 4.2: Running times (seconds) of the two exact serial implementations and the two exact parallel five-cycle counting implementations without the work scheduling optimization. All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. For the serial algorithms, T_E is our implementation of the ESCAPE algorithm with arboricity orientation, and T_K is our implementation of the serial Kowalik's algorithm. The serial runtimes are measured using the Goodrich-Pszona degeneracy ordering algorithm. For the parallel algorithms, we use superscripts to indicate the orientation that achieved the best running time. ^g refers to Goodrich-Pszona, ^b refers to Barenboim-Elkin, and ^k refers to k -core orientation. Note that degree orientation is never the fastest orientation. For the parallel algorithms, we list the runtimes obtained on a single thread (T_1), 36 cores without hyper-threading (T_{36}), and 36 cores with hyper-threading (T_{36h}). We also tested all implementations on friendster, but they all exceeded the time limit.

| Dataset | # binary searches | | |
|---------|-----------------------|-----------------------|----------------|
| | Kowalik | ESCAPE | ESCAPE/Kowalik |
| email | 5.15×10^5 | 2.98×10^7 | 58 |
| dblp | 8.41×10^6 | 2.32×10^8 | 28 |
| youtube | 6.28×10^7 | 2.96×10^9 | 47 |
| lj | 1.39×10^9 | 1.72×10^{11} | 124 |
| orkut | 1.25×10^{10} | 3.44×10^{12} | 273 |

Table 4.3: These are the number of binary searches each exact algorithm performed for each dataset, and the ratio of the number of binary searches in the ESCAPE algorithm to Kowalik’s algorithm.

| Dataset | Exact Parallel Kowalik Algorithm | | | Exact Parallel ESCAPE Algorithm | | |
|------------|----------------------------------|----------------------|---------------|---------------------------------|---------------------|---------------|
| | Running times (s) | | Speedup | Running times (s) | | Speedup |
| | T_1 | T_{36h} | T_1/T_{36h} | T_1 | T_{36h} | T_1/T_{36h} |
| email | 0.0265 ^b | 0.00252 [◦] | 10.5 | 0.357 ^b | 0.0165 ^b | 21.6 |
| dblp | 0.46 ^g | 0.0143 ^g | 32.2 | 3.07 ^b | 0.0866 ^g | 35.5 |
| youtube | 4.75 ^b | 0.338 ^b | 14.1 | 43.32 ^g | 1.42 ^g | 30.0 |
| lj | 171.92 ^b | 5.85 ^g | 29.4 | 2510.97 ^b | 58.75 ^g | 42.7 |
| orkut | 2858.18 ^b | 136.98 ^g | 20.9 | TL | 1269.1 ^b | – |
| friendster | TL | 8417.31 ^g | – | TL | TL | – |

Table 4.4: Single-thread (T_1) and 36-core with hyper-threading (T_{36h}) running times (seconds) of the exact parallel Kowalik and ESCAPE algorithms with the work scheduling optimization, and their parallel speedups. All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. The superscripts indicate the orientation that achieved the best runtime. ^g refers to Goodrich-Pszona, ^b refers to Barenboim-Elkin, and [◦] refers to degree orientation. In Table 4.5, we present the data for all orientations.

timization. The comparison shows that work scheduling significantly improves the running time and scaling of the parallel Kowalik algorithm.

Throughout our tests, we use the sum of neighbors’ degrees as the estimator of the amount of work. Other work estimators were tested, including a simple out-degree count and the two-hop neighbor out-degree sum, but did not result in improved performance.

Compared to the state-of-the-art exact serial five-cycle counting implementation provided in the ESCAPE package, using the work scheduling optimization, our parallel Kowalik implementation achieves a speedup of 162.70–818.12x, and our parallel ESCAPE implementation achieves a speedup of 23.56–72.13x. Compared to our best serial baselines, our parallel Kowalik implementation achieves a speedup of 10.5–32.2x.

| Exact Kowalik | | | | | | Exact ESCAPE | | | | | | | | |
|---------------|-------------------|---------------|---------------|-------------|---------------|-------------------|---------------|--------------|---------------|----------------|-------------------|-----------|---------------|--|
| Dataset | Running times (s) | | Speedup | | T_1 | Running times (s) | | Speedup | | T_1 | Running times (s) | | Speedup | |
| | T_1 | T_{36h} | T_1/T_{36h} | T_1 | T_{36h} | T_1/T_{36h} | T_1 | T_{36h} | T_1/T_{36h} | | T_1 | T_{36h} | T_1/T_{36h} | |
| email | 0.0267 | 0.00289 | 9.3 | 0.396 | 0.0174 | 22.7 | 0.0265 | 0.00294 | 9.0 | 0.357 | 0.0165 | 21.6 | | |
| dblp | 0.459 | 0.0143 | 32.2 | 3.17 | 0.0866 | 36.6 | 0.465 | 0.0147 | 31.6 | 3.07 | 0.0992 | 31.0 | | |
| youtube | 4.81 | 0.361 | 13.3 | 43.3 | 1.42 | 30.6 | 4.75 | 0.338 | 14.0 | 48.05 | 1.43 | 33.6 | | |
| jj | 174.40 | 5.85 | 29.8 | 2546.95 | 58.75 | 43.4 | 171.92 | 5.95 | 28.9 | 2510.97 | 59.03 | 42.5 | | |
| orkut | 2867.78 | 136.98 | 20.9 | TL | 1552.70 | — | 2858.18 | 139.87 | 20.4 | TL | 1269.07 | — | | |

| Exact Kowalik | | | | | | Exact ESCAPE | | | | | | | | |
|---------------|-------------------|----------------|---------------|----------|-----------|-------------------|---------|-----------|---------------|----------|-------------------|-----------|---------------|--|
| Dataset | Running times (s) | | Speedup | | T_1 | Running times (s) | | Speedup | | T_1 | Running times (s) | | Speedup | |
| | T_1 | T_{36h} | T_1/T_{36h} | T_1 | T_{36h} | T_1/T_{36h} | T_1 | T_{36h} | T_1/T_{36h} | | T_1 | T_{36h} | T_1/T_{36h} | |
| email | 0.0276 | 0.00252 | 10.9 | 1.49 | 0.0403 | 37.0 | 0.0278 | 0.00304 | 9.2 | 0.675 | 0.0245 | 27.6 | | |
| dblp | 0.472 | 0.0144 | 32.7 | 10.71 | 0.773 | 13.9 | 0.473 | 0.0161 | 29.3 | 7.80 | 0.240 | 32.4 | | |
| youtube | 4.79 | 0.344 | 13.9 | 2177.52 | 59.61 | 36.5 | 5.84 | 0.531 | 11.0 | 922.19 | 23.62 | 39.0 | | |
| jj | 178.02 | 5.96 | 29.9 | 16651.40 | 417.00 | 39.9 | 198.16 | 8.06 | 24.6 | 11151.50 | 309.71 | 36.0 | | |
| orkut | 2949.47 | 139.37 | 21.2 | TL | 16129.4 | — | 3100.79 | 147.83 | 21.0 | TL | 7113.44 | — | | |

| (a) Goodrich-Pszona | | | | | | (b) Barenboim-Elkin | | | | | | | | |
|---------------------|-------------------|----------------|---------------|----------|-----------|---------------------|---------|-----------|---------------|----------|-------------------|-----------|---------------|--|
| Exact Kowalik | | | | | | Exact ESCAPE | | | | | | | | |
| Dataset | Running times (s) | | Speedup | | T_1 | Running times (s) | | Speedup | | T_1 | Running times (s) | | Speedup | |
| | T_1 | T_{36h} | T_1/T_{36h} | T_1 | T_{36h} | T_1/T_{36h} | T_1 | T_{36h} | T_1/T_{36h} | | T_1 | T_{36h} | T_1/T_{36h} | |
| email | 0.0276 | 0.00252 | 10.9 | 1.49 | 0.0403 | 37.0 | 0.0278 | 0.00304 | 9.2 | 0.675 | 0.0245 | 27.6 | | |
| dblp | 0.472 | 0.0144 | 32.7 | 10.71 | 0.773 | 13.9 | 0.473 | 0.0161 | 29.3 | 7.80 | 0.240 | 32.4 | | |
| youtube | 4.79 | 0.344 | 13.9 | 2177.52 | 59.61 | 36.5 | 5.84 | 0.531 | 11.0 | 922.19 | 23.62 | 39.0 | | |
| jj | 178.02 | 5.96 | 29.9 | 16651.40 | 417.00 | 39.9 | 198.16 | 8.06 | 24.6 | 11151.50 | 309.71 | 36.0 | | |
| orkut | 2949.47 | 139.37 | 21.2 | TL | 16129.4 | — | 3100.79 | 147.83 | 21.0 | TL | 7113.44 | — | | |

| (c) Degree | | | | | | (d) k -Core | | | | | | | | |
|---------------|-------------------|----------------|---------------|----------|-----------|-------------------|---------|-----------|---------------|----------|-------------------|-----------|---------------|--|
| Exact Kowalik | | | | | | Exact ESCAPE | | | | | | | | |
| Dataset | Running times (s) | | Speedup | | T_1 | Running times (s) | | Speedup | | T_1 | Running times (s) | | Speedup | |
| | T_1 | T_{36h} | T_1/T_{36h} | T_1 | T_{36h} | T_1/T_{36h} | T_1 | T_{36h} | T_1/T_{36h} | | T_1 | T_{36h} | T_1/T_{36h} | |
| email | 0.0276 | 0.00252 | 10.9 | 1.49 | 0.0403 | 37.0 | 0.0278 | 0.00304 | 9.2 | 0.675 | 0.0245 | 27.6 | | |
| dblp | 0.472 | 0.0144 | 32.7 | 10.71 | 0.773 | 13.9 | 0.473 | 0.0161 | 29.3 | 7.80 | 0.240 | 32.4 | | |
| youtube | 4.79 | 0.344 | 13.9 | 2177.52 | 59.61 | 36.5 | 5.84 | 0.531 | 11.0 | 922.19 | 23.62 | 39.0 | | |
| jj | 178.02 | 5.96 | 29.9 | 16651.40 | 417.00 | 39.9 | 198.16 | 8.06 | 24.6 | 11151.50 | 309.71 | 36.0 | | |
| orkut | 2949.47 | 139.37 | 21.2 | TL | 16129.4 | — | 3100.79 | 147.83 | 21.0 | TL | 7113.44 | — | | |

Table 4.5: Single-thread (T_1) and 36-core with hyper-threading (T_{36h}) running times (seconds) of the exact parallel Kowalik and ESCAPE algorithms with the work scheduling optimization using all four orientations. All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. The bold values mark the best serial and parallel runtimes for each of Kowalik and ESCAPE, out of the four orientations, which are used in Table 4.4.

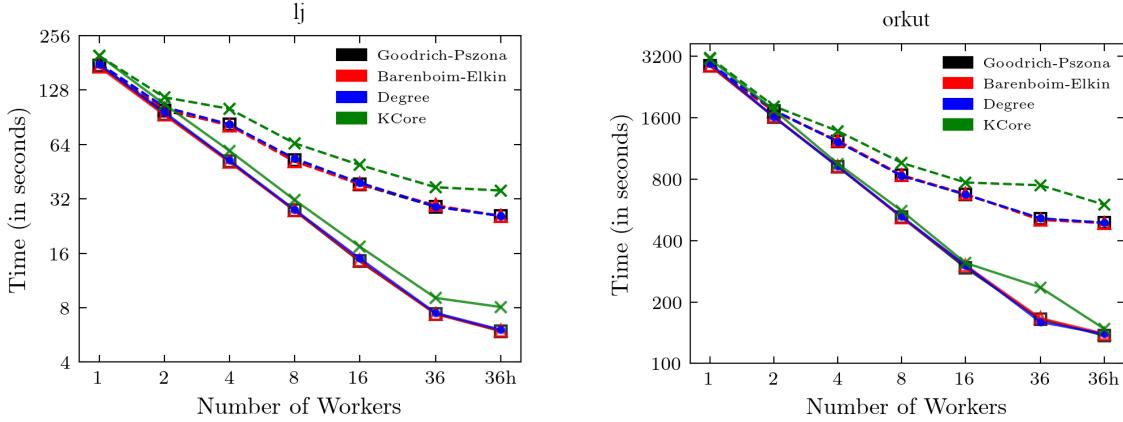


Figure 4.5: Running time of the exact parallel Kowalik algorithm vs. number of threads. “36h” is 36 cores with hyper-threading. Dashed lines indicate that the work scheduling optimization is disabled and solid lines indicate that the work scheduling optimization is enabled. The lines for Goodrich-Pszona, Barenboim-Elkin, and degree ordering overlap each other for the most part.

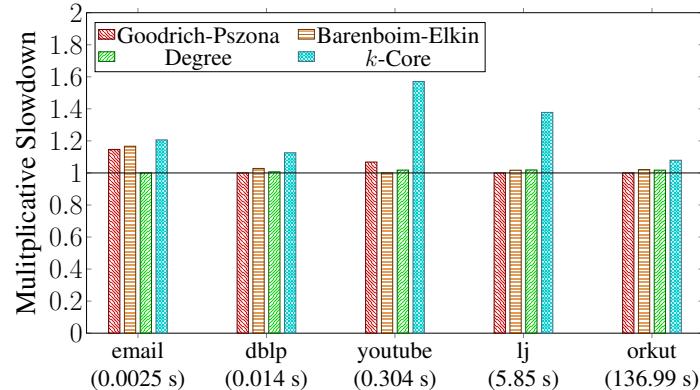


Figure 4.6: Exact five-cycle counting times using our parallel Kowalik implementation, excluding preprocessing steps like relabeling and orienting the graph, under different orientation schemes for each of the graphs, using 36 cores with hyper-threading. The fastest time is labeled under each graph.

Graph Orientation. Figure 4.6 compares the performance of our exact parallel Kowalik implementation using different parallel orientation schemes [296, 95]. The Goodrich-Pszona and Barenboim-Elkin algorithms give very similar performance. k -core performs slightly worse on all graphs except for the small email graph. From our experiments, degree ordering results in running times that are comparable to both Goodrich-Pszona and Barenboim-Elkin.

While Goodrich-Pszona, Barenboim-Elkin, and k -core produce arboricity orderings, we may want to use degree ordering as it is much more efficient to compute and can compensate for the potentially worse counting time. Figure 4.7 shows the proportion of time spent on preprocessing (T_p) versus counting (T_c) on different ori-

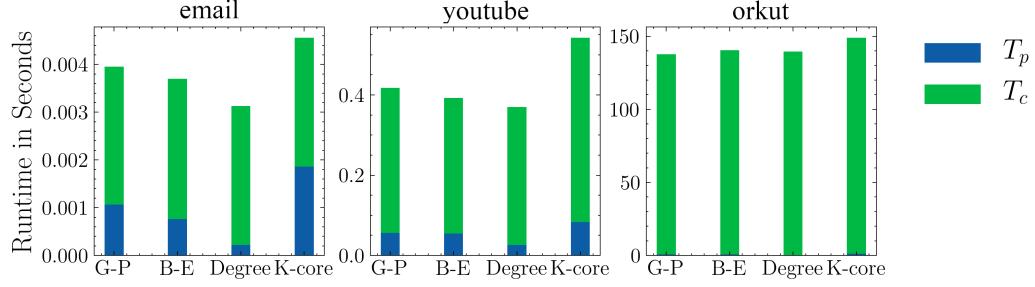


Figure 4.7: Breakdown of time spent in the exact parallel Kowalik implementation on preprocessing (T_p) vs. counting (T_c) for different orientation subroutines, using 36 cores with hyper-threading. G-P is Goodrich-Pszona; B-E is Barenboim-Elkin. For the orkut graph, the time spent on preprocessing is not visible.

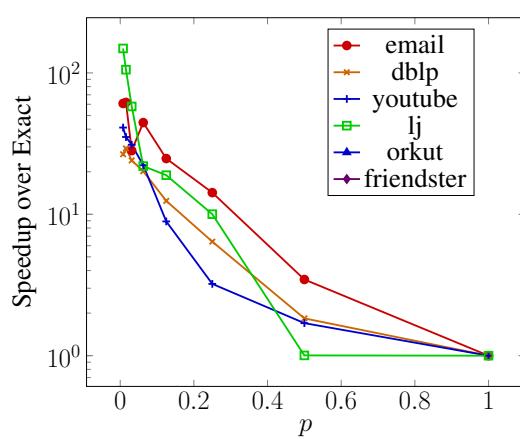


Figure 4.8: These are the parallel speedups over exact running times for approximate five-cycle counting using edge sparsification, varying over different p . The best runtimes considering all orientations were used. The speedups are given in a log-scale.

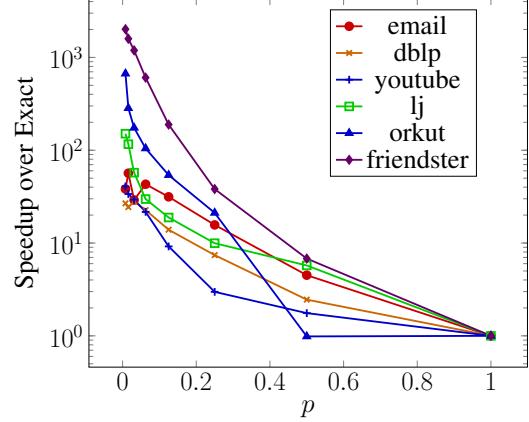


Figure 4.9: These are the parallel speedups over exact running times for approximate five-cycle counting using colorful sparsification, varying over $p = 1/c$, where c is the number of colors used. The best runtimes considering all orientations were used. The speedups are given in a log-scale.

entation methods on three of the graphs. As the graph size grows, the preprocessing time takes up a smaller fraction of the total running time and becomes negligible in the case of the orkut graph. However, for smaller graphs, degree orientation has a clear advantage, because it takes much less time to compute while allowing for similar performance in the counting step. k -core ordering does not perform well when considering the times for both preprocessing and counting.

Approximate Five-Cycle Counting. For our approximate five-cycle counting experiments, we use the parallel Kowalik algorithm on the sparsified graph, as it outperforms the parallel ESCAPE algorithm in the exact setting. Figures 4.8 and

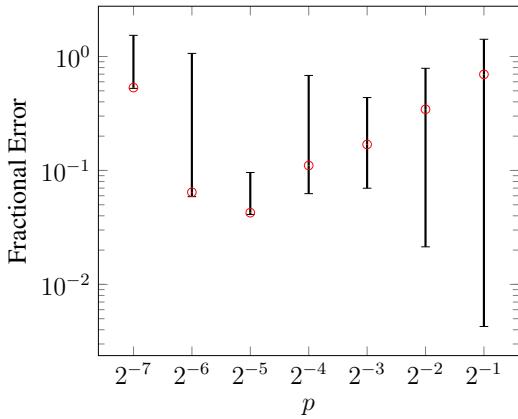


Figure 4.10: These are the fractional errors of the approximate five-cycle count obtained using edge sparsification, varying over different p . The errors are taken over all graphs, with the red circle marking the median error and the bars marking the minimum and maximum errors.

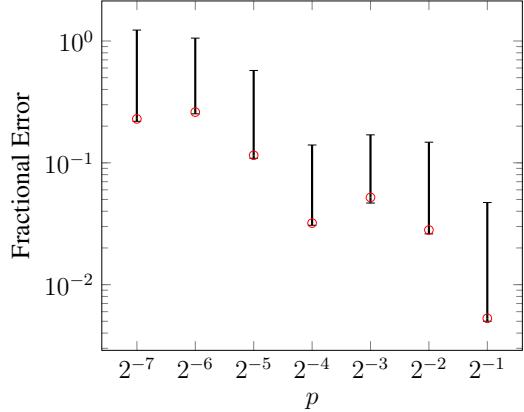


Figure 4.11: These are the fractional errors of the approximate five-cycle count obtained using colorful sparsification, varying over $p = 1/c$, where c is the number of colors used. The errors are taken over all graphs, with the red circle marking the median error and the bars marking the minimum and maximum errors.

4.9 show the speedups of approximate five-cycle counting using edge sparsification and colorful sparsification, respectively, over exact five-cycle counting, considering different probabilities p for preserving edges and different numbers of colors c where $p = 1/c$, respectively. We generally observe higher speedups for smaller p , although for the smallest graphs email and dblp, the speedups degrade once the preserved subgraphs are small enough such that there are few to no five-cycles. Figures 4.10 and 4.11 show the fractional errors of the approximate five-cycle counts obtained using edge sparsification and colorful sparsification respectively, compared to the exact five-cycle counts. We compute our error using the formula $|\text{exact} - \text{approximate}|/\text{exact}$. For $p = 1/4$, across all graphs, we see 3.22–30.30x speedups of approximate five-cycle counting using edge sparsification over exact five-cycle counting, with error rates between 32.26–44.40%. For $p = 1/4$ using colorful sparsification, we see 2.99–38.16x speedups, with error rates between 0.20–11.95%. We observe more significant speedups for smaller p ; in particular, for $p = 1/8$, across all graphs, we see 8.90–113.42x speedups using edge sparsification, with error rates between 9.93–26.70%, and we see 9.10–188.94x speedups using colorful sparsification, with error rates between 0.52–11.77%. For larger p , we note that while $p = 1/2$ reduces the error rate in colorful sparsification to a median of 0.53%, it gives much lower speedups, of up to 4.63x for edge sparsification and up to 6.80x for colorful sparsification, and the variance in the percent error is higher in edge sparsification. Overall, we observe more significant speedups on larger graphs, and we note that colorful sparsification generally produces more accurate five-cycle counts compared to edge sparsification.

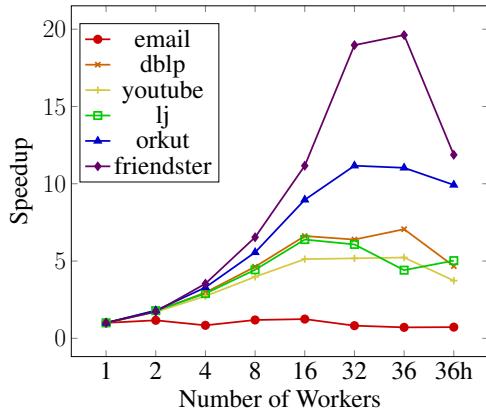


Figure 4.12: These are the self-relative speedups of approximate five-cycle counting over the single-threaded running times, using edge sparsification with $p = 1/8$ over varying numbers of workers. “36h” refers to 36 cores with hyper-threading. The Goodrich-Pszona orientation was used.

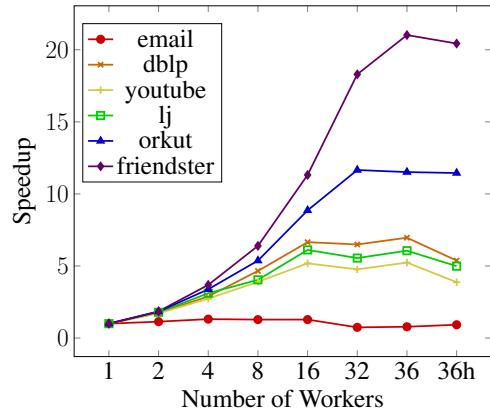


Figure 4.13: These are the self-relative speedups of approximate five-cycle counting over the single-threaded running times, using colorful sparsification with 8 colors ($p = 1/8$) over varying numbers of workers. “36h” refers to 36 cores with hyper-threading. The Goodrich-Pszona orientation was used.

Notably, on friendster, which takes over two hours to complete exact five-cycle counting, we can obtain an approximate count with 0.20% error in under a minute using colorful sparsification, with $p = 1/4$.

Figures 4.12 and 4.13 show the self-relative parallel speedups of approximate five-cycle counting using edge sparsification and colorful sparsification, respectively, over different numbers of workers, where $p = 1/8$ and using the Goodrich-Pszona orientation. We note that very little self-relative speedup is observed on the smaller graphs, particularly the email graph, where the sparsified graph is exceedingly small. The largest self-relative speedups appear on the largest graphs, and on these large graphs, we see good scalability. Overall, we observe up to 11.87x and 20.43x self-relative speedups over the single-threaded running times, using edge sparsification and colorful sparsification, respectively. For these experiments, we see that hyper-threading usually does not help, likely due to the smaller sizes of the graphs compared to the experiments on the exact algorithms.

4.6 Discussion

We designed the first theoretically work-efficient parallel five-cycle counting algorithms with polylogarithmic span. On 36 cores, our best exact implementation outperforms the fastest existing exact serial implementation by up to 818x, and achieves self-relative speedups of 10–46x. Our best approximate implementation, for a fixed

probability parameter $p = 1/8$, achieves up to 20.43x self-relative speedups and is able to approximate five-cycle counts 9.10–188.94x faster than our best exact implementation, with between 0.52–11.77% error. Designing parallel algorithms for counting larger cycles is interesting for future work, although such algorithms are likely to require super-linear work, even for low-arboricity graphs [34].

Chapter 5

k-clique Counting and Listing

5.1 Introduction

Finding k -cliques in a graph is a fundamental graph-theoretic problem with a long history of study both in theory and practice. In recent years, k -clique counting and listing have been widely applied in practice due to their many applications, including in learning network embeddings [270], understanding the structure and formation of networks [341], identifying dense subgraphs for community detection [317, 284, 127, 153], and graph partitioning and compression [131].

Motivated by these applications, the basic problem of counting and listing cliques has received significant attention [175, 86, 322, 192, 117, 245, 176, 133, 255], and state-of-the-art implementations today can count and list all k -clique for large k in graphs with millions to hundreds of millions of edges within a few hours on commodity multicore machines. The fastest of these implementations is from a recent paper by Danisch et al. [86]. They design a parallel algorithm kCLIST and show that it is highly practical and, interestingly, also provably work-efficient, matching the work of the Chiba-Nishizeki algorithm, which even after close to 40 years is the fastest sequential algorithm for the clique listing problem on sparse graphs [72].

Despite this impressive progress, several fundamental open problems remain. Firstly, Danisch et al. [86] do not theoretically bound the span of their kCLIST algorithm. To the best of our knowledge, despite the fundamental nature of k -clique counting and listing, it is not known whether it can be done work-efficiently with respect to the Chiba-Nishizeki algorithm in polylogarithmic span. Secondly, kCLIST requires computing an orientation of the graph's edges that bounds the maximum out-degree to achieve work-efficiency with respect to the Chiba-Nishizeki algorithm. They use the degeneracy ordering, introduced by Matula and Beck [229]. Unfortunately, computing this ordering is known to be P -complete, and thus there is little hope of parallelizing this approach in polylogarithmic span [17]. Thirdly, all of these algorithms have limited scalability for graphs with more than a few hundred million edges, but real-world graphs today frequently contain billions—hundreds of billions of edges [233]. Therefore, an interesting and timely question is the following:

Is there a work-efficient, polylogarithmic-span k -clique finding algorithm that can

scale to massive real-world graphs containing billions to hundreds of billions of edges?

In this chapter, we answer this question affirmatively, and build on this fundamental graph processing primitive to design provably-efficient parallel algorithms solving the k -clique densest subgraph problem [317].

We introduce a new parallel k -clique counting algorithm that is based on using low out-degree orientations of the graph to reduce the total amount of work. Specifically, we generate rankings of the vertices such that for every vertex v , the number of neighbors that rank higher than v is small. Then we direct each edge in the graph from lower-ranked to higher-ranked endpoint, so that each k -clique in the original graph has a canonical directed representation. In particular, the j 'th ranked vertex in the k -clique is directed towards the $k - j - 1$ higher-ranked vertices in the k -clique. Larger cliques can be found from smaller cliques by intersecting the outgoing neighbors of vertices in the clique. Assuming that we have a low out-degree ranking of the graph, we show that for a constant k we can count or list all k -cliques in $O(m\alpha^{k-2})$ work and $O(\log^{k-2} m)$ span, where m is the number of edges in the graph and α is the arboricity of the graph. Theoretically, our counting algorithm requires only $O(\alpha)$ extra space per processor; in contrast, the KCLIST algorithm requires $O(\alpha^2)$ extra space per processor. Our k -clique counting algorithm can easily be extended to support k -clique listing, and more generally k -clique enumeration, where an arbitrary function is applied on each k -clique found. We also present an approximate counting algorithm based on counting cliques on a sparsified graph, and prove that it produces unbiased estimates and runs in $O(pma^{k-2} + m)$ work and $O(\log^{k-2} m)$ span for a sampling probability of p .

Furthermore, we design two new parallel algorithms for ranking the vertices. We show that a distributed algorithm by Barenboim and Elkin [29] can be implemented in linear work and polylogarithmic span. We also parallelize an external-memory algorithm by Goodrich and Pszona [150] and obtain the same complexity bounds. Both algorithms are based on peeling a constant fraction of the vertices on each iteration in parallel. We believe that our parallel ranking algorithms may be of independent interest, as many other subgraph finding algorithms use low out-degree orderings (e.g., [150, 258, 195, 176]).

We also present implementations of our algorithms that use various optimizations, including different intersection methods and varying levels of parallelism, to achieve good practical performance. We perform a thorough experimental study of these algorithms on a 60-core machine with two-way hyper-threading and compare them with prior work. We show that on a variety of real-world graphs and different values of k , our k -clique counting algorithm achieves 1.31–9.88x speedup compared to the state-of-the-art parallel KCLIST algorithm [86], and achieves self-relative speedups of 13.23–38.99x. Evaluating our new parallel ordering algorithms separately, we find that they achieve 6.69–19.82x self-relative speedup on 60 cores with hyper-threading. Furthermore, by integrating state-of-the-art parallel graph compression techniques, our implementation is able to process graphs containing tens—hundreds of billions of edges, significantly improving on the capabilities of existing implementations. As far as we know, we are the first to report 4-clique counts for Hyperlink2012, the largest publicly-available graph, which has two hundred billion undirected edges. We also

study the accuracy-time tradeoff of our sampling algorithm, and show that it is able to approximate the clique counts with 5.05% error 5.32–6573.63 times more quickly than running our exact counting algorithm on the same graph.

We summarize the main contributions of this chapter below:

- (1) A work-efficient parallel algorithm with polylogarithmic span for k -clique counting.
- (2) Work-efficient and polylogarithmic-span parallel algorithms for computing low-outdegree orientations.
- (3) An approximate parallel algorithm using colorful sparsification that can compute unbiased and accurate estimates of the clique counts of a graph much more quickly than exact counting.
- (4) Highly-optimized implementations of our algorithms that achieve significant speedups over existing state-of-the-art methods, and scale to the largest publicly-available graphs, containing billions of vertices and hundreds of billions of edges.

Our code is publicly available at: <https://github.com/ParAlg/gbbs/tree/master/benchmarks/CliqueCounting>.

5.2 Related Work

There has been a large body of work on k -clique counting, listing, and related problems, which we describe here.

Theory. A trivial algorithm can compute all k -cliques in $O(n^{k/2})$ work. Using degree-based thresholding enables counting cliques in time $O(m^{k/2})$, which is asymptotically faster for sparse graphs [14]. Chiba and Nishizeki give an algorithm that further improves the complexity for sparse graphs, showing that all k -cliques can be found in $O(m\alpha^{k-2})$ work [72]. For finding k -cliques, their algorithm processes vertices in non-increasing order of degree, and for each vertex v , recursively finds all $(k-1)$ -cliques incident on the induced subgraph of v 's neighbors, and then deletes v from the graph to prevent cliques containing v from being explored again. Due to the fact that such a graph can have up to $O(m\alpha^{k-2})$ distinct k -cliques, the Chiba-Nisheziki work bound is necessary for k -clique listing, but can be improved upon for dense graphs with large arboricity in the case of k -clique counting. The counting algorithm developed in this paper is built on the ideas of Chiba and Nishizeki.

For arbitrary graphs, the fastest algorithm in theory relies on matrix multiplication, and counts $3l$ cliques in $O(n^{l\omega})$ time where ω is the matrix multiplication exponent [245]. The k -clique problem is one of the canonical hard problems in the FPT literature, and is known to be $W[1]$ -complete when parametrized by k [111]. We refer the reader to the paper of Vassilevska, which surveys other theoretical algorithms for this problem [322].

Practice. The special case of counting and listing triangles ($k = 3$) has received a huge amount of attention over the past two decades (e.g., [289, 22, 319, 232, 318, 251, 250, 292, 39, 304], among many others). Finocchi et al. [133] present parallel k -clique counting algorithms for MapReduce. Jain and Seshadri [175] provide algorithms for estimating k -clique counts. The state-of-the-art k -clique counting and listing

algorithm is the κ CLIST algorithm by Danisch et al. [86]. Their algorithm is based on Chiba and Nishizeki’s algorithm, but using the k -core (degeneracy) ordering to rank vertices instead of a non-decreasing degree ordering. κ CLIST explicitly generates a directed version of the graph based on the ranking, unlike Chiba and Nishizeki. The κ CLIST algorithm is parallelized by either launching a parallel task per vertex to find all directed cliques incident on that vertex (node-parallel) or launching a parallel task per edge to find all directed cliques incident on that edge (edge-parallel). Danisch et al. [86] do not analyze the span of their κ CLIST, and in fact their algorithm cannot have polylogarithmic span since they only parallelize at the top one or two levels of recursion and they use the k -core ordering, which is P-complete to compute.

Additionally, many algorithms have been developed for finding 4-vertex and 5-vertex subgraphs (e.g., [258, 255, 9, 119, 7, 333, 8, 271]) as well as estimating larger subgraph counts (e.g., [59, 60]), and these algorithms can be used for counting exact or approximate k -clique counting as a special case. Worst-case optimal join algorithms from the database literature [4, 247, 234] can also be used for k -clique listing and counting as a special case, and would require $O(m^{k/2})$ work.

Very recently and independently of our work, Jain and Seshadri [176] present a sequential and a vertex parallel PIVOTER algorithm for counting all cliques in a graph. By terminating early, their algorithm can also be used for k -clique counting for fixed k . However, their algorithm cannot be used for clique listing as they avoid processing all cliques, and their algorithm is not work-efficient for fixed k . We compare with their clique counting runtimes for fixed k in Section 5.4.

Low Out-degree Orientations. A canonical technique in the graph algorithms literature on clique counting, listing, and related tasks is the use of a low out-degree orientation (sometimes also referred to as a degeneracy ordering). An l -orientation of an undirected graph is a total ordering on the vertices where the oriented out-degree of each vertex (the number of its neighbors higher than it in the ordering) is bounded by l . A classic method for producing such an ordering is due to Matula and Beck [229], who show that the k -core order gives a $c(G)$ -orientation of the graph, where $c(G)$ is the degeneracy of the graph, which is within a constant factor of α , the graph’s arboricity. However, computing this ordering is a P-complete problem [17]. More recent work in the distributed and external-memory literature has shown that such orderings can be efficiently computed in these settings. Barenboim and Elkin give a distributed algorithm that finds an $O(\alpha)$ -orientation in $O(\log n)$ rounds [29]. Goodrich and Pszona give a similar orientation algorithm that repeatedly peels a constant fraction of the lowest degree vertices, and analyze this algorithm in the external-memory model [150]. Such orientations are widely used in other related problems, such as maximal clique listing [121, 176] and fixed-sized subgraph finding [258, 195].

5.3 Clique Counting

In this section, we present our main algorithms for counting k -cliques in parallel. Our algorithms can be used for the more general task of k -clique enumeration, in which

Algorithm 5.1 – Goodrich-Pszona Orientation

```
1: procedure ORIENT( $G = (V, E)$ ,  $\epsilon$ )
2:    $n \leftarrow |V|$ ,  $L \leftarrow []$ 
3:   while  $G$  is not empty do
4:      $S \leftarrow \epsilon n / (2 + \epsilon)$  vertices of lowest induced degree
5:     Append  $S$  to  $L$ 
6:     Remove vertices in  $S$  from  $G$ 
7:   return  $L$ 
```

Algorithm 5.2 – Barenboim-Elkin Orientation

```
1: procedure ORIENT( $G = (V, E)$ ,  $\epsilon$ ,  $\alpha$ )
2:    $n \leftarrow |V|$ ,  $L \leftarrow []$ 
3:   while  $G$  is not empty do
4:      $S \leftarrow \{v \in V \mid v\text{'s induced degree less than } (2 + \epsilon)\alpha\}$ 
5:     Append  $S$  to  $L$ 
6:     Remove vertices in  $S$  from  $G$ 
7:   return  $L$ 
```

a user-supplied function is applied on each k -clique (this includes global counting, local counting, and listing as special cases). We start in Section 5.3.1 by describing parallel methods for computing low out-degree orderings of the vertices in parallel. Then, in Section 5.3.2, we describe our parallel k -clique listing algorithm. Then, in Section 5.3.3, we describe how this algorithm can be optimized to generate unbiased estimates of the global k -clique count using graph sparsification. Finally, in Section 5.3.4 we discuss practical optimizations that improve our methods’ running time in practice.

5.3.1 Low Out-degree Orientation (Ranking)

Recall that an l -orientation of an undirected graph is a total ordering on the vertices where the oriented out-degree of each vertex (the number of its neighbors higher than it in the ordering) is bounded by l . Although this problem has been widely studied in other contexts, to the best of our knowledge, we are not aware of any previous work-efficient parallel algorithms for solving this problem. Here, we show that the Barenboim-Elkin and Goodrich-Pszona algorithms, which are efficient in the CONGEST and I/O models of computation lead to work-efficient low-span algorithms in the work-span setting.

Algorithm 5.1 shows pseudocode for the Goodrich-Pszona algorithm, and Algorithm 5.2 shows pseudocode for the Barenboim-Elkin algorithm. Both algorithms take as input a user-defined parameter ϵ . The Barenboim-Elkin requires an additional parameter, α , which is the arboricity of the graph (or an estimate of the arboricity). For this parameter, we use a $(2 + \epsilon)$ -approximation to the arboricity, using the approximate densest-subgraph algorithm from [98] which runs in $O(m + n)$ work and $O(\log^2 n)$ span.

Note that both algorithms remove a constant fraction of the vertices in each round.

For the Goodrich-Pszona algorithm, this follows from construction—an $\epsilon/(2 + \epsilon)$ fraction of vertices are removed on each round, and as such the algorithm finishes after $O(\log n)$ rounds. For the Barenboim-Elkin algorithm, by definition of arboricity, there are at most $n\alpha/d$ vertices with degree at least d . Thus, the number of vertices with degree more than $(2 + \epsilon)\alpha$ is at most $n/(2 + \epsilon)$, and a constant fraction of the vertices have degree at most $(2 + \epsilon)\alpha$. Since a subgraph of graph with arboricity α also has arboricity at most α , each round will peel at least a constant fraction of remaining vertices, and the algorithm will terminate in $O(\log n)$ rounds.

The bound on the out-degree for the Barenboim-Elkin algorithm follows by construction, since it only peels vertices with induced degree less than $(2 + \epsilon)\alpha$. The bound for the Goodrich-Pszona algorithm follows due to a similar density argument as the round-complexity argument for the Barenboim-Elkin algorithm. The number of vertices with degree at least $(2 + \epsilon)\alpha$ can be at most $n/(2 + \epsilon)$, and thus the $\epsilon/(2 + \epsilon)$ fraction of the lowest degree vertices must have degree less than $(2 + \epsilon)\alpha$ as desired.

Theorem 5.1. *The Goodrich-Pszona and Barenboim-Elkin algorithms compute orientations with out-degree $O(\alpha)$ in $O(m + n)$ work and $O(\log^2 n)$ span. The bounds are randomized for the Goodrich-Pszona algorithm.*

Proof. The bounds on out-degree follow from the discussion above. For the work and span, we first observe that both algorithms run in $O(\log n)$ rounds, and each round removes a constant fraction of the vertices. For both algorithms, we maintain the induced degrees of all vertices in an array.

For the Goodrich-Pszona algorithm, we can filter out the vertices with degree less than the c -th smallest degree vertex for $c = \epsilon n/(2 + \epsilon)$ using parallel integer sort, which runs in $O(n')$ expected work and $O(\log n)$ span w.h.p. where n' is the number of remaining vertices [265]. For the Barenboim-Elkin algorithm, we use a parallel filter, which takes linear work and $O(\log n)$ span.

We can update the degrees of the remaining vertices after removing the peeled vertices by mapping over all edges incident to these vertices, and applying an atomic add instruction to decrement the degree of each neighbor. Each edge is processed exactly once in each direction, when its corresponding endpoints are peeled, and each vertex is peeled exactly once, so the total work is $O(m + n)$. Since each peeling round can be implemented in $O(\log n)$ span, and there are $O(\log n)$ such rounds, the span of both algorithms is $O(\log^2 n)$.

Finally, computing an estimate of the arboricity using the parallel densest-subgraph algorithm from [98] can be done in $O(m + n)$ work and $O(\log^2 n)$ span, which does not asymptotically increase the cost of running the Barenboim-Elkin algorithm. \square

Finally, in the parallel algorithms in the remainder of this paper that utilize orientation, we will usually direct the input graph in the CSR format after computing an orientation. This can be done in $O(m)$ work and $O(\log n)$ span using prefix sum and filter.

Algorithm 5.3 – Parallel work-efficient k -clique counting

```
1: procedure REC-COUNT-CLIQUE(DG, I, ℓ)
2:   if  $\ell = 1$  then return  $|I|$ 
3:   Initialize  $T$  to store clique counts per vertex in  $I$ 
4:   parfor  $v$  in  $I$  do
5:      $I' \leftarrow \text{INTERSECT}(I, N_{DG}(v))$             $\triangleright$  Intersect  $I$  with directed nbhrs of  $v$ 
6:      $t' \leftarrow \text{REC-COUNT-CLIQUE}(DG, I', \ell - 1)$ 
7:     Store  $t'$  in  $T$ 
8:    $t \leftarrow \text{REDUCE-ADD}(T)$                     $\triangleright$  Sum clique counts in  $T$ 
9:   return  $t$ 

10: procedure COUNT-CLIQUE(G = (V, E), k, ORIENT)
11:    $DG \leftarrow \text{ORIENT}(G)$             $\triangleright$  Apply a user-specified orientation algorithm
12:   return REC-COUNT-CLIQUE(DG, V, k)
```

5.3.2 Counting Algorithm

Our algorithm for k -clique counting is shown as COUNT-CLIQUE in Algorithm 5.3. COUNT-CLIQUE first directs the edges of G such that every vertex has out-degree $O(\alpha)$, as described in Section 5.3.1 (Line 11). Then, it calls a recursive subroutine REC-COUNT-CLIQUE that takes as input the directed graph DG , candidate vertices I that can be added to a clique, and the number of vertices ℓ left to complete a k -clique. With every recursive call to REC-COUNT-CLIQUE, a new candidate vertex v from I is added to the clique and I is pruned to contain only out-neighbors of v (Line 5). REC-COUNT-CLIQUE terminates when precisely one vertex is needed to complete the k -clique, in which the number of vertices in I represents the number of completed k -cliques (Line 2). The counts obtained from recursive calls are aggregated using a REDUCE-ADD and returned (Lines 8–9).

We note that COUNT-CLIQUE and REC-COUNT-CLIQUE can be modified to store k -clique counts per vertex. To do so, we add to COUNT-CLIQUE an array C to store k -clique counts per vertex, and in our recursive subroutine, after obtaining the k -clique counts t' in Line 6, we add t' to the count corresponding to vertex v . In addition, before returning the number of completed k -cliques in Line 2, we also increment the count corresponding to each vertex in I . Note that the updates to C must be done through atomic adds. We call the algorithms for counting k -cliques per vertex COUNT-CLIQUE-V and REC-COUNT-CLIQUE-V respectively.

Similarly, COUNT-CLIQUE and REC-COUNT-CLIQUE can be modified to support enumeration. We add to REC-COUNT-CLIQUE a parameter K that stores the k -clique built so far, and prior to the recursive call to REC-COUNT-CLIQUE on Line 6, we add v to K . Then, prior to returning the number of k -cliques on Line 2, we note that each vertex in I added to K produces a completed k -clique, upon which we can apply a user-supplied function.

Note that aside from the initial call to REC-COUNT-CLIQUE which takes $I = V$ as an input, in subsequent calls, the size of I is bounded by $O(\alpha)$. This is because at every recursive step, I is intersected with the out-neighbors of some vertex v ,

which is bounded by $O(\alpha)$. The additional space required by COUNT-CLIQUEs on a one processor is $O(\alpha)$, and since the space is allocated in a stack-allocated fashion, we can bound the total additional space by $O(P\alpha)$ on P processors when using a work-stealing scheduler [52].

Moreover, considering the first call to REC-COUNT-CLIQUEs, the total expected work of INTERSECT is given by $O(m)$, because the sum of the degrees of each vertex v is bounded above by the number of edges. Also, using parallel adjacency hash table, the expected work of INTERSECT in each subsequent recursive step is given by the minimum of $|I|$ and $|N_{DG}(v)|$, and as such is bounded above by $O(\alpha)$. We recursively call REC-COUNT-CLIQUEs k times as ℓ ranges from 1 to k , but the first recursive call involves a trivial intersect where we simply retrieve all directed neighbors of v , and the final recursive call returns with the size of I immediately. Hence, we have $k - 2$ recursive steps that call INTERSECT in a non-trivial manner, so in total, COUNT-CLIQUEs takes $O(m\alpha^{k-2})$ expected work.

Finally, the span of COUNT-CLIQUEs is defined by the span of INTERSECT and the span of REDUCE-ADD in each recursive call. As discussed in Chapter 2, the span of INTERSECT is given by $O(\log n)$ w.h.p., and the span of REDUCE-ADD is given by $O(\log n)$. Thus, since we have $k - 2$ recursive steps with non-constant span, COUNT-CLIQUEs takes $O(\log^{k-2} n)$ span w.h.p.

Note that COUNT-CLIQUEs-V obtains the same work and span bounds as COUNT-CLIQUEs, since the atomic add operations do not increase the work or span.

The total complexity of k -clique counting is as follows.

Theorem 5.2. *k -clique counting can be performed in $O(m\alpha^{k-2})$ expected work and $O(\log^{k-2} n)$ span w.h.p., using $O(\alpha)$ additional space per processor.*

5.3.3 Sampling

We now discuss a sparsification technique known as colorful sparsification that allows us to produce approximate k -clique counts, based on previous work on approximate triangle and butterfly (biclique) counting [251, 277]. We sparsify our input graph G by coloring each vertex with one of c colors uniformly at random, and preserving edges only if both endpoints have the same color. We call the resulting graph with only preserved edges G' . Let $p = 1/c$. We run our k -clique counting algorithm (Algorithm 5.3) on G' which outputs a count C , and we output $Y = C/p^{k-1}$ as the estimate for the global k -clique count. We prove the following theorem about the properties of this estimator.

Theorem 5.3. *Let X be the true k -clique count in G , C be the k -clique count in G' , $p = 1/c$, and $Y = C/p^{k-1}$. Then $\mathbb{E}[Y] = X$ and $\text{Var}[Y] = p^{-2(k-1)}(X(p^{k-1} - p^{2(k-1)}) + \sum_{z=2}^{k-1} s_z(p^{2(k-1)-z+1} - p^{2(k-1)}))$, where s_z is the number of pairs of k -cliques that share z vertices.*

Proof. Let C_i be an indicator variable denoting whether the i 'th k -clique in G is preserved in G' . For a k -clique to be preserved, all k vertices in the clique must have the same color. This happens with probability p^{k-1} since after fixing the color of

one vertex v in the clique, the remaining $k - 1$ vertices must have the same color as v . Each vertex picks a color independently and uniformly at random, and so the probability of a vertex choosing the same color as v is p . The number of k -cliques in G' is equal to $C = \sum_i C_i$. Therefore $\mathbb{E}[C] = \sum_i \mathbb{E}[C_i] = Xp^{k-1}$. We have that $\mathbb{E}[Y] = \mathbb{E}[C/p^{k-1}] = (1/p^{k-1})\mathbb{E}[C] = (1/p^{k-1})Xp^{k-1} = X$.

The variance of Y is $\text{Var}[Y] = \text{Var}[C/p^{k-1}] = \text{Var}[C]/p^{2(k-1)} = \text{Var}[\sum_i C_i]/p^{2(k-1)}$. By definition $\text{Var}[\sum_i C_i] = \sum_i (\mathbb{E}[C_i] - \mathbb{E}[C_i]^2) + \sum_{i \neq j} \text{Cov}[C_i, C_j]$. The first term is equal to $X(p^{k-1} - p^{2(k-1)})$. To compute the second term, note that $\text{Cov}[C_i, C_j] = \mathbb{E}[C_i C_j] - \mathbb{E}[C_i]\mathbb{E}[C_j]$ depends on the number of vertices that cliques i and j share. Their covariance is 0 if they share no vertices. Their covariance is also 0 if they share one vertex since the event that the remaining vertices of each clique have the same color as the shared vertex is independent between the two cliques. Let s_z denote the number of pairs of cliques that share $z > 1$ vertices. For pairs of cliques sharing z vertices, we have $\mathbb{E}[C_i C_j] - \mathbb{E}[C_i]\mathbb{E}[C_j] = p^{2(k-1)-z+1} - p^{2(k-1)}$. This is because given z shared vertices with the same color, the probability that the remaining $2(k-1) - z$ vertices across the two cliques have the same color as the shared vertices is $p^{2(k-1)-z+1}$. Therefore, $\text{Var}[Y] = p^{-2(k-1)}(X(p^{k-1} - p^{2(k-1)}) + \sum_{z=2}^{k-1} s_z(p^{2(k-1)-z+1} - p^{2(k-1)}))$. \square

Also, we obtain the following theorem about the complexity of our sampling algorithm.

Theorem 5.4. *Our sampling algorithm with parameters $p = 1/c$ gives an unbiased estimate of the global k -clique count and takes $O(pma^{k-2} + m)$ expected work and $O(\log^{k-2} n)$ span w.h.p.*

Proof. Theorem 5.3 states that our algorithm produces an unbiased estimate of the global count. We now analyze the work and span of the algorithm. Choosing colors for the vertices can be done in $O(n)$ work and $O(1)$ span. Creating a subgraph containing edges with endpoints having the same color can be done using prefix sum and filtering in $O(m)$ work and $O(\log m)$ span. Each edge is kept with probability p as the two endpoints will have matching colors with this probability. Therefore, our subgraph has pm edges in expectation. The arboricity of our subgraph is upper bounded by the arboricity of our original graph, α , and so running our k -clique counting algorithm on the subgraph takes $O(pma^{k-2} + m)$ expected work and $O(\log^{k-2} m)$ span w.h.p. \square

5.3.4 Practical Optimizations

We introduce here practical optimizations that offer tradeoffs between performance and space complexity.

First, in the initial call to REC-COUNT-CLIQUEs, for each vertex $v \in I = V$, we construct the induced subgraph on $N_{DG}(v)$ and replace DG with said induced subgraph for subsequent levels of recursion. This allows subsequent levels of recursion to skip processing edges that have already been pruned as a result of the first level of recursion. Note that because the out-degree of each vertex v is bounded above by α , we use $O(\alpha^2)$ extra space per processor to store these induced subgraphs.

Moreover, as mentioned in Chapter 2, we store our graphs (and induced subgraphs) in compressed sparse row format in practice. As such, in order to efficiently intersect the candidate vertices in I with the requisite out-neighbors, we relabel the vertices in the induced subgraph constructed in the second level of recursion to be in the range 0 to α , and then we use an array of size α to mark vertices in I . For each vertex v in I , we simply check if the out-neighbors of v are marked in our array to perform INTERSECT.

While this would require $O(k\alpha)$ extra space per processor to maintain an array of size α per recursive call, we find that in practice, parallelizing only the first or first two recursive levels is sufficient. Past the first two levels, subsequent recursive calls are sequential, and we can reuse the aforementioned array between recursive calls by using the labeling scheme from Chiba and Nishizeki's serial k -clique counting algorithm [72]. We record ℓ in our array for each vertex v in I , perform INTERSECT by checking if the out-neighbors have been marked in the array with ℓ , and then reset the marks to $\ell + 1$ before returning. As such, marks are preserved for INTERSECT operations during the same recursive call. This allows us to use only $O(\alpha)$ extra space per processor to perform INTERSECT operations.

We note that in our implementation, *node parallelism* refers to parallelizing only the first level of recursion and *edge parallelism* refers to parallelizing only the first two levels of recursion. These correspond with the ideas of node and edge parallelism in Danisch et al.'s KCLIST algorithm [86].

Finally, in order to perform the intersections on the second level of recursion (the first set of non-trivial intersections), it is in practice faster to use an array of size n to mark vertices in $N(v) = I$, and perform a constant-time lookup to determine which out-neighbors of $u \in I$ are also in I . Past the second level of recursion, we relabel vertices in the induced subgraph as mentioned previously, and only require the size α array to perform intersections. Thus, we use linear extra space per processor for the second level of recursion only.

In total, the space complexity for intersecting in the second level of recursion and storing the induced subgraph on $N_{DG}(v)$ dominates, so we use $O(\max(n, \alpha^2))$ extra space per processor.

5.4 Experiments

Environment. We run most of our experiments on a c2-standard-60 Google Cloud instance, which consist of 60 cores (with two-way hyper-threading), with 3.8GHz Intel Xeon Scalable (Cascade Lake) processors and 240 GiB of main memory. For our large compressed graphs, we instead use a m1-ultramem-160 Google Cloud instance, which consists of 80 cores (with two-way hyper-threading), with 2.6GHz Intel Xeon E7 (Broadwell E7) processors and 3844 GiB of main memory. We use OpenMP for our k -clique counting runtimes (approximate and exact). Finally, we compile our programs with g++ (version 7.3.1) using the -O3 flag. We terminate any experiment that takes over 5 hours, except for experiments on compressed graphs.

We test our algorithms on real-world undirected graphs from the Stanford Net-

| | n | m | $k = 4$ | $k = 5$ | $k = 6$ | $k = 7$ | $k = 8$ | $k = 9$ | $k = 10$ | $k = 11$ |
|-----------------------|---------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|------------------------|
| as-skitter | 1,696,415 | 11,095,298 | $148,834,439$ | 1.184×10^9 | 9.759×10^9 | 7.314×10^{10} | 4.816×10^{11} | 2.782×10^{12} | 1.422×10^{13} | 6.498×10^{13} |
| com-dblp | 317,080 | 1,049,866 | 16,713,192 | 262,663,639 | 4.222×10^9 | 6.091×10^{10} | 7.772×10^{11} | 8.813×10^{12} | 8.956×10^{13} | — |
| com-orkat | 3,072,441 | 117,185,083 | 3.222×10^9 | 1.577×10^{10} | 7.525×10^{10} | 3.540×10^{11} | 1.633×10^{12} | 7.248×10^{12} | 3.029×10^{13} | 1.171×10^{14} |
| com-friendster | 65,608,366 | 1.806×10^9 | 8.964×10^9 | 2.171×10^{10} | 5.993×10^{10} | 2.969×10^{11} | 3.120×10^{12} | 4.003×10^{13} | 4.871×10^{14} | — |
| com-lj | 3,997,962 | 34,681,189 | 5.217×10^9 | 2.464×10^{11} | 1.099×10^{13} | 4.490×10^{14} | — | — | — | — |
| ClueWeb | 978,408,098 | 7.474×10^{10} | 2.968×10^{14} | — | — | — | — | — | — | — |
| Hyperlink2014 | 1.725×10^9 | 1.241×10^{11} | 7.500×10^{14} | — | — | — | — | — | — | — |
| Hyperlink2012 | 3.564×10^9 | 2.258×10^{11} | 7.306×10^{15} | — | — | — | — | — | — | — |

Table 5.1: Number of vertices, number of undirected edges (once in each direction), and total k -clique counts for our input graphs. We do not have statistics for certain graphs for large values of k , because algorithm did not terminate in under 5 hours; these entries are represented by a dash.

work Analysis Project (SNAP) [207], namely autonomous systems by Skitter (as-skitter), DBLP communities (com-dblp), Orkut communities (com-orkut), Friendster communities (com-friendster), and LiveJournal communities (com-lj). Also, we test our algorithms on ClueWeb, a Web graph from CMU’s Lemur project [54], and on Hyperlink2012 and Hyperlink2014, hyperlink graphs from the WebDataCommons dataset [233]. Note that our large graphs ClueWeb, Hyperlink2012, and Hyperlink2014 are symmetrized to be undirected graphs, and stored and read in a compressed format from the Graph-Based Benchmark Suite (GBBS) [98]. More precisely, we compress the edges for each vertex using byte codes that can be decoded in parallel [302]. Table 5.1 describes the sizes and k -clique counts for these graphs.

Also, for our counting algorithms, we test different orientations, including the Goodrich-Pszona and Barenboim-Elkin orientations discussed in Section 5.3.1, both with $\varepsilon = 1$. We additionally test other orientations that do not give work-efficient and polylogarithmic-span bounds for counting, but are fast in practice, including the orientation given by ranking vertices by increasing degree and directing low-degree vertices to high-degree vertices, the orientation given by k -core ordering, and the orientation given by the original ordering of vertices in the graph as given.

Moreover, we compare our algorithms against Danisch et al.’s KCLIST algorithm [86], which contains the state-of-the-art parallel and sequential k -clique listing implementations. We include a simple modification to their k -clique listing code to support faster k -clique counting; we forego the final iteration over completed k -cliques and simply return the number of these k -cliques, to be added to a total clique count. Note that KCLIST also offers the option of node or edge parallelism, but only offers a k -core ordering to orient the input graphs.

Counting results. Table 5.2 shows the best parallel and sequential runtimes for k -clique counting over the SNAP datasets, from our implementation and from KCLIST, considering different orientations for our implementation, and considering node versus edge parallelism for both implementations. Figure 5.3 shows parallel runtimes for 4-clique counting over the large compressed graphs; note that KCLIST cannot handle such large graphs.

Overall, we obtain between 1.31–9.88x speedups over KCLIST’s best parallel runtimes. Our largest speedups are seen in the larger graphs (notably com-friendster) and for smaller k values, because we obtain more parallelism proportionally to the necessary work. Comparing our parallel runtimes to KCLIST’s serial runtimes, we obtain between 2.26–79.20x speedups, and considering only our parallel runtimes over 0.7 seconds, we obtain between 16.32–79.20x speedups. In fact, by virtue of our orientations, our single-threaded running times are often faster than KCLIST’s serial runtimes, with up to 2.26x speedups particularly for larger graphs and large values of k . Our self-relative speedups are 13.23–38.99x. Finally, we obtain 4-clique counts on the largest publicly-available graphs in under 45 hours, processing the ClueWeb graph with 74 billion edges in under 2 hours, the Hyperlink2014 graph with over one hundred billion edges in under 4 hours, the Hyperlink2012 graph with over two hundred billion edges in under 45 hours. As far as we know, these are the first results

| | | $k = 4$ | $k = 5$ | $k = 6$ | $k = 7$ | $k = 8$ | $k = 9$ | $k = 10$ | $k = 11$ |
|----------------|-------------------------|---------------|---------------|---------------------|-------------------------|---------------------------|----------------------------|------------------------------|-----------------------------|
| as-skitter | COUNT-CLIQUEST T_{60} | 0.60 | 0.67 | 1.24 | 5.73 | 28.38^e | 158.45^e | 854.87^e | 4158.53^e |
| | COUNT-CLIQUEST T_1 | 9.52 | 11.55 | 28.5 | 154.33 | 993.79 | 5952.51 | > 5 hrs | > 5 hrs |
| | KCLIST T_{60} | 1.34 | 1.51 | 2.51 | 8.79 ^e | 49.10 ^e | 289.28 ^e | 1571.92 ^e | 7876.08 ^e |
| | KCLIST T_1 | 3.75 | 9.54 | 47.07 | 304.6 | 2247.52 | 11053.64 | > 5 hrs | > 5 hrs |
| com-dblp | COUNT-CLIQUEST T_{60} | 0.10 | 0.13 | 0.30 | 2.05^e | 24.06^e | 281.39^e | 2981.74^{*e} | > 5 hrs |
| | COUNT-CLIQUEST T_1 | 1.57 | 1.71 | 5.58 | 64.27 | 837.82 | 9913.01 | > 5 hrs | > 5 hrs |
| | KCLIST T_{60} | 0.16 | 0.17 | 0.43 ^e | 4.28 ^e | 55.78 ^e | 640.48 ^e | 6895.16 ^e | > 5 hrs |
| | KCLIST T_1 | 0.23 | 0.81 | 9.80 | 141.56 | 1858.63 | > 5 hrs | > 5 hrs | > 5 hrs |
| com-orkut | COUNT-CLIQUEST T_{60} | 3.10 | 4.94 | 12.57 | 42.09 | 150.87^o | 584.39^o | 2315.89^o | 8843.51^{oe} |
| | COUNT-CLIQUEST T_1 | 79.62 | 158.74 | 452.47 | 1571.49 | 5882.83 | > 5 hrs | > 5 hrs | > 5 hrs |
| | KCLIST T_{60} | 25.27 | 27.40 | 42.23 | 91.67 ^e | 293.92 ^e | 1147.50 ^e | 4666.03 ^e | > 5 hrs |
| | KCLIST T_1 | 106.42 | 252.54 | 813.13 | 3045.5 | 11928.56 | > 5 hrs | > 5 hrs | > 5 hrs |
| com-friendster | COUNT-CLIQUEST T_{60} | 109.46 | 111.75 | 115.52 | 139.98 | 300.62 | 1796.12^e | 16836.41^{oe} | > 5 hrs |
| | COUNT-CLIQUEST T_1 | 2127.79 | 2328.48 | 2723.53 | 3815.24 | 8165.76 | > 5 hrs | > 5 hrs | > 5 hrs |
| | KCLIST T_{60} | 1079.22 | 1104.28 | 1117.31 | 1162.84 | 1576.61 ^e | 4449.81 ^e | > 5 hrs | > 5 hrs |
| | KCLIST T_1 | 2529.27 | 2929.63 | 3730.37 | 5689.11 | 15954.01 | > 5 hrs | > 5 hrs | > 5 hrs |
| com-li | COUNT-CLIQUEST T_{60} | 1.77 | 7.52 | 258.46 | 10733.21 | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs |
| | COUNT-CLIQUEST T_1 | 33.04 | 231.15 | 8956.53 | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs |
| | KCLIST T_{60} | 7.53 | 22.13 | 647.77 ^e | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs |
| | KCLIST T_1 | 28.9 | 517.6 | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs |

Table 5.2: Best runtimes in seconds for our parallel (T_{60}) and single-threaded (T_1) k -clique counting algorithm (COUNT-CLIQUEST), as well as the best parallel and sequential runtimes from KCLIST [86]. The fastest runtimes for each experiment are bolded and in green. All runtimes are from tests in the same computing environment, and include time spent preprocessing and counting (but not time spent loading the graph). For our parallel runtimes and KCLIST, we have chosen the fastest orientations and choice between node and edge parallelism per experiment, while for our serial runtimes, we have fixed the orientation given by degree ordering. For the parallel runtimes from COUNT-CLIQUEST, we have noted the orientation used; ^o refers to the Goodrich-Pszona orientation, ^{*} refers to the orientation given by k -core, and no superscript refers to the orientation given by degree ordering. For both implementations we have noted whether node or edge parallelism was used; ^e refers to edge parallelism, and no superscript refers to node parallelism.

| | $k = 4$ |
|----------------------|-----------|
| Clueweb | 5824.76 |
| Hyperlink2014 | 12945.25 |
| Hyperlink2012 | 161418.89 |

Table 5.3: The parallel runtimes in seconds for our 4-clique counting algorithm (COUNT-CLIQUEs) with degree ordering and node parallelism on large compressed graphs, using 160 cores with hyper-threading.

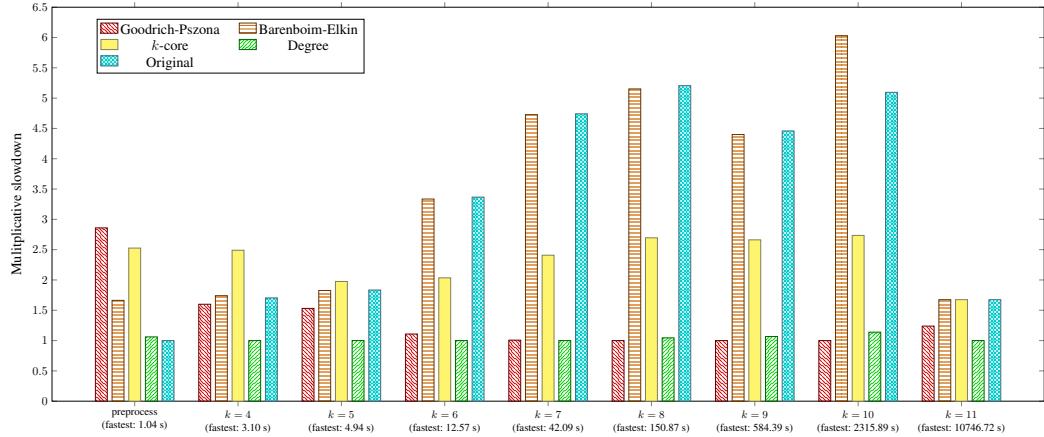


Figure 5.1: Parallel runtimes for k -clique counting (COUNT-CLIQUEs) on com-orkut, considering different orientations using node parallelism. Note that all times are scaled by the fastest parallel runtime, as indicated in parentheses, and the first set of bars indicate the preprocessing overhead of the different orientations. The rest of the k -clique counting runtimes include time spent preprocessing and counting.

for 4-clique counting for graphs of this scale.

We note that of the different orientations, using degree ordering is generally the fastest for small k because it requires almost no preprocessing overhead and results in sufficiently low out-degrees. However, for larger k , this overhead becomes less significant and other orientations, notably the Goodrich-Pszona and k -core orientations, result in faster counting. Figure 5.1 shows the preprocessing overheads and total counting runtimes for com-orkut using different orientations, fixing node parallelism. In this case, the Goodrich-Pszona orientation is 2.86x slower than the orientation using degree ordering, but this overhead is not significant for large k and the Goodrich-Pszona orientation produces the fastest counting runtimes for large of k . We also found that the self-relative speedups of orienting the graph alone were between 6.69–19.82x across all orientations, the larger of which were found in larger graphs.

Moreover, in both COUNT-CLIQUEs and kCLIST, node parallelism is faster on small k , while edge parallelism is faster on large k . This is because parallelizing the first level of recursion is sufficient for small k , and edge parallelism introduces greater parallel overhead. Figure 5.2 shows this behavior in COUNT-CLIQUEs’s k -

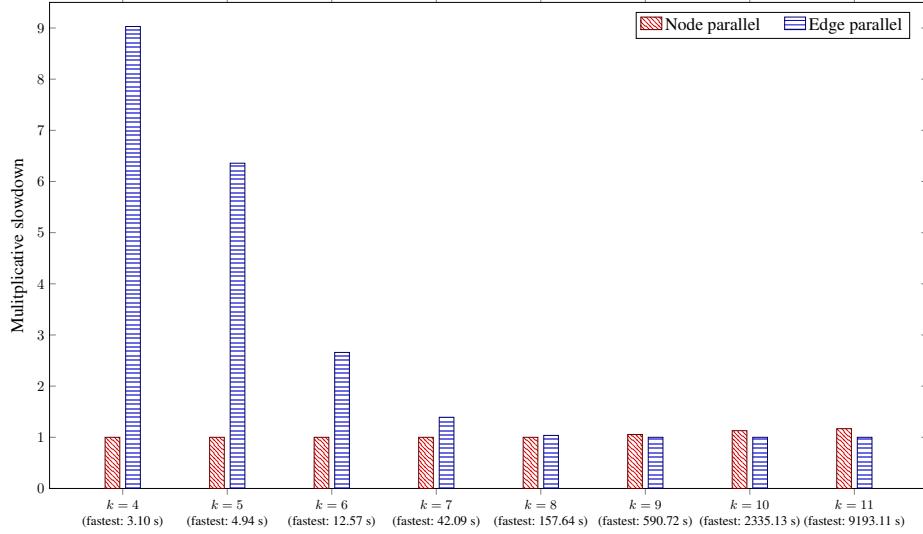


Figure 5.2: These are the parallel runtimes for k -clique counting (COUNT-CLIQUEs) on com-orkut, considering node parallelism and edge parallelism, and fixing the orientation given by degree ordering. Note that all times are scaled by the fastest parallel runtime, as indicated in parentheses.

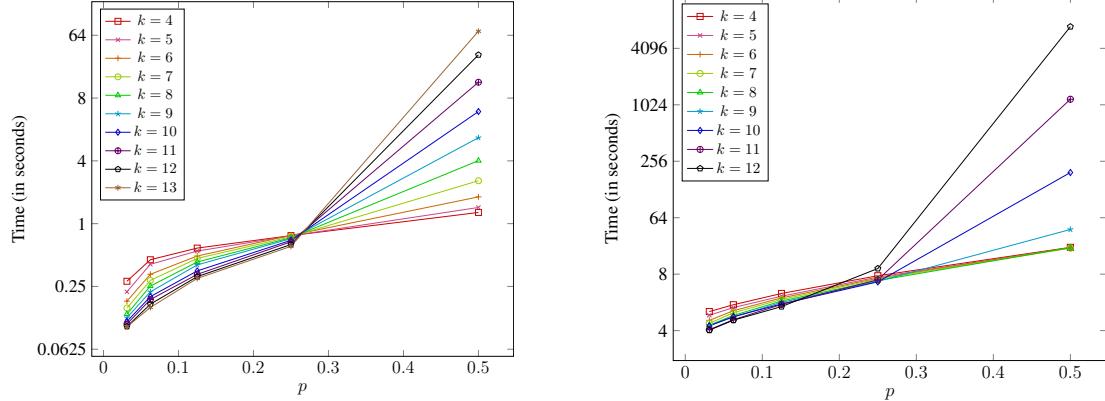


Figure 5.3: These are the parallel runtimes for approximate k -clique counting (COUNT-CLIQUEs) using colorful sparsification on com-orkut, varying over $p = 1/c$ where c is the number of colors used. Note that these runtimes were obtained using the orientation given by degree ordering and node parallelism. The runtimes are given in a log-scale.

Figure 5.4: These are the parallel runtimes for approximate k -clique counting (COUNT-CLIQUEs) using colorful sparsification on com-friendster, varying over $p = 1/c$ where c is the number of colors used. Note that these runtimes were obtained using the orientation given by degree ordering and node parallelism. The runtimes are given in a log-scale.

clique counting runtimes on com-orkut, where for $k \geq 9$ edge parallelism becomes

faster than node parallelism.

We tested Jain and Seshadhri’s [176] serial PIVOTER for k -clique counting. Note that their algorithm cannot support enumeration or listing, and uses compression to efficiently count all cliques, stopping early if only k -cliques for a fixed k are desired. Their algorithm has fast runtimes for counting all cliques, and is able to count all cliques for as-skitter, com-dblp, and com-orkut, in under 5 hours.

However, their algorithm is not theoretically-efficient for fixed k and as such is up to 3014.50x slower compared to parallel COUNT-CLIQUEs and up to 184.76x slower compared to single-threaded COUNT-CLIQUEs for small k , particularly on com-lj and com-orkut. Furthermore, PIVOTER requires too much space to build its recursive tree and runs out of memory for large graphs; it is unable to compute k -clique counts at all for $k \geq 4$ on com-friendster. We note that Jain and Seshadhri developed a parallel version of PIVOTER, but we were not able to obtain this code to test.

Approximate counting results. Figures 5.3 and 5.4 show runtimes for colorful sparsification on com-orkut and com-friendster respectively. We see that in both graphs, there is an inflection point for which after enough sparsification, obtaining k -clique counts for large k is faster than for small k ; this is because we cut off the recursion when it becomes clear there are not enough vertices to complete a k -clique. Moreover, we obtain significant speedups over exact k -clique counting through sparsification and have low error percentages compared to the exact global counts. For $p = 0.5$ on both com-orkut and com-friendster across all k , we see between 2.42–473.63x speedups over exact counting and between 0.39–1.85% error. Our error percentages degrade for higher k and lower p , but even up to $p = 0.125$, we obtain between 5.32–6573.63x speedups over exact counting and between 0.42–5.05% error.

5.5 Discussion

We have presented new work-efficient parallel algorithms for k -clique counting with low span. We have shown experimentally that our implementations achieve good parallel speedups and significantly outperform state-of-the-art implementations.

Chapter 6

Batch-Dynamic k -core Decomposition and Clique Counting

6.1 Introduction

One of the key challenges in discovering the structure of large-scale networks is to detect communities in which individuals (or vertices) have close ties with one another, and to understand how well-connected a particular individual is to the community. The well-connectedness of a vertex or a group of vertices is naturally captured by the concept of a k -core or, more generally, the k -core decomposition; hence, this particular problem and its variants have been widely studied in the machine learning [15, 124, 144], database [75, 208, 122, 55, 231], social network analysis and graph analytics [189, 185, 95, 98], computational biology [77, 191, 217, 226], and other communities [140, 189, 223, 280].

Given an undirected graph G , with n vertices and m edges, the k -core of the graph is the maximal subgraph $H \subseteq G$ such that the induced degree of every vertex in H is at least k . The k -core decomposition of the graph is defined as a partition of the graph into layers such that a vertex v is in layer k if it belongs to a k -core but not a $(k+1)$ -core; this value is known as the *coreness* of the vertex, and the coreness values induce a natural hierarchical clustering. Classic algorithms for k -core decomposition are inherently sequential. A well-known algorithm for finding the decomposition is to iteratively select and remove all vertices v with smallest degree from the graph until the graph is empty [229]. Unfortunately, the length of the sequential dependencies, or the *span*, of such a process can be $\Omega(n)$ given a graph with n vertices. As k -core decomposition is a P-complete problem [17], it is unlikely to have a parallel algorithm with polylogarithmic span. To obtain parallel methods with $\text{poly}(\log n)$ span, we relax the condition of obtaining an *exact* decomposition to one of obtaining a close *approximate* decomposition.

Previous works studied approximate k -core decompositions as a way for obtaining faster and more scalable algorithms in larger graphs than in exact settings [65, 144, 310, 124, 75]. Approximate coreness values are useful for applications where existing methods are already approximate, such as diffusion protocols in epidemiological stud-

ies [77, 191, 217, 226], community detection and network centrality measures [110, 125, 162, 235, 327, 346], network visualization and modeling [15, 63, 340, 347], protein interactions [16, 24], and clustering [146, 205]. Furthermore, due to the rapidly changing nature of today’s large networks, many recent studies have focused on the *dynamic* setting, where edges and vertices can be inserted and deleted, and the k -core decomposition is computed in real time. There has been significant interest in obtaining fast and practical dynamic, approximate and exact k -core algorithms. Dynamic algorithms have been developed for both the sequential [209, 279, 349, 337, 209, 310, 214] and parallel [169, 184, 18] settings. There has also been interest in the closely-related dynamic k -truss problem [171, 11, 351, 224]. However, to the best of our knowledge, *there are no existing parallel batch-dynamic k -core algorithms with provable polylogarithmic span*, which our algorithm achieves.

This chapter focuses on the *batch-dynamic* setting where updates are performed over a batch of *multiple* edge updates applied simultaneously. Such a setting is conducive to parallelization, which we leverage to obtain scalable algorithms. We provide a work-efficient batch-dynamic approximate k -core decomposition algorithm based on a parallel level data structure that we design. We implement our algorithm and show experimentally that it performs favorably compared to the state-of-the-art. Furthermore, we show that our parallel level data structure can be used to obtain work-efficient parallel batch-dynamic algorithms for several other problems, specifically, low out-degree orientation and k -clique coloring. In our paper [216], we show that our data structure is additionally applicable to maximal matching and vertex coloring.

We introduce the necessary definitions in Section 6.2 before giving a technical overview of our results in Section 6.3. Section 6.5 presents our parallel level data structure and k -core decomposition algorithm in more detail. Section 6.6 presents experimental results. Section 6.7 gives our parallel, static, approximate algorithm for k -core decomposition. Finally, Section 6.8 gives our low out-degree framework for our k -clique counting (Section 6.9) results.

6.2 Preliminaries

Definition 2.2 defines an *exact k -core* decomposition. We let $k(v)$ denote the *coreness* of a vertex v . A *c -approximate k -core* decomposition is defined as follows.

Definition 6.1 (*c -Approximate k -Core Decomposition*). *A c -approximate k -core decomposition is a partition of vertices into layers such that a vertex v is in layer k' only if $\frac{k(v)}{c} \leq k' \leq ck(v)$, where $k(v)$ is the coreness of v .*

We let $\hat{k}(v)$ denote the *estimate* of v ’s coreness. Fig. 6.1 shows an example of a k -core decomposition and a $(3/2)$ -approximate k -core decomposition.

Our parallel algorithms operate in the batch-dynamic setting. A *batch-dynamic* algorithm processes updates (vertex or edge insertions/deletions) in batches \mathcal{B} of size $|\mathcal{B}|$. For simplicity, since we can reprocess the graph using an efficient parallel static algorithm when $|\mathcal{B}| \geq m$, we consider $1 \leq |\mathcal{B}| < m$ for our bounds.

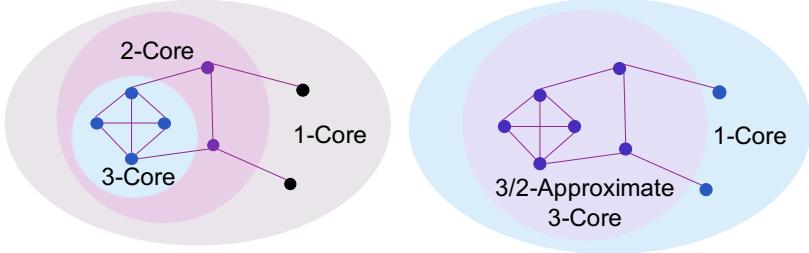


Figure 6.1: Exact k -core decomposition (left) and $(3/2)$ -approximate k -core decomposition (right).

| Symbol | Meaning |
|--------------------------------|--|
| $G = (V, E)$ | undirected/unweighted graph |
| n, m | number of vertices, edges, resp. |
| α | current arboricity of graph |
| Δ | current maximum degree of graph |
| K | number of levels in PLDS |
| $N(v)$ (resp. $N(S)$) | set of neighbors of vertex v (resp. vertices S) |
| $\text{dl}(v)$ | <i>desire-level</i> of vertex v |
| $\ell, \ell(v)$ | a level (starting with level $\ell = 0$), current level of vertex v , resp. |
| V_ℓ, Z_ℓ | set of vertices in level ℓ , set of vertices in levels $\geq \ell$, resp. |
| g_i | set of levels in group i (starting with g_0) |
| $g(v), gn(\ell)$ | <i>group number</i> of vertex v , index i where level $\ell \in g_i$, resp. |
| $k(v), \hat{k}(v)$ | coreness of v , estimate of the coreness of v , resp. |
| $\text{up}(v), \text{up}^*(v)$ | <i>up-degree</i> of v , <i>up*-degree</i> of v , resp. |
| $\varepsilon, \lambda, \delta$ | constants where $\varepsilon, \lambda, \delta > 0$ |

Table 6.1: Table of notations used in this chapter.

Given a graph $G = (V, E)$ and a sequence of batches of edge insertions and deletions, $\mathcal{B}_1, \dots, \mathcal{B}_N$, where $\mathcal{B}_i = (E_{\text{delete}}^i, E_{\text{insert}}^i)$, the goal is to efficiently maintain a $(2 + \varepsilon)$ -approximate k -core decomposition (for any constant $\varepsilon > 0$) after applying each batch \mathcal{B}_i (in order) on G . In other words, let $G_i = (V, E_i)$ be the graph after applying batches $\mathcal{B}_1, \dots, \mathcal{B}_i$ and suppose that we have a $(2 + \varepsilon)$ -approximate k -core decomposition on G_i ; then, for \mathcal{B}_{i+1} , our goal is to efficiently find a $(2 + \varepsilon)$ -approximate k -core decomposition of $G_{i+1} = (V, (E_i \cup E_{\text{insert}}^{i+1}) \setminus E_{\text{delete}}^{i+1})$.

All notations used are summarized in Table 6.1. Our data structure also maintains a *low out-degree orientation*, which may be parameterized by the *arboricity*.

Definition 6.2 (c -Approximate Low Out-Degree Orientation). *Given an undirected graph $G = (V, E)$, a c -approximate low out-degree orientation is an acyclic orientation of all edges in G such that the maximum out-degree of any vertex, d_{\max}^+ , is within a c -factor of the minimum possible maximum out-degree, d_{opt}^+ of any acyclic orientation.¹*

¹ d_{opt}^+ is equal to the *degeneracy*, d , of G , and is closely related to α : $d/2 \leq \alpha \leq d$.

$$d_{opt}^+ / c \leq d_{max}^+ \leq c \cdot d_{opt}^+.$$

For an oriented graph, we call neighbors of vertex v connected by outgoing edges the ***out-neighbors*** of v and neighbors of v connected by incoming edges the ***in-neighbors*** of v .

6.3 Technical Overview

In this chapter, we provide a number of parallel work-efficient algorithms for various problems. This section gives an overview of our algorithms and how they compare with prior work. Table 6.2 summarizes our algorithmic results.

We first discuss k -core decomposition. A number of previous works [212, 221, 279, 352, 351] provided methods for maintaining the *exact* k -core decomposition under single edge updates in the sequential setting. Unfortunately, none of these works provide algorithms with provable polylogarithmic update time. The main bottleneck for obtaining *provably-efficient* methods is that a single edge update can cause *all* coreness values to change: consider a cycle with one edge removed as a simple example. Removing and adding the edge into this cycle, repeatedly in succession, causes the coreness of all vertices to change by one with each update. In the parallel setting, a number of previous works [169, 18, 332, 184, 139] investigated batch-dynamic algorithms for exact k -core decomposition. Unfortunately, none of these works have $\text{poly}(\log n)$ span and some even have $\Omega(n)$ span.

This chapter shows that we can surprisingly obtain a parallel batch-dynamic k -core decomposition algorithm with amortized time bounds that are independent of the number of vertices that *changed coreness* for *approximate* coreness. Such provable time bounds can be obtained by cleverly avoiding updating coreness values until enough error has accumulated; once such error has accumulated, we can charge the amount of time required to update the coreness to the number of updates that occurred. Doing so carefully allows a provable $O(\log^2 n)$ amortized work per update that is independent of the number of changed coreness values. A recent paper by Sun et al. [310] provides a *sequential* dynamic approximate k -core decomposition algorithm that takes $O(\log^2 n)$ amortized time per update. Their algorithm is a threshold peeling/elimination procedure that gives a $(2 + \varepsilon)$ -approximation bound. They also provide another sequential algorithm, which they call *round-indexing*, that performs faster in practice.² However, they do not provide formal runtime proofs for this algorithm. Their threshold peeling algorithm is inherently sequential since a vertex that changes thresholds can cause another to change their threshold (and coreness estimate), resulting in a long chain of sequential dependencies; such a situation results in polylogarithmic *amortized* span, whereas efficient parallel algorithms require polylogarithmic span w.h.p.n the *worst case*, which we obtain.

To design our k -core decomposition algorithm, we formulate a *parallel level data structure (PLDS)* inspired by the sequential level data structures (LDS) of Bhattacharya et al. [42] and Henzinger et al. [164] to maintain a partition of the vertices

²Our experiments compare against the round-indexing algorithm since it is faster than their thresholding peeling algorithm in practice.

| Problem | Approx | Work | Span | Adversary |
|-------------|----------------------|--|-------------------------------------|-----------|
| k -core | $(2 + \varepsilon)$ | $O(\mathcal{B} \log^2 n)$ | $\tilde{O}(\log^2 n)^3$ | Adaptive |
| k -core | $(2 + \varepsilon)$ | $O(m + n)$ | $\tilde{O}(\log^3 n)$ | Static |
| Orientation | $(4 + \varepsilon)$ | $O(\mathcal{B} \log^2 n)$ | $\tilde{O}(\log^2 n)$ | Adaptive |
| Matching | Maximal | $O(\mathcal{B} (\alpha + \log^2 n))$ | $\tilde{O}(\log \Delta \log^2 n)^6$ | Adaptive |
| k -clique | Exact | $O(\mathcal{B} \alpha^{k-2} \log^2 n)$ | $\tilde{O}(\log^2 n)$ | Adaptive |
| Coloring | $O(\alpha \log n)^4$ | $O(\mathcal{B} \log^2 n)$ | $\tilde{O}(\log^2 n)$ | Oblivious |
| Coloring | $O(2^\alpha)$ | $O(\mathcal{B} \log^3 n)$ | $\tilde{O}(\log^2 n)$ | Adaptive |

Table 6.2: Work and span bounds of algorithms in this chapter.⁵

satisfying specific degree properties in certain induced subgraphs. In the LDS, vertices are updated one at a time. One of our main technical insights is that we can update many vertices *simultaneously*, leading to high parallelism. Our k -core decomposition algorithm is work-efficient, and matches the approximation factor of the best-known sequential dynamic approximate k -core decomposition algorithm of Sun et al. [310], while achieving polylogarithmic span w.h.p. Dynamic problems related to k -core decompositions have been recently studied in the theory community, such as densest subgraph [42, 287] and low out-degree orientations [36, 161, 61, 196, 194, 187, 164, 307]; some of these works use the LDS. However, none of these previous works proved guarantees regarding the k -core decomposition that can be maintained via a LDS. Notably, we show via a new, intuitive proof that one can use the level of a vertex to estimate its coreness in the LDS of [164]. Unlike the proof in [310] for their dynamic algorithm, our proof does not require densest subgraphs nor any additional information besides the two invariants maintained by the structure.

Our main theoretical and practical technical contributions for k -core decomposition are three-fold: (1) we present a simple modification and a new $(2 + \varepsilon)$ -approximate coreness proof for the sequential level data structure of [42, 164] (which were not previously used for coreness values) using only the levels of the vertices—no such modification was known prior to this work since [310] requires an additional elimination/peeling/round-indexing procedure; (2) we provide the first parallel work-efficient batch-dynamic level data structure that takes $O(\log^2 n \log \log n)$ span w.h.p. which we use to obtain a $(2 + \varepsilon)$ -approximate batch-dynamic k -core decomposition algorithm; and (3) we provide multicore implementations of our new algorithm and demonstrate its practicality through extensive experimentation with state-of-the-art parallel and sequential algorithms.

The following theorems give our theoretical bounds.

Theorem 6.1 (Batch-Dynamic k -Core Decomposition). *Given $G = (V, E)$ where $n = |V|$ and batch of updates \mathcal{B} , our algorithm maintains $(2 + \varepsilon)$ -approximations of*

³ \tilde{O} hides a factor of $O(\log \log n)$.

⁴We denote by α the current arboricity of the graph after processing all updates including the most recent ones.

⁵All bounds are w.h.p. except for the work of static k -core and $O(\alpha \log n)$ -coloring.

core values for all vertices (for any constant $\varepsilon > 0$) in $O(|\mathcal{B}| \log^2 n)$ amortized work and $O(\log^2 n \log \log n)$ span w.h.p. using $O(n \log^2 n + m)$ space.

Using the same parallel level data structure, we also obtain the following result for maintaining a low out-degree orientation.

Theorem 6.2 (Batch-Dynamic Low Out-Degree Orientation). *Our algorithm maintains an $(4 + \varepsilon)$ -approximation of a minimum acyclic out-degree orientation, with the same bounds as Theorem 6.1, where the amortized number of edge flips is $O(|\mathcal{B}| \log^2 n)$.*

A consequence of Theorem 6.2 is the following corollary.

Corollary 6.1 ($O(\alpha)$ Out-Degree Orientation). *Our algorithm maintains an $O(\alpha)$ out-degree orientation, where α is the current arboricity (Definition 2.4), with the same bounds as Theorem 6.2.*

Using Theorem 6.2, we design a framework for parallel batch-dynamic algorithms on bounded-arboricity graphs for batch of updates \mathcal{B} , which in addition to problem-specific techniques allows us to obtain a set of batch-dynamic algorithms for a variety of other fundamental graph problems including maximal matching, clique counting, and vertex coloring. We present the bounds that we obtain for these problems here for completeness, but note that we have omitted a detailed discussion of maximal matching and vertex coloring from this chapter, which we defer to the paper [216]. The coloring algorithms are based heavily on the sequential algorithms of Henzinger et al. [164], and Fig. 6.3 summarizes the update times of the previous best-known sequential results.

Theorem 6.3 (Batch-Dynamic Maximal Matching). *We maintain a maximal matching in $O(|\mathcal{B}|(\alpha + \log^2 n))$ amortized work and $O(\log^2 n (\log \Delta + \log \log n))$ span w.h.p.⁶ in $O(n \log^2 n + m)$ space.*

Theorem 6.4 (Batch-Dynamic Implicit $O(2^\alpha)$ -Vertex Coloring). *We maintain an implicit $O(2^\alpha)$ -vertex coloring⁷ in $O(|\mathcal{B}| \log^3 n)$ amortized work and $O(\log^2 n)$ span w.h.p. or updates, and $O(Q\alpha \log n)$ work and $O(\log n)$ span w.h.p. for Q queries, using $O(n \log^2 n + m)$ space.*

Theorem 6.5 (Batch-Dynamic k -Clique Counting). *We maintain the count of k -cliques in $O(|\mathcal{B}| \alpha^{k-2} \log^2 n)$ amortized work and $O(\log^2 n \log \log n)$ span w.h.p. in $O(m\alpha^{k-2} + n \log^2 n)$ space.*

All of the above results are robust against *adaptive* adversaries which have access to the algorithm's previous outputs. The following algorithm is robust against *oblivious* adversaries which do not have access to previous outputs.

Theorem 6.6. *We maintain an $O(\alpha \log n)$ -vertex coloring in $O(|\mathcal{B}| \log^2 n)$ amortized expected work and $O(\log^2 n \log \log n)$ span w.h.p. in $O(m + n \log^2 n + \alpha \log n)$ space.*

⁶ Δ denotes the maximum *current* degree of the graph after processing all updates.

⁷An *implicit* vertex coloring algorithm returns valid colorings for queried vertices.

Our k -core, low out-degree orientation, and vertex coloring algorithms are work-efficient when compared to the best-known sequential, dynamic algorithms for the respective problems [42, 164, 310]. For maximal matching, our algorithm is work-efficient when $\alpha = \Omega(\log^2 n)$ when compared to the best-known sequential algorithm that is robust against adaptive adversaries [161, 244]; the extra work when $\alpha = o(\log^2 n)$ comes from the fact that our bounds are with respect to the *current* arboricity, compared to [161, 244] whose bounds are with respect to the *maximum* arboricity over the sequence of updates.

The best-known batch-dynamic algorithm for k -clique counting, by Dhulipala et al. [104], takes $O(|\mathcal{B}|m\alpha^{k-4})$ expected work and $O(\log^{k-2} n)$ span w.h.p. using $O(m + |\mathcal{B}|)$ space. Compared with their algorithm, our algorithm uses less work when $m = \omega(\alpha^2 \log^2 n)$. In many real-world networks, $\alpha \ll \sqrt{m}$ (see e.g., Table 6.3, for maximum k -core values, which upper bound α); thus, our result is more efficient in many cases at an additional multiplicative space cost of $O(\alpha^{k-2})$. We also obtain smaller span for all $k > 4$. We provide further comparisons with the best-known sequential clique counting algorithm [115], and we describe more specific batch-dynamic challenges we face in designing the above algorithms in their respective sections. The components of the PLDS used in each of the above results are summarized in Fig. 6.2.

Finally, using ideas from our batch-dynamic k -core decomposition algorithm, we provide a new parallel static $(2 + \varepsilon)$ -approximate k -core decomposition algorithm. We compare this algorithm with the best-known parallel static exact algorithm of [95] which uses $O(m + n)$ expected work and $O(\rho \log m)$ span w.h.p. where ρ is the *number of steps necessary to peel all vertices* (ρ could potentially be $\Omega(n)$). Hence, [95] does not guarantee $\text{poly}(\log n)$ span.

Theorem 6.7. *Given $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, for any constant $\varepsilon > 0$, our algorithm finds an $(2 + \varepsilon)$ -approximate k -core decomposition in $O(n + m)$ expected work and $O(\log^3 n)$ span w.h.p. using $O(n + m)$ space.*

Experimental Contributions. In addition to our theoretical contributions, we also provide optimized multicore implementations of our k -core decomposition algorithms. We compare the performance of our algorithms with state-of-the-art algorithms on a variety of real-world graphs using a 30-core machine with two-way hyper-threading. Our parallel static approximate k -core algorithm achieves a 2.8–3.9x speedup over the fastest parallel exact k -core algorithm [95] and achieves a 14.76–36.07x self-relative speedup.

We show that our parallel batch-dynamic k -core algorithm achieves up to 544.22 \times speedups over the state-of-the-art sequential dynamic approximate k -core algorithm of Sun et al. [310], while achieving comparable accuracy. We also achieve up to 114.52 \times speedups over the state-of-the-art parallel batch-dynamic exact k -core algorithm of Hua et al. [169], and up to 723.72 \times speedups against the state-of-the-art sequential exact k -core algorithm of Zhang and Yu [351]. Our batch-dynamic algorithm outperforms the best multicore static k -core algorithms by up to 121.76 \times on batch sizes that are less than 1/3 of the number of edges in the entire graph.

Our algorithm exhibits improvements in runtime while maintaining the same or smaller error, even when using only four threads (available on a standard laptop),

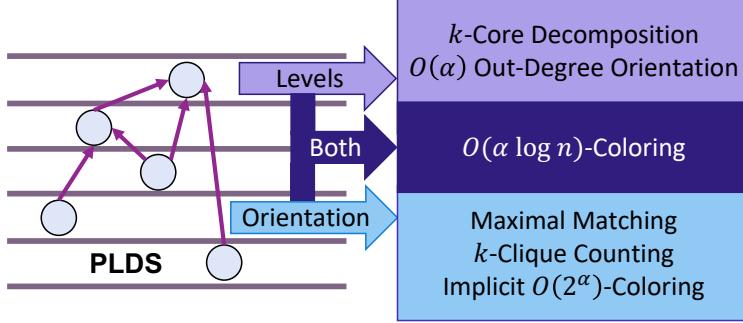


Figure 6.2: This figure shows what parts of the PLDS are used in each result. The level of each vertex is used to determine the k -core decomposition (Theorem 6.1) and low out-degree orientation (Theorem 6.2 and Corollary 6.1). The orientation of the edges is used for maximal matching (Theorem 6.3), implicit $O(2^\alpha)$ -coloring (Theorem 6.4), and k -clique counting (Theorem 6.5). Finally, both are used for $O(\alpha \log n)$ -coloring (Theorem 6.6).

and remains competitive at one thread. We demonstrate that existing exact dynamic implementations are not efficient or scalable enough to handle graphs with billions of edges, whereas our algorithm is able to. Furthermore, our demonstrated speedups of up to two orders of magnitude indicates that our implementation not only fills the gap for processing graphs that are orders of magnitude larger than can be handled by existing implementations, but also that it is the best option for many smaller networks. Our code is publicly available at <https://github.com/qqliu/batch-dynamic-kcore-decomposition>.

6.4 Comparisons with Other Related Work

Parallel Exact Batch-Dynamic Algorithms The most recent, state-of-the-art parallel batch-dynamic algorithm by Hua et al. [169] improves upon the previous parallel algorithms of Aridhi et al. [18], Wang et al. [332], and Jin et al. [184]. Their algorithm relies on the concept of a *joint edge set*, whose insertion and removal determines the core numbers of the vertices. However, their algorithm could take $\Omega(n)$ span as they use a standard depth-first search to traverse vertices in the joint edge set as well as vertices outside of the joint edge set. In comparison, our algorithm probably has $O(\log^2 n \log \log n)$ span w.h.p. ur theoretical improvements also translate to practical gains since we demonstrate greater scalability in our experiments.

Another recent work by Gabert et al. [139] provides a scalable exact k -core maintenance algorithm. Both their asymptotic work and span is super-polylogarithmic (in fact, in the worst case it could be as bad as computing from scratch). Unfortunately, the code for their experiments is proprietary and hence not available for comparison. However, their reported experimental results overall appear slower than our results, described in more detail in Section 6.6.4.

| Summary of Best-Known Sequential Results | | | |
|--|---------------------|--|-----------|
| Problem | Approx | Update Time | Adversary |
| k -core | $(2 + \varepsilon)$ | $O(\log^2 n)$ [164, 310], Lemma 6.1 | Adaptive |
| Orientation | $(4 + \varepsilon)$ | $O(\log^2 n)$ [164] | Adaptive |
| Matching | Maximal | $O(\alpha_{\max} + \log n / \log \log n)$ [244, 161] | Adaptive |
| k -clique | Exact | $O(\alpha_{\max}^{k^2} \log^{k^2} n)$ [115] | Adaptive |
| Coloring | $O(\alpha \log n)$ | $O(\log^2 n)$ [164] | Oblivious |
| Coloring | $O(2^\alpha)$ | $O(\log^3 n)$ [164] | Adaptive |

Figure 6.3: Previous best-known sequential algorithm results.

Low Out-Degree Orientations Many previous works give dynamic algorithms for low out-degree orientations with respect to bounds on the *maximum* arboricity that ever exists in the graph, α_{\max} [61, 161, 36, 196, 194, 187, 307]. Noticeably, these sequential, dynamic works save a $O(\log n)$ factor in the running time compared to sequential dynamic algorithms that compute the orientation with respect to the *current* arboricity [42, 164]. In practice, the arboricity of real-world graphs may vary as batches of updates are applied, and in particular, the k -core numbers of each vertex can change drastically (e.g., many follows and unfollows can occur in a very short period of time following a viral post). Our work matches the update time of [42, 164] for maintaining a low out-degree orientation for the current α . This explains why our work bounds for maximal matching requires an additional $O(\log n)$ factor compared to previous works [161, 244] that were in terms of α_{\max} .

Other Graph Problems Using low out-degree orientations, a number of works in the past have studied the other dynamic graph problems we study in this paper, including maximal matching, vertex coloring, and clique counting [37, 38, 115, 104, 138, 197, 244, 164, 161, 307, 28, 72, 256, 160, 220]. The best update time for these problems in the sequential settings are summarized in Fig. 6.3.

In the sequential setting, the best-known algorithm for k -clique counting uses $O(\log^{k^2} n)$ update time in bounded *expansion* graphs for any k -vertex subgraph [115]. Bounded expansion is a more restricted class of graphs than bounded arboricity.⁸ Their algorithm crucially requires the *fraternal augmentation* graph, G' , which is created from an input directed graph, $G = (V, E)$, by adding an edge (u, v) (direction chosen arbitrarily) if and only if (w, u) and (w, v) exist. Provided an out-degree orientation of size σ , their algorithm runs in $O(\sigma^{k^2} \log^{k^2} n)$ time; so for bounded arboricity graphs, their algorithm can find any subgraph of size k with $O(\alpha^{k^2} \log^{k^2} n)$ update time [115]. Our algorithm also gives a better update time in the sequential setting than [115] for counting cliques (for $|\mathcal{B}| = 1$).

⁸Graphs with bounded expansion have bounded arboricity, but not vice versa.

6.5 Batch-Dynamic k -Core Decomposition

In this section, we describe our parallel, batch-dynamic algorithm for maintaining an $(2 + \varepsilon)$ -approximate k -core decomposition (for any constant $\varepsilon > 0$) and prove its theoretical efficiency.

6.5.1 Algorithm Overview

We present a *parallel level data structure (PLDS)* that maintains a $(2 + \varepsilon)$ -approximate k -core decomposition that is inspired by the class of sequential level data structures (LDS) of [42, 164]. Our algorithm achieves $O(\log^2 n)$ amortized work per update and $O(\log^2 n \log \log n)$ span w.h.p. We also present a deterministic version of our algorithm that achieves the same work bound with $O(\log^3 n)$ span. Our data structure can also handle batches of vertex insertions/deletions. Our data structure requires $O(\log^2 n)$ amortized work, which matches the $O(\log^2 n)$ amortized update time of [42, 164]. We also present a *deterministic* version of our algorithm that achieves the same work bound with $O(\log^3 n)$ span in citeShiDynArb.

In addition to edge updates, our data structure also handles batches of vertex insertions/deletions, discussed in [216]. As in [164], our data structure can handle *changing arboricity* that is not known a priori. Such adaptivity is necessary to successfully maintain accurate approximations of coreness values.

The LDS and our PLDS consists of a partition of the vertices into $K = O(\log^2 n)$ *levels*.⁹ We provide a very high level overview of PLDS in this section. The levels are partitioned into equal-sized *groups* of consecutive levels. Updates are partitioned into insertions and deletions. Vertices move up and down levels depending on the type of edge update incident to the vertex. Rules governing the induced degrees of vertices to neighbors in different levels determine whether a vertex moves. Using information about the level of a vertex, we obtain a $(2 + \varepsilon)$ -approximation on the coreness of the vertex.

After every edge update, vertices update their levels depending on whether they satisfy two invariants. One invariant upper bounds the induced degree of each vertex v in the subgraph consisting of all vertices in the same or higher level. Vertices whose degree exceeds this bound move up one or more levels. We process the levels from smallest to largest level and move all vertices from the same level in parallel. The second invariant lower bounds the induced degree of each vertex v in the subgraph consisting of all vertices in the level below v , the level of v and all levels higher than the level of v . Vertices that violate this invariant calculate a *desire-level* or the closest level they can move to that satisfies this invariant. Then, vertices with the same desire-level are moved in parallel to that level. Finally, the coreness estimates of the vertices are computed based on the current level of each vertex. We obtain the low out-degree orientation by orienting edges from lower to higher levels (breaking ties by vertex index). Fig. 6.5 shows the invariants maintained by our algorithm; Figs. 6.6 and 6.7 show how our algorithm processes insertion and deletion updates.

⁹When $m = o(n)$, we can also show that $O(\log^2 m)$ levels suffice.

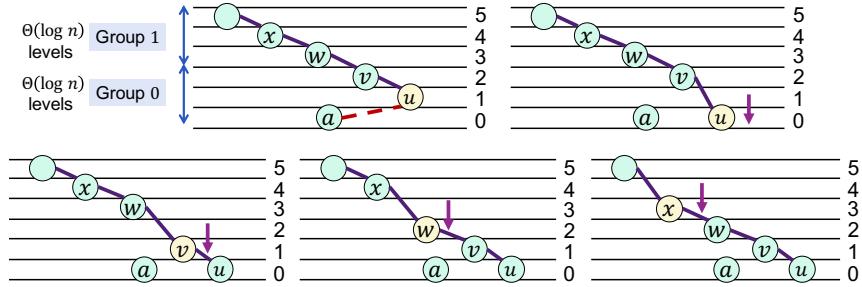


Figure 6.4: Example of a cascade of vertex movements caused by an edge deletion on u (shown by the dashed red line).

Together, they demonstrate an example run of our algorithm.

6.5.2 Sequential Level Data Structure (LDS)

The sequential level data structures (LDS) of [42, 164] maintains a low out-degree orientation under dynamic updates. Within their LDS, a vertex moves up or down levels one by one, where a vertex v (incident to an edge update) first checks whether an invariant is violated, and then may move up or down one level. Then, the vertex checks the invariants and repeats. Such movements may cause other vertices to move up or down levels. The LDS combined with our Section 6.5.4 directly gives an $O(\log^2 n)$ update time sequential, dynamic algorithm that outputs $(2 + \varepsilon)$ -approximate coreness values.

Unfortunately, such a procedure can be slow in practice. Specifically, a vertex that moves one level could cause a cascade of vertices to move one level. Then, if the vertex moves again, the same cascade of movements may occur. An example is shown in Fig. 6.4. Furthermore, any trivial parallelization of the LDS to support a batch of updates will run into race conditions and other issues, requiring the use of locks which blows up the runtime in practice.

Thus, our PLDS solves several challenges posed by the sequential LDS. Given a batch \mathcal{B} of edge updates: (1) our algorithm processes the levels in a careful order that yields provably low span for batches of updates; (2) our insertion algorithm processes vertices on each level at most once, which is key to the span bounds—after vertices move up from level ℓ , no future step in the algorithm moves a vertex up from level ℓ ; and (3) our deletion algorithm moves vertices to their final level in one step. In other words, a vertex moves at most once in a deletion batch.

6.5.3 Detailed PLDS Algorithm

As mentioned previously, the vertices of the input graph $G = (V, E)$ in our PLDS are partitioned across K **levels**. For each level $\ell = 0, \dots, K - 1$, let V_ℓ be the set of vertices that are currently assigned to level ℓ . Let Z_ℓ be the set of vertices in levels $\geq \ell$. Provided a constant $\delta > 0$, the levels are partitioned into **groups**

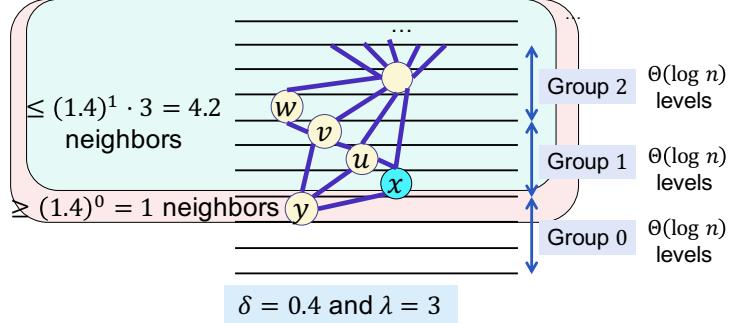


Figure 6.5: Example of invariants maintained by the PLDS for $\delta = 0.4$ and $\lambda = 3$. There are $\Theta(\log n)$ groups, each with $\Theta(\log n)$. Each vertex is in exactly one level of the structure and moves up and down by some movement rules. For example, vertex x (blue) is on level 3 and in group 1.

$g_0, \dots, g_{\lceil \log_{(1+\delta)} n \rceil}$, where each group contains $4 \lceil \log_{(1+\delta)} n \rceil$ consecutive levels. Each $\ell \in [i \lceil \log_{(1+\delta)} n \rceil, \dots, (i+1) \lceil \log_{(1+\delta)} n \rceil - 1]$ is a level in group i . Our data structure consists of $K = O(\log^2 n)$ total levels. The PLDS satisfies the following invariants as introduced in [42, 164], which also govern how the data structure is maintained. The invariants assume a given constant $\delta > 0$ and a constant $\lambda > 0$.

Invariant 1 (Degree Upper Bound). *If vertex $v \in V_\ell$, level $\ell < K$ and $\ell \in g_i$, then v has at most $(2 + 3/\lambda)(1 + \delta)^i$ neighbors in Z_ℓ .*

Invariant 2 (Degree Lower Bound). *If vertex $v \in V_\ell$, level $\ell > 0$, and $\ell - 1 \in g_i$, then v has at least $(1 + \delta)^i$ neighbors in $Z_{\ell-1}$.*

Vertices with no neighbors are placed in level 0. An example partitioning of vertices and maintained invariants is shown in Fig. 6.5. Let $\ell(v)$ be the level that v is currently on. We define the **group number**, $g(v)$, of a vertex v to be the index i of the group g_i where $\ell(v) \in g_i$. Similarly, we define $gn(\ell) = i$ to be the group number for level ℓ where $\ell \in g_i$. We define the **up-degree**, $up(v)$, of a vertex v to be the number of its neighbors in $Z_{\ell(v)}$ (**up-neighbors**), and **up*-degree**, $up^*(v)$, to be the number of its neighbors in $Z_{\ell(v)-1}$ (**up*-neighbors**). These two notions of induced degree correspond to the requirements of the two invariants of our data structure. We define neighbors w of v at levels $\ell(w) < \ell(v)$ to be the **down-neighbors** of v . Lastly, the **desire-level** $dl(v)$ of a vertex v is the *closest level to the current level of the vertex* that satisfies both Invariant 1 and Invariant 2.

Definition 6.3 (Desire-level). *The desire-level, $dl(v)$, of vertex v is the level ℓ' that minimizes $|\ell(v) - \ell'|$, and where $up^*(v) \geq (1 + \delta)^{i'}$ and $up(v) \leq (2 + 3/\lambda)(1 + \delta)^i$ where $\ell' - 1 \in g_{i'}$, $\ell' \in g_i$, and $i' \leq i$. In other words, the desire-level of v is the closest level ℓ' to the current level of v , $\ell(v)$, where both Invariant 1 and Invariant 2 are satisfied.*

We show that the invariants are always maintained except for a period of time when processing a new batch of insertions/deletions. During this period, the data

Algorithm 6.1 – Update(\mathcal{B})

Require: A batch of edge updates \mathcal{B} .

- 1: Let \mathcal{B}_{ins} = all edge insertions in \mathcal{B} , and \mathcal{B}_{del} = all edge deletions in \mathcal{B} .
 - 2: Call RebalanceInsertions(\mathcal{B}_{ins}). [Algorithm 6.2]
 - 3: Call RebalanceDeletions(\mathcal{B}_{del}). [Algorithm 6.3]
-

Algorithm 6.2 – RebalanceInsertions(B_{ins})

Require: A batch of edge insertions B_{ins} .

- 1: Let U contain all up-neighbors of each vertex, keyed by vertex. So $U[v]$ contains all up-neighbors of v .
 - 2: Let L_v contain all neighbors of v in levels $[0, \dots, \ell(v) - 1]$, keyed by level number.
 - 3: **parfor** each edge insertion $e = (u, v) \in \mathcal{B}_{ins}$ **do**
 - 4: Insert e into the graph.
 - 5: **for** each level $l \in [0, \dots, K - 1]$ starting with $l = 0$ **do**
 - 6: **parfor** each vertex v incident to B_{ins} or is marked, where $\ell(v) = l \cap \text{up}(v) > (2 + 3/\lambda)(1 + \delta)^{gn(l)}$ **do**
 - 7: Mark and move v to level $l + 1$ and create $L_v[l]$ to store v 's neighbors at level l .
 - 8: **parfor** each $w \in N(v)$ of a vertex v that moved to level $l + 1$ and w stayed in level l **do**
 - 9: $U[v] \leftarrow U[v] \setminus \{w\}, L_v[l] \leftarrow L_v[l] \cup \{w\}$.
 - 10: **parfor** each $u \in N(v)$ of a vertex v that moved to level $l + 1$ and u is in level $l + 1$ **do**
 - 11: Mark u if $\text{up}(u) > (2 + 3/\lambda)(1 + \delta)^{gn(l+1)}$.
 - 12: $U[u] \leftarrow U[u] \cup \{v\}, L_u[l] \leftarrow L_u[l] \setminus \{v\}$.
 - 13: **parfor** each $x \in N(v)$ of a vertex v that moved to level $l + 1$ and x is in level $\ell(x) \geq l + 2$ **do**
 - 14: $L_x[l] \leftarrow L_x[l] \setminus \{v\}, L_x[l + 1] \leftarrow L_x[l + 1] \cup \{v\}$.
 - 15: Unmark v if $\text{up}(v) \leq (2 + 3/\lambda)(1 + \delta)^{gn(l+1)}$. Otherwise, leave v marked.
-

structure undergoes a *rebalance procedure*, where the invariants may be violated. The main update procedure in Algorithm 6.1 separates the updates into insertions and deletions (Line 1), and then calls RebalanceInsertions (Line 2) and RebalanceDeletions (Line 3). We make two *crucial* observations: when processing a batch of insertions, Invariant 2 is never violated; and, similarly, when processing a batch of deletions, Invariant 1 is never violated. Thus, no vertex needs to move *down* when processing an insertion batch and no vertex needs to move *up* when processing a deletion batch. The two procedures are asymmetric, and so we first describe RebalanceInsertions (Algorithm 6.2), and then describe RebalanceDeletions (Algorithm 6.3).

Data Structures. Each vertex v keeps track of its set of neighbors in two structures. U keeps track of the neighbors at v 's level and above. We denote this set of v 's neighbors by $U[v]$. L_v keeps track of neighbors of v for every level below $\ell(v)$ —in particular, $L_v[j]$ contains the neighbors of v at level $j < \ell(v)$.

RebalanceInsertions(B_{ins}). Algorithm 6.2 shows the pseudocode. Provided a batch of insertions B_{ins} , we iterate through the K levels from the lowest level $\ell = 0$ to

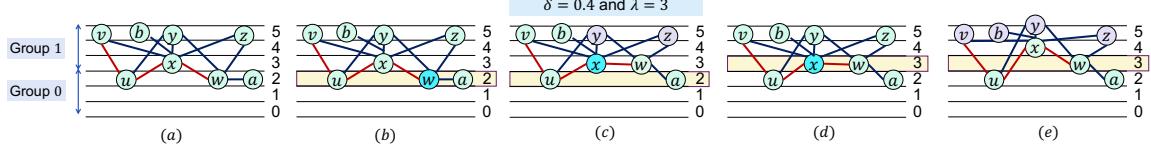


Figure 6.6: Example of RebalanceInsertions described in the text for $\delta = 0.4$ and $\lambda = 3$. The red lines represent the batch of edge insertions.

the highest level $\ell = K - 1$ (Line 5). For each level, in parallel we check the vertices incident to edge insertions in B_{ins} or is marked to see if they violate Invariant 1 (Line 6). If a vertex v in the current level l violates Invariant 1, we move v to level $l + 1$ (Line 7). After moving v , we update structures $U[v]$, L_v , and the structures of $w \in N(v)$ where $\ell(w) \in [l, l + 1]$. First, we create $L_v[l]$ to store the neighbors of v in level l (Line 7). If v moved to level $l + 1$ and w stayed in level l , then we delete w from $U[v]$ and instead insert w into $L_v[l]$ (Lines 8–9). We do not need to make any data structure modifications for w since v stays in $U[w]$. Similarly, no data structure modifications to v and w are necessary when both v and w move to level $l + 1$. For each neighbor of v on level $l + 1$, we need to check whether it now violates Invariant 1 (Line 10). If it does, then we mark the vertex (Line 11). We process any such marked vertices when we process level $l + 1$. We also update the U and L arrays of every neighbor of v on level $l + 1$ (Line 12). Specifically, let u be one such neighbor, we add v to $U[u]$ and remove v from $L_u[l]$. We conclude by making appropriate modifications to L for each neighbor on levels $\geq l + 2$ (Lines 13–14). Specifically, let x be one such neighbor, we remove v from $L_x[l]$ and add v to $L_x[l + 1]$. All neighbors of vertices that moved can be checked and processed in parallel. Finally, v becomes unmarked if it satisfies all invariants; otherwise, it remains marked and must move again in a future step (Line 15).

Fig. 6.6 shows an example of our entire insertion procedure described in Algorithm 6.2 for $\delta = 0.4$ and $\lambda = 3$. The red lines in the example represent the batch of edge insertions. Thus, in (a), the newly inserted edges are the edges (u, v) , (u, x) , and (x, w) . We iterate from the bottommost level (level 0) to the topmost level (level $K - 1$).

The first level where we encounter vertices that are marked or are adjacent to an edge insertion is level 2. Since level 2 is part of group 0, the cutoff for Invariant 1 is $(2 + 3/\lambda)(1 + \delta)^0 = 3$ provided $\lambda = 3$ and $\delta = 0.4$. In level 2, only w violates Invariant 1 since the number of its neighbors on levels ≥ 2 is 4 (x, y, z , and a), so $\text{up}(w) = 4 > 3$ (shown in (b)). Then, in (c), we move w up to level 3. We need to update the data structures for neighbors of w at level 3 and above (as well as w 's own data structures); the vertices with data structure updates are x, w, y , and z . After the move, x becomes marked because it now violates Invariant 1 (the cutoff for level 3 is $(2 + 3/3)(1 + 0.4) = 4.2$ since level 3 is in group 1); w becomes unmarked because it no longer violates Invariant 1. In (d), we move on to process level 3. The only vertex that is marked or violates Invariant 1 is x . Therefore, we move x up one level (shown in (e)) and update relevant data structures (of x, v, y, z , and b).

Algorithm 6.3 – RebalanceDeletions(\mathcal{B}_{del})

Require: A batch of edge deletions \mathcal{B}_{del} .

```

1: Let  $U$  contain all up-neighbors of each vertex, keyed by vertex. So  $U[v]$  contains all
   up-neighbors of  $v$ . Let  $L_v$  contain all neighbors of  $v$  in levels  $[0, \dots, \ell(v) - 1]$ , keyed by
   level number.
2: parfor each edge deletion  $e = (u, v) \in \mathcal{B}_{del}$  do
3:   Remove  $e$  from the graph.
4: parfor each vertex  $v$  where  $up^*(v) < (1 + \delta)^{gn(\ell(v)-1)}$  do
5:   Calculate  $dl(v)$  using CalculateDesireLevel( $v$ ).
6: for each level  $l \in [0, \dots, K - 1]$  starting with level  $l = 0$  do
7:   parfor each vertex  $v$  where  $dl(v) = l$  do
8:     Move  $v$  to level  $l$ .
9:   parfor each vertex  $v$  where  $dl(v) = l$  do
10:    parfor each neighbor  $w$  of  $v$  where  $\ell(w) \geq l$  do
11:      Let  $p_v$  and  $p_w$  be the previous levels of  $v$  and  $w$ , respectively, before the move.
12:      if  $\ell(w) = l$  then
13:         $L_w[p_v] \leftarrow L_w[p_v] \setminus \{v\}$ ,  $L_v[p_w] \leftarrow L_v[p_w] \setminus \{w\}$ .
14:         $U[w] \leftarrow U[w] \cup \{v\}$ ,  $U[v] \leftarrow U[v] \cup \{w\}$ .
15:      else
16:        if  $p_v > \ell(w)$  then
17:           $U[w] \leftarrow U[w] \setminus \{v\}$ ,  $L_v[\ell(w)] \leftarrow L_v[\ell(w)] \setminus \{w\}$ .
18:        else if  $p_v = \ell(w)$  then
19:           $U[w] \leftarrow U[w] \setminus \{v\}$ .
20:        else  $L_w[p_v] \leftarrow L_w[p_v] \setminus \{v\}$ .
21:         $L_w[l] \leftarrow L_w[l] \cup \{v\}$ ,  $U[v] \leftarrow U[v] \cup \{w\}$ .
22:        if  $up^*(w) < (1 + \delta)^{gn(\ell(w)-1)}$  then
23:          Recalculate  $dl(w)$  using Algorithm 6.4.

```

RebalanceDeletions(\mathcal{B}_{del}). Unlike in LDS, deletions in PLDS are handled by moving each vertex at most once, directly to its final level (the vertex *does not move* again during this procedure). We show in the analysis that this guarantee is *crucial to obtaining low span*. The pseudocode is shown in Algorithm 6.3. For each vertex v incident to an edge deletion, we check whether it violates Invariant 2 (Line 4). On Line 4, $gn(\ell(v) - 1)$ gives the group number i where $\ell(v) - 1 \in g_i$. If v violates Invariant 2, we calculate its desire-level, $dl(v)$, using CalculateDesireLevel (Line 5), described next. We iterate through the levels from $l = 0$ to $l = K - 1$ (Line 6). Then, in parallel for each vertex v whose desire-level is l , we move v to level l (Lines 7–8). We update the data structures of each v that moved and $w \in N(v)$ where $\ell(w) \geq l$ (Lines 9–21). Specifically, we need to update $U[v]$, $U[w]$, L_v , and L_w if v was originally an up-neighbor of w and becomes a down-neighbor or vice versa. Finally, we update the desire-level of neighbors of v that no longer satisfy Invariant 2 (Lines 22–23). We process all vertices that move and their neighbors in parallel.

Fig. 6.7 shows an example of Algorithm 6.3 for $\delta = 1$ and $\lambda = 3$. In (a), the newly deleted edges are (x, z) and (y, w) . For each vertex adjacent to an edge deletion,

Algorithm 6.4 – CalculateDesireLevel(v)

Require: A vertex v that needs to move to a level $j < \ell(v)$.

Ensure: The desire-level $\text{dl}(v)$ of vertex v .

- 1: $d \leftarrow \text{up}^*(v), p \leftarrow 1, i \leftarrow 2$
 - 2: **while** $d < (1 + \delta)^{gn(\ell(v) - p)}$ and $\ell(v) - p > 0$ **do**
 - 3: $d \leftarrow d + \sum_{j=p}^{i-1} |L_v[\ell(v) - j - 1]|$
 - 4: **if** $d \geq (1 + \delta)^{gn(\ell(v) - i)}$ **then**
 - 5: Binary search within levels $[\ell(v) - i + 1, \dots, \ell(v) - p]$ to find the closest level to $\ell(v)$ that satisfies Invariants 1 and 2; **return** this level.
 - 6: $p \leftarrow i, i \leftarrow \min(2 \cdot i, \ell(v))$.
 - 7: **return** 0.
-

we calculate its desire-level, or the closest level to its current level that satisfies Invariant 2. In (b), only x and z violate Invariant 2. The lower bound on the number of neighbors that must be at or above level 3 for x and level 4 for z is $(1 + \delta)^1 = 2$ since $\delta = 1$ and levels 3 and 4 are in group 1. (Recall that the lower bound is calculated with respect to the level *below* x and z .) We calculate that the desire-levels of x and z are both 3. The desire-levels of y and w are their current levels because they do not violate the invariant. Then, we iterate from the bottommost level (starting with level 0) to the topmost level (level $K - 1$). Level 3 is the first level where vertices want to move. Then, we move x and z to level 3 (shown in (c)). We only need to update the data structures of neighbors at or above x and z so we only update the structures of x , y , and z . Invariant 2 is no longer violated for x and z . In fact, our algorithm guarantees that each vertex *moves at most once*. We check whether any of x or z 's up-neighbors violate Invariant 2. Indeed, y now violates the invariant. In (d), we recompute the desire-level of y and its desire-level is now 4. Then, we move y to level 4 in (e).

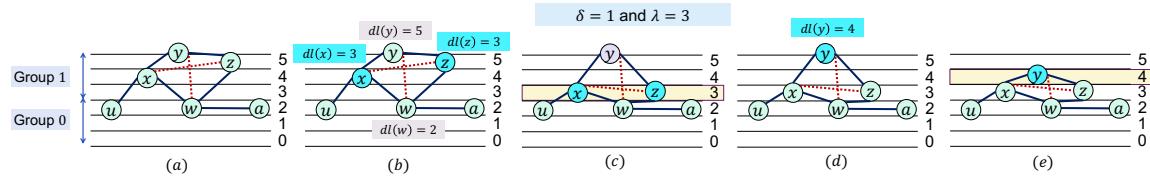


Figure 6.7: Example of RebalanceDeletions described in the text for $\delta = 1$ and $\lambda = 3$. The red dotted lines represent the batch of edge deletions.

CalculateDesireLevel(v). Algorithm 6.4 shows the procedure for calculating the desire-level, $\text{dl}(v)$, of vertex v , which is used in Algorithm 6.3. Let $gn(\ell)$ be the index i where level $\ell \in g_i$. We use a doubling procedure followed by a binary search to calculate the desire-level. We initialize a variable d to $\text{up}^*(v)$ (number of neighbors at or above level $\ell(v) - 1$). Starting with level $\ell(v) - 2$, we add the number of neighbors in level $\ell(v) - 2$ to d (Algorithm 6.4, Line 3). This procedure checks whether moving v to $\ell(v) - 1$ satisfies Invariant 2 (Line 4). If it passes the check, then we are done and we move v to $\ell(v) - 1$. Otherwise, we iteratively double the number of levels

from which we count neighbors until we find a level where Invariant 2 is satisfied (Line 6). On each iteration, we sum the number of neighbors (Line 3) in the range of levels using a parallel reduce. We continue until we find a level where Invariant 2 is satisfied. Let this level be ℓ' and the previous cutoff be ℓ_{prev} . Finally, we perform a binary search within the range $[\ell', \dots, \ell_{prev}]$ to find the *closest* level to $\ell(v)$ that satisfies Invariant 2 (Line 5).

We defer an analysis of the correctness, and the work and span of our PLDS to the paper [216].

6.5.4 Estimating the Coreness and Orientation

($2 + \varepsilon$)-Approximation of Coreness. The *coreness estimate*, $\hat{k}(v)$, is an estimate of the coreness of a vertex v . We compute a coreness estimate using *only* v 's level and the number of levels per group (which is fixed). We show how to use such information to obtain a $(2 + \varepsilon)$ -approximation to the actual coreness of v for any constant $\varepsilon > 0$. (We can find an approximation for any fixed ε by appropriately setting δ and λ .) To calculate $\hat{k}(v)$, we find the largest index i of a group g_i , where $\ell(v)$ is at least as high as the highest level in g_i .

Definition 6.4 (Coreness Estimate). *The coreness estimate $\hat{k}(v)$ of vertex v is $(1 + \delta)^{\max(\lfloor \ell(v) + 1 \rfloor / 4 \lceil \log_{1+\delta} n \rceil - 1, 0)}$, where each group has $4 \lceil \log_{(1+\delta)} n \rceil$ levels.*

To see an example, consider vertex y in Fig. 6.7 (e). We estimate $\hat{k}(y) = 1$ since the highest level that is the last level of a group and is equal to or below level $\ell(y) = 4$ is level 2. Level 2 is part of group 0, and so our coreness estimate for y is $(1 + \delta)^0 = 1$. This is a 2-approximation of its actual coreness of 2. Using Definition 6.4, we prove that our PLDS maintains a $(2 + 3/\lambda)(1 + \delta)$ -approximation of the coreness value of each vertex, for any constants $\lambda > 0$ and $\delta > 0$. Therefore, we obtain the following lemma giving the desired $(2 + \varepsilon)$ -approximation. Our experimental analysis shows that our theoretical bounds limit the maximum error of our experiments, although our average errors are much smaller. To get a maximum error bound of $(2 + \varepsilon)$ for any $\varepsilon > 0$, we can set $\delta = \varepsilon/3$ and $\lambda = \frac{9}{\varepsilon} + 3$.

By Lemma 6.2, it suffices to return $\hat{k}(v)$ as the estimate of the coreness of v ; this proves the approximation factor in Theorem 6.1.

We use the following lemma, the proof of which we defer to the paper [216].

Lemma 6.1. *Let $\hat{k}(v)$ be the coreness estimate and $k(v)$ be the coreness of v , respectively. If $k(v) > (2 + 3/\lambda)(1 + \delta)^{g'}$, then $\hat{k}(v) \geq (1 + \delta)^{g'}$. Otherwise, if $k(v) < \frac{(1 + \delta)^{g'}}{(2 + 3/\lambda)(1 + \delta)}$, then $\hat{k}(v) < (1 + \delta)^{g'}$.*

We show that Lemma 6.1 implies Lemma 6.2.

Lemma 6.2. *The coreness estimate $\hat{k}(v)$ of a vertex v satisfies $\frac{k(v)}{(2 + \varepsilon)} \leq \hat{k}(v) \leq (2 + \varepsilon)k(v)$ for any constant $\varepsilon > 0$.*

Proof. Suppose $\hat{k}(v) = (1 + \delta)^g$. Then, by Lemma 6.1, we have $\frac{(1+\delta)^g}{(2+3/\lambda)(1+\delta)} \leq k(v) \leq (2 + 3/\lambda)(1 + \delta)^{g+1}$. Then, substituting $\hat{k}(v) = (1 + \delta)^g$ and solving the above bounds, $\frac{k(v)}{(2+3/\lambda)(1+\delta)} \leq \hat{k}(v) \leq (2 + 3/\lambda)(1 + \delta)k(v)$. For any constant $\varepsilon > 0$, there exists constants $\lambda, \delta > 0$ where $\frac{k(v)}{2(1+\varepsilon)} \leq \hat{k}(v) \leq 2(1 + \varepsilon)k(v)$. \square

For arbitrary batch sizes, getting better than a 2-approximation for coreness values is P-complete [17], and so there is unlikely to exist a polylogarithmic-span algorithm with such guarantees.

Proof of Theorem 6.2. The approximation factor for our algorithm is given by Lemma 6.2. We defer the proof of the work and span bounds to the paper [216]. Altogether, we prove our main theorem. \square

6.5.5 $O(\alpha)$ Out-Degree Orientation

We orient all edges from vertices in lower levels to higher levels, breaking ties for vertices on the same level by using their indices. Such an orientation can be maintained dynamically in the same work and span as our PLDS via a parallel hash table keyed by the edges and where the values give the orientation. Specifically, we require the following data structures for maintaining a low out-degree orientation. First, we maintain a parallel hash table, H , containing the edges of the graph. The edge (u, v) is the key in the hash table where $u < v$ (i.e. the index of u is less than the index of v). The value for key (u, v) is 0 if the edge is oriented from u to v and 1 if the edge is oriented from v to u . The pseudocode is shown in Algorithm 6.5. Additionally, we make a slight modification to our update algorithm that keeps track of the edges that were searched when a vertex moves to a higher or lower level. The pseudocode for our algorithm is given in Algorithm 6.5.

Proof of Corollary 6.1. Let the degeneracy of the graph be d . As is well-known, the degeneracy of the graph is equal to k_{max} where k_{max} is the maximum k -core of the graph. Furthermore, it is well-known that $\frac{d}{2} \leq \alpha \leq d$. By Lemma 6.2, the vertices in the largest k -core in the graph are in a level with group number at most $\log_{(1+\delta)}((2 + 3/\lambda)(1 + \delta)d) + 1$. This means that the up-degree of each vertex in that group is at most $(2 + 3/\lambda)(1 + \delta)^{\log_{(1+\delta)}((2+3/\lambda)(1+\delta)d)} = (4 + \varepsilon)d$ for any constant $\varepsilon > 0$ for appropriate settings of $\lambda, \delta > 0$. We then also obtain an $(8 + \varepsilon)\alpha$ out-degree orientation where α is the arboricity of the graph. \square

6.6 Experimental Evaluation

In this section, we compare the performance of our dynamic PLDS with existing approaches on a set of large real-world graphs. Our results show that our algorithms consistently achieve speedups, by up to two orders of magnitude, compared with all of the previous state-of-the-art dynamic k -core decomposition algorithms.

Algorithm 6.5 – LowOutdegreeOrient(\mathcal{B})

Require: A batch \mathcal{B} of updates.

Ensure: A set of edges F that were flipped after processing the batch of updates. An edge $(u, v) \in F$ represents the orientation of the edge *before* the flip. Also returns oriented updates $(u, v) \in \mathcal{B}$ where for edge deletions (u, v) is the orientation of the edge *before* the deletion and for edge insertions (u, v) is the orientation of the edge *after* the insertion.

```
1:  $F \leftarrow \emptyset$ .
2: parfor each searched edge  $(u, v)$  for a vertex that moved levels do
3:   if  $H[(u, v)] = 0$  and  $((\ell(u) > \ell(v))$  or  $(\ell(u) = \ell(v) \text{ and } v < u)$  then
4:      $F \leftarrow F \cup (u, v)$ .
5:   else if  $H[(u, v)] = 1$  and  $((\ell(v) > \ell(u))$  or  $(\ell(u) = \ell(v) \text{ and } u < v)$  then
6:      $F \leftarrow F \cup (u, v)$ .
7:  $J \leftarrow \emptyset$ .
8: parfor each edge update  $\{u, v\} \in \mathcal{B}$  do
9:   if  $\{u, v\}$  is an insertion then
10:    Add to  $J$  the orientation of edge after processing  $\mathcal{B}$ .
11:   else
12:    Add to  $J$  the orientation of edge before processing  $\mathcal{B}$ .
return  $F, J$ .
```

Evaluated Algorithms. We evaluate two versions of our algorithm: **PLDS**: an exact implementation of our theoretical algorithm and **PLDSOpt**: a version with $\lceil \log_{1+\delta} n / 50 \rceil$ levels per group. **PLDS** maintains the approximation guarantees given by Lemma 6.2, while **PLDSOpt** achieves better performance while maintaining slightly worse approximation bounds.

We compare our algorithms with the following *dynamic* implementations: **Sun**: the sequential, approximate algorithm of Sun et al. [310], specifically their faster, round-indexing algorithm, which is publicly available [310]; **Hua**: the parallel, exact algorithm of Hua et al. [169], kindly provided by the authors; **Zhang**: the sequential, exact algorithm of Zhang and Yu [351], kindly provided by the authors; and **LDS**: our implementation of the sequential, approximate algorithm of Henzinger et al. [164], but using our coreness approximation procedure in Section 6.5.4. All are state-of-the-art algorithms, outperforming previous algorithms in their respective categories.

We also implemented **ApproxKCore**, our new static parallel approximate k -core decomposition algorithm (Theorem 6.7). We compared it with **ExactKCore**, the state-of-the-art parallel, static, exact k -core algorithm of Dhulipala et al. [95].

Setup. We use c2-standard-60 Google Cloud instances (3.1 GHz Intel Xeon Cascade Lake CPUs with a total of 30 cores with two-way hyper-threading, and 236 GiB RAM) and m1-megamem-96 Google Cloud instances (2.0 GHz Intel Xeon Skylake CPUs with a total of 48 cores with two-way hyper-threading, and 1433.6 GB RAM). We use hyper-threading in our parallel experiments by default. Our programs are written in C++, use a work-stealing scheduler [46], and are compiled using g++ (version 7.5.0) with the -O3 flag. We terminate experiments that take over 3 hours. PLDS and PLDSOpt finished within 3 hours for all experiments.

Datasets. We test our algorithms on 11 real-world undirected graphs from SNAP [207],

| Graph Dataset | Num. Vertices | Num. Edges | Largest value of k |
|----------------------|---------------|---------------|----------------------|
| <i>dblp</i> | 317,080 | 1,049,866 | 101 |
| <i>brain</i> | 784,262 | 267,844,669 | 1200 |
| <i>wiki</i> | 1,094,018 | 2,787,967 | 124 |
| <i>youtube</i> | 1,138,499 | 2,990,443 | 51 |
| <i>stackoverflow</i> | 2,584,164 | 28,183,518 | 163 |
| <i>livejournal</i> | 4,846,609 | 42,851,237 | 329 |
| <i>orkut</i> | 3,072,441 | 117,185,083 | 253 |
| <i>ctr</i> | 14,081,816 | 16,933,413 | 2 |
| <i>usa</i> | 23,947,347 | 28,854,312 | 3 |
| <i>twitter</i> | 41,652,230 | 1,202,513,046 | 2484 |
| <i>friendster</i> | 65,608,366 | 1,806,067,135 | 304 |

Table 6.3: Graph sizes and largest values of k for k -core decomposition.

the DIMACS Shortest Paths challenge road networks [93], and the Network Repository [269], namely *dblp*, *brain*, *wiki*, *orkut*, *friendster*, *stackoverflow*, *usa*, *ctr*, *youtube*, and *livejournal*. We also used *twitter*, a symmetrized version of the Twitter network [200]. We remove duplicate edges, zero-degree vertices, and self-loops. Table 6.3 reflects the graph sizes *after* this removal, and gives the largest k -core values. Both *stackoverflow* and *wiki* are temporal networks; for these, we maintain the edge insertions and deletions in the temporal order from SNAP. *usa* and *ctr* are two high-diameter road networks and *brain* is a highly dense human brain network from NeuroData (<https://neurodata.io/>). All experiments are run on the `c2-standard-60` instances, except for *twitter* and *friendster*, which are run on the `m1-megamem-96` instances as they require more memory.

Ins/Del/Mix Experiments. Our experiments are run for *three different types of batched updates*, referred to by: (1) **Ins**: starting with an empty graph, *all* edges are inserted in multiple size $|\mathcal{B}|$ batches of insertion updates, (2) **Del**: starting with the original graph, *all* edges are deleted in multiple size $|\mathcal{B}|$ batches of deletion updates, and (3) **Mix**: starting with the initial graph minus a random set I of $|\mathcal{B}|/2$ edges, a set D of $|\mathcal{B}|/2$ random edges is chosen among the edges in the graph; then, a single size $|\mathcal{B}|$ mixed batch of updates with insertions I and deletions D is applied. For the temporal graphs, *stackoverflow* and *wiki*, the order of updates in the batches follows the order in SNAP [207]. For the rest, updates are generated by taking two random permutations of the edge list, one for **Ins** and one for **Del**. Batches are generated by taking regular intervals of the permuted lists. For **Mix**, I and D are chosen uniformly at random.

Some past works only ran experiments in the **Mix** setting [169, 351], while others [310] also consider **Ins** and **Del**. In this paper, we run experiments in all three settings. For **Ins** and **Del**, we consider the average running time across all batches as a good indicator of how well the algorithm performs. For **Mix**, we test each algorithm and dataset 3 rounds each and take the average.

We use the original timing functions provided by Hua, Sun, Zhang, and ExactK-

Core. We use the original code of Hua and Zhang for **Mix** and modify their code to perform **Ins** and **Del**. We note that Hua’s timing function does not include the time to process the graph and maintain their data structures; we include all such times in our code. All other benchmarks also include this time. If we include this time in their implementation, their running times increase by up to $8\times$ for some experiments. This explains some of Hua’s experimental performance improvements over the other benchmarks.

The static algorithms, ExactKCore and ApproxKCore, are re-run on the entire graph after each batch of updates in **Ins** and **Del**. For the **Mix** batch, we order all insertions in the batch before all deletions. Then, we generate two static graphs per batch, one following all insertions, and the other following all deletions. We re-run the static algorithms on each static graph and take the average of the times to obtain comparable per-batch running times. We do this because some of the deletion updates may cancel the insertion updates in the batch.

6.6.1 PLDS Implementation Details

We implemented our algorithms using the primitives from the Graph Based Benchmark Suite [98]. We implemented the PLDS with work, span, and space bounds given in Theorem 6.1. One can choose to instead implement our space-efficient version of our data structure in exchange for additional $\text{poly}(\log n)$ factors in the theoretical span.

Our data structure uses concurrent hash tables with linear probing [299], which support x concurrent insertions, deletions, or finds in $O(x)$ amortized work and $O(\log^* x)$ span w.h.p.[148]. For deletions, we used the folklore *tombstone* method: when an element is deleted, we mark the slot in the table as a tombstone, which can be reused, or cleared during a table resize. We also use dynamic arrays, which support adding or deleting x elements from the end in $O(x)$ amortized work and $O(1)$ span.

We first assign each vertex a unique ID in $[n]$. Then, we maintain an array U of size n keyed by vertex ID that returns a parallel hash table containing neighbors of v on levels $\geq \ell(v)$. For each vertex v , we maintain a dynamic array L_v keyed by indices $i \in [0, \ell(v) - 1]$. The i ’th entry of the array contains a pointer to a parallel hash table containing the neighbors of v in level i . Appropriate pointers exist that allow $O(1)$ work to access elements in structures. Furthermore, we maintain a hash table which contains pointers to vertices v where $\text{dl}(v) \neq \ell(v)$, partitioned by their levels. This allows us to quickly determine which vertices to move up (in Algorithm 6.2) or move down (in Algorithm 6.3).

We make one modification in our parallel implementation of our insertion procedure from our theoretical algorithm which is instead of moving vertices up level-by-level, we perform a parallel filter and sort that calculates the desire-level of vertices we move up. This results in more work theoretically, but we find that, practically, it results in faster runtimes. Also, notably, in practice, we optimized the performance of our PLDS by considering $\lceil \frac{\log_{(1+\delta)} m}{50} \rceil$ levels per group instead of $\lceil \log_{(1+\delta)} m \rceil$. We also implemented a version of our structure that *exactly follows* our theoretical algorithm

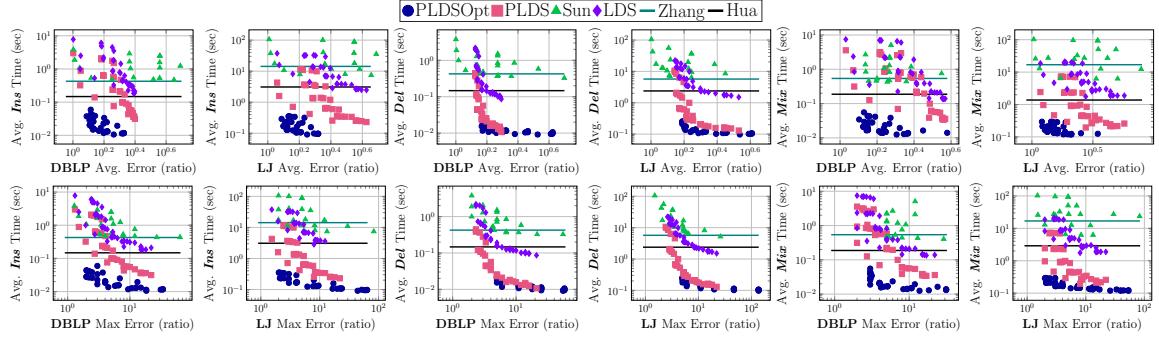


Figure 6.8: Comparison of the average per-batch time versus the average (top row) and maximum (bottom row) per-vertex core estimate error ratio of PLDSOpt, PLDS, Sun, and LDS, using varying parameters, on the *dblp* and *livejournal* graphs, with batch sizes 10^5 and 10^6 , respectively. Experiments were run for **Ins**, **Del**, and **Mix**. The data uses theoretically-efficient parameters as well as the heuristic parameters where $(2 + 3/\lambda) = \alpha_{sun} = 1.1$. Runtimes for Hua and Zhang are shown as horizontal lines.

and compared the performance of both structures. We see that even such a simple optimization resulted in significant gains in performance, up to $23.89\times$.

6.6.2 Accuracy vs. Running Time

We start by evaluating the empirical error ratio of the per-vertex core estimates given by our implementations (PLDSOpt, PLDS, LDS) and Sun on *dblp* and *livejournal*, using batches of size 10^5 and 10^6 , respectively. Fig. 6.8 shows the average batch time (in seconds) against the average and maximum *per-vertex* core estimate error ratio. This error ratio is computed as $\max\left(\frac{\hat{k}(v)}{k(v)}, \frac{k(v)}{\hat{k}(v)}\right)$ for each vertex v (where $\hat{k}(v)$ is the core estimate and $k(v)$ is the exact core value). The average is the error ratio averaged across all vertices and the maximum is the maximum error. If the exact core number is 0, we ignore the vertex in our error ratio since our algorithm guarantees an estimate of 0; for vertices of non-zero degree, the lowest estimated core number is 1 for all implementations.

The parameters we use for PLDSOpt, PLDS, and LDS are all combinations of $\delta = \{0.2, 0.4, 0.8, 1.6, 3.2, 6.4\}$ and $\lambda = \{3, 6, 12, 24, 48, 96\}$. We call these *theoretically-efficient parameters*, since they maintain the work-efficiency of our algorithms. For Sun, we use all combinations of their parameters $\varepsilon_{sun} = \lambda_{sun} = \{0.2, 0.4, 0.8, 1.6, 3.2\}$, and $\alpha_{sun} = \{2(1 + 3\varepsilon_{sun})\}$. We also tested $\alpha_{sun} = \{1.1, 2, 3.2\}$, as done in Sun et al.'s work [310]. When $\alpha = 1.1$, the theoretical efficiency bounds by Sun et al. [310] no longer hold, but they yield better estimates empirically. We compare this heuristic setting to a similar one in our algorithms, where we replace $(2 + 3/\lambda)$ with 1.1 in our code (where our efficiency bounds no longer hold) for $\delta = \{0.4, 0.8, 1.6, 3.2\}$. We refer to these as the *heuristic parameters*.

Fig. 6.8 shows that, using theoretically-efficient parameters, our PLDSOpt, PLDS,

and LDS implementations are faster than Sun, Zhang, and Hua, for parameters that give similar average and maximum per-vertex core estimate error ratios. Furthermore, besides PLDS, PLDSOpt *outperforms all other algorithms*, regardless of approximation factor and error. This set of experiments demonstrates the flexibility of our algorithm; one can achieve smaller error at the cost of slightly increased runtime. However, as the experiments demonstrate, PLDSOpt still outperforms all other algorithms even when the parameters are tuned to give small error; this performance gain is maintained for **Ins**, **Del**, and **Mix**. Greater speedups are achieved on *livejournal* compared to *dblp*. Such a result is expected since larger batches allow for greater parallelism.

Concretely, compared with Zhang, PLDSOpt achieves $7.19\text{--}147.59\times$, $19.70\text{--}58.41\times$, and $9.75\text{--}142.79\times$ speedups on **Ins**, **Del**, and **Mix** batches, respectively. Compared with Hua, PLDSOpt achieves $2.49\text{--}33.95\times$, $6.81\text{--}24.51\times$, and $2.94\text{--}21.77\times$ speedups. Against PLDS, PLDSOpt obtains $2.98\text{--}47.8\times$, $1.03\text{--}25.58\times$, and $1.5\text{--}76.94\times$ speedups for **Ins**, **Del**, and **Mix**, respectively, on parameters that give similar approximations. Compared with Sun, on parameters that give similar theoretical guarantees and smaller empirical average error, PLDSOpt achieves $21.34\text{--}544.22\times$, $25.49\text{--}128.65\times$, and $19.04\text{--}248.36\times$ speedups for **Ins**, **Del**, and **Mix**, respectively. Neither Zhang nor Hua guarantee polylogarithmic work. The peeling-based algorithm of Sun can have large span and they do not provide a concrete bound on their amortized work for their faster, round-indexing implementation. Thus, the speedups we obtain over the benchmarks are due to the greater theoretical efficiency and because our algorithms are parallel.

Finally, PLDSOpt achieves average error in the ranges $1.26\text{--}2.13$, $1.47\text{--}4.20$, and $1.28\text{--}2.33$ for **Ins**, **Del**, and **Mix**, respectively. PLDS gives comparable average errors in the ranges $1.27\text{--}4.22$, $1.33\text{--}3.39$, and $1.63\text{--}5.73$, for **Ins**, **Del**, and **Mix**, respectively, while running slower than PLDSOpt for all parameters, despite the guarantee that the maximum error of PLDS is bounded by $(1 + \delta)(2 + 3/\lambda)$ (Lemma 6.2). Thus, our optimized version allows us to obtain good error bounds empirically while drastically improving performance.

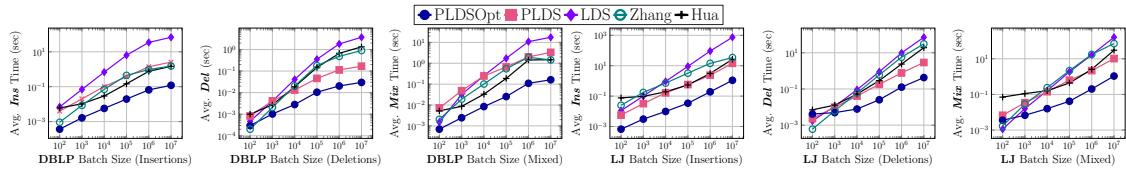


Figure 6.9: Average **Ins**, **Del**, and **Mix** per-batch running times on varying batch sizes for PLDSOpt, PLDS, LDS, Zhang, and Hua on *dblp* and *livejournal*.

For all of the remaining experiments, set $\delta = 0.4$ and $\lambda = 3$.

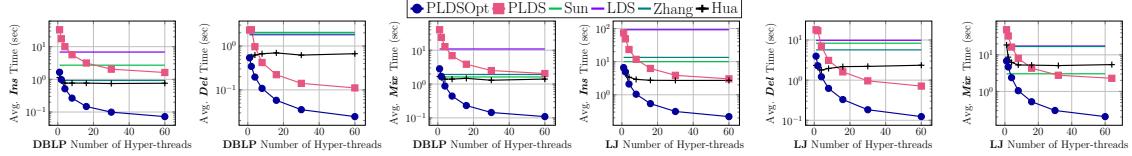


Figure 6.10: Parallel speedup of PLDSOpt, PLDS, and Hua, with respect to their single-threaded running times on *dblp* and *livejournal* on **Ins**, **Del**, and **Mix** batches of size 10^6 for all algorithms. The “60” on the *x*-axis indicates 30 cores with hyper-threading. LDS, Sun, and Zhang are shown as horizontal lines since they are sequential.

6.6.3 Batch Size vs. Running Time

Fig. 6.9 shows the average per-batch running times for **Ins**, **Del**, and **Mix** on varying batch sizes for PLDSOpt, PLDS, Hua, LDS, and Zhang on *dblp* and *livejournal*. We do not run this experiment on Sun since their implementation does not have batching. Our experiments show that PLDSOpt is faster for all batch sizes except for the smallest **Del** and **Mix** batches.

Against PLDS, PLDSOpt achieves a speedup over all batches from $10.85\text{--}21.25\times$, $2.81\text{--}5.65\times$, and $10.42\text{--}29.28\times$ for **Ins**, **Del**, and **Mix**, respectively, on *dblp* and *livejournal* for all but the batch of size 100 for **Del**. On the batch size of 100, PLDS performs better than PLDSOpt by a $1.79\times$ factor. Compared with Hua, PLDSOpt achieves speedups over all batches from $5.17\text{--}16.43\times$, $3.39\text{--}44.58\times$, and $2.53\text{--}13.05\times$ for **Ins**, **Del**, and **Mix**, respectively, on *dblp* and $15.97\text{--}114.52\times$, $1.71\text{--}45.01\times$, and $9.10\text{--}19.82\times$ for **Ins**, **Del**, and **Mix**, respectively, on *livejournal*. Compared with Zhang, PLDSOpt achieves speedups of $2.49\text{--}22.74\times$, $2.00\text{--}29.92\times$, and $2.95\text{--}21.57\times$ for **Ins**, **Del**, and **Mix**, respectively, on *dblp*, and $31.53\text{--}95.33\times$, $1.25\text{--}73.19\times$ and $4.26\text{--}87.05\times$ for **Ins**, **Del**, and **Mix**, respectively, on *livejournal* on all but the smallest batches for **Del** and **Mix**. For **Del** with a batch size of 100, Zhang is the fastest with speedups of $1.46\times$ and $6.86\times$ over PLDSOpt on *dblp* and *livejournal*, respectively. For **Mix** with batch size 100, LDS is the fastest with speedups of $3.19\times$ over PLDSOpt on *livejournal*. For small batch sizes, sequential algorithms perform better than parallel algorithms since the runtimes of parallel algorithms are dominated by parallel overheads.

6.6.4 Thread Count vs. Running Time

Fig. 6.10 shows the scalability of PLDSOpt, PLDS, and Hua with respect to their single-thread running times on *dblp* and *livejournal* using a batch size of 10^6 . LDS, Sun, and Zhang are represented as horizontal lines since they are sequential. For **Ins**, **Del**, and **Mix** batches, PLDSOpt and PLDS achieve up to $30.28\times$, $32.02\times$, and $33.02\times$, and $26.46\times$, $25.33\times$, and $21.15\times$, self-relative speedup, respectively. Hua achieves up to a $3.6\times$ self-relative speedup. We see that our PLDS algorithms

achieve greater self-relative speedups than Hua. Also, with just 4 threads (available on a standard laptop), PLDSOpt already outperforms all other algorithms. Hua’s algorithm performs DFS/BFS, which could lead to linear span, potentially explaining the bottleneck to their scalability with more cores.

Gabert et al. [139] present a parallel batch-dynamic k -core decomposition algorithm but their code is proprietary. However, their algorithm appears slower and less scalable based on their paper’s stated results. For example, their algorithm on 10^5 edges using 32 threads for the *livejournal* graph requires 4 seconds, while our algorithm on a batch of 10^6 edges using 30 threads (more edges and fewer threads) requires a *maximum* of 0.35 seconds. Also, they appear to exhibit a maximum of $8\times$ self-relative speedup on *livejournal* while we exhibit $21.2\times$ self-relative speedup on *livejournal*.

6.6.5 Results on Large Graphs

Fig. 6.11 shows the runtimes of PLDSOpt, PLDS, Hua, Sun, and Zhang compared with the static algorithms ExactKCore and ApproxKCore on additional graphs, using **Ins**, **Del**, and **Mix** batches, all of size 10^6 . ExactKCore and ApproxKCore are run from scratch over the entire graph after every batch since they do not handle batch updates. PLDSOpt and PLDS finished for all graphs and experiments while all other algorithms timed out on **Ins** and **Del** batches for *twitter* and *friendster*. Zhang was able to finish on **Mix** because their indexing algorithm (used to create their data structures provided the initial graph without the mixed batch) was able to finish; since only one mixed batch is used to update the graph, the sum of the time needed for indexing plus the update time of one batch fell under the timeout. The same is true for ExactKCore and ApproxKCore. However, these algorithms were not able to finish for **Ins** and **Del** because the sum of the update times across all batches is too high.

PLDSOpt is faster than all other dynamic algorithms on all types of batches, except for PLDS on *ctr* and *usa*. We report concrete speedups for experiments which finished within the timeout. For **Ins**, it gets $10.01\text{--}229.71\times$ speedups over Zhang, $6.20\text{--}58.66\times$ speedups over Hua, $26.02\text{--}119.77\times$ speedups over Sun, and $1.45\text{--}23.89\times$ speedups over PLDS. For **Del**, it gets $30\text{--}176.48\times$ speedups over Zhang, $15.79\text{--}52.36\times$ speedups over Hua, $41.02\text{--}100.34\times$ speedups over Sun, and $2.51\text{--}23.45\times$ speedups over PLDS (except on *ctr* and *usa*). For **Mix**, it gets $17.54\text{--}723.72\times$ speedups over Zhang, $11.34\text{--}91.95\times$ over Hua, $6.95\text{--}35.59\times$ speedups over Sun, and $2.81\text{--}18.68\times$ speedups over PLDS (except on *ctr* and *usa*). These massive speedups over previous work demonstrate the utility of PLDSOpt not only on large graphs but also on smaller graphs. Notably, our PLDSOpt and PLDS algorithms perform not only well on dense networks but also on very sparse road networks. For *ctr* and *usa*, PLDS performs better than PLDSOpt, achieving up to a $1.09\times$ speedup on **Del** and $1.12\times$ speedup on **Mix**.

Compared to the static algorithms, PLDSOpt achieves speedups for all but the smallest graphs, *dblp*, *wiki*, and *youtube*. For these graphs, the batch of size 10^6 accounts for more than $1/3$ of the edges, and so even if the static algorithm reprocesses

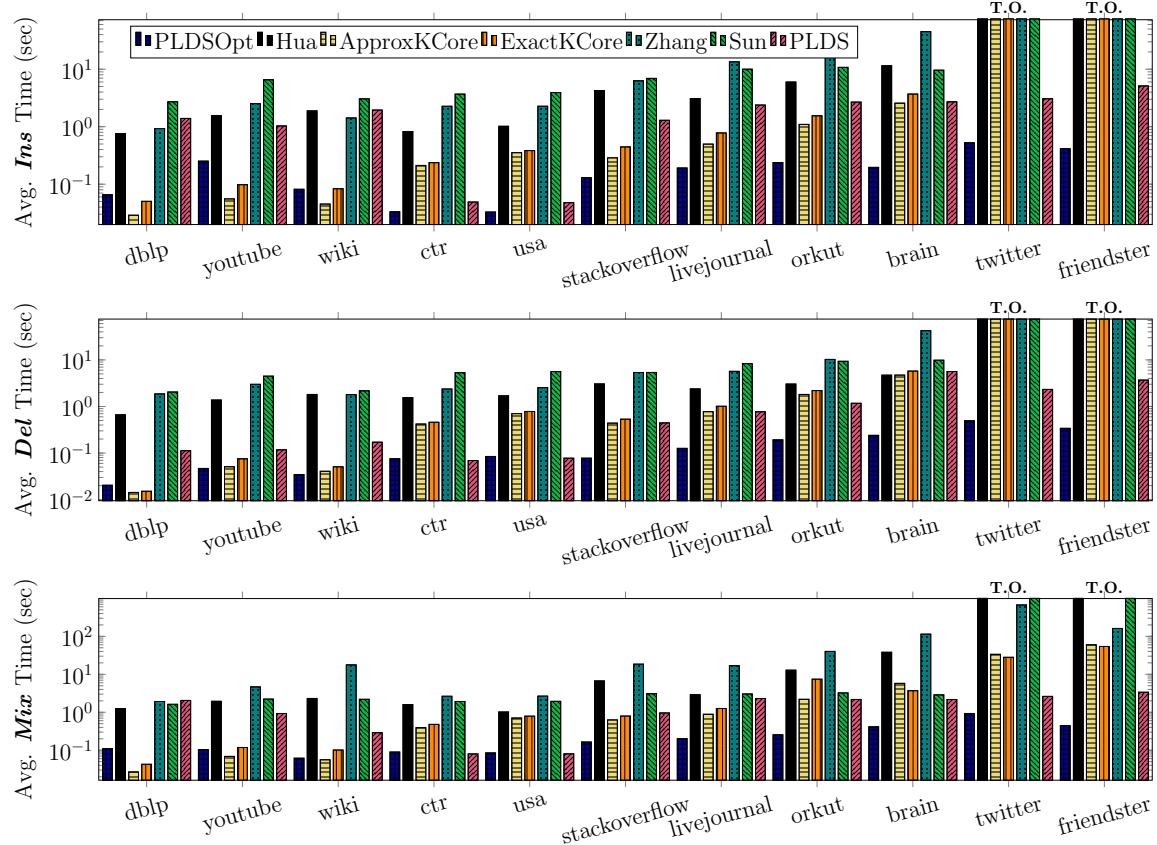


Figure 6.11: Average per-batch running times for PLDSOpt, Hua, PLDS, Sun, Zhang, ApproxKCore, and ExactKCore, on *dblp*, *youtube*, *wiki*, *ctr*, *usa*, *stackoverflow*, *livejournal*, *orkut*, *brain*, *twitter*, and *friendster* with batches of size 10^6 (and approximation settings $\delta = 0.4$ and $\lambda = 3$ for PLDSOpt and PLDS). All benchmarks (except PLDSOpt and PLDS) timed out (T.O.) at 3 hours for **Ins** and **Del**. Hua and Sun timed out on *twitter* and *friendster* for **Mix**. The top graph shows insertion-only, middle graph shows deletion-only, and bottom graph shows mixed batch runtimes.

the entire graph per batch, it does not process many more edges past the batch size. Thus, it is expected that the parallel static algorithms perform better on small graphs and large batches. For all but the smallest graphs, PLDSOpt obtains 2.22–13.09 \times , 5.56–19.64 \times , and 4.4–121.76 \times speedups over the *fastest* static algorithm for each graph for **Ins**, **Del**, and **Mix**, respectively. ExactKCore and ApproxKCore both timeout for **Ins** and **Del** on *twitter* and *friendster*; otherwise, we expect to see the large improvements that we see for **Mix** on these experiments.

6.6.6 Accuracy of Approximation Algorithms

We also computed the average and maximum errors of all of our approximation algorithms for our experiments shown in Fig. 6.11. According to our theoretical proofs, the maximum error (for PLDS) should be $(2 + 3/3)(1 + 0.4) = 4.2$. We confirm that the maximum empirical error for PLDS falls under this constraint. PLDSOpt achieves an average error of 1.24–2.37 compared to errors of 1.26–3.48 for PLDS, 1.01–4.17 for ApproxKCore, and 1.03–3.23 for Sun. PLDSOpt gets a maximum error of 3–6 compared to 2–4.19 for PLDS, 3–5 for ApproxKCore, and 3–5.99 for Sun. We conclude that our error bounds match those of the current best-known algorithms and are sufficiently small to be of use for many applications.

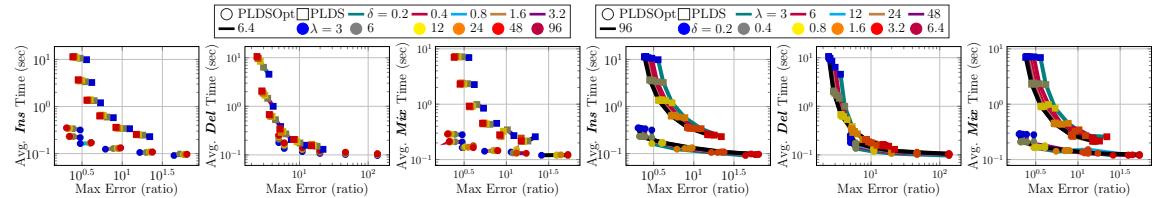


Figure 6.12: Sensitivity analysis of PLDSOpt and PLDS on *livejournal*. The first three plots fix δ ; each line is a fixed δ value and each point is a different λ value. The last three plots fix λ and vary δ .

6.6.7 Sensitivity of PLDS and PLDSOpt to δ and λ

In Fig. 6.12, we provide a sensitivity analysis for the parameters δ and λ on the *maximum* error of our PLDS and PLDSOpt algorithms since our theoretical guarantees are for the *maximum error* and as we showed in Section 6.6.2, the average error does not vary significantly for our chosen set of parameters. The first three graphs of Fig. 6.12 shows the effect of fixing δ while varying λ and the last three show the opposite.

We see that for both PLDSOpt and PLDS, different λ values do not affect either the error by much (each line is essentially a cluster of points). This matches what we expect theoretically. Recall our bound on error, $(1 + \delta)(2 + 3/\lambda)$; suppose we set $\delta = 0.4$ and $\lambda = 3$ as in our experiments. This leads to an upper bound of 4.2. If we increase λ to 6, this only decreases the error to 3.5. On the other hand, if δ is increased to 0.8, then the error increases to 5.4, resulting in greater sensitivity to δ .

However, *increasing* δ leads to a drastic decrease in running time (each line is a decreasing curve) at the expense of a large increase in error. Again, this matches what we expect theoretically, since δ affects the number of levels in PLDS and PLDSOpt (recall that in our algorithm, the number of levels per group is $\lceil \log_{(1+\delta)}(m) \rceil$). A larger number of levels leads to larger running time and we see this in our results. We do not see as large an increase for PLDSOpt since we divide the number of levels by 50. This means that for *livejournal* the number of levels per group is $\lceil \log_{(1+\delta)}(42851237)/50 \rceil = 1$ for all $\delta < 0.42$. We see this in our experiments as the curves for PLDSOpt are flat for $\delta \in [0.8, 6.4]$.

For the rest of the experiments, we fix $\delta = 0.4$ and $\lambda = 3$ based on our sensitivity analysis; these parameters offer a reasonable tradeoff between approximation error and speed, as shown in Fig. 6.8 and Fig. 6.12. For Sun, we choose the parameters $\varepsilon = \lambda = 2$ and $\alpha = 2$ since we observe these parameters give similar approximation errors to the parameters that we chose for our algorithms.

6.6.8 Space Usage

For each program, we implemented functions that measured the space usage of the data structures used in the algorithms (specifically, the private and public variables maintained in their data structure classes); for all of the algorithms, we do not count ephemeral space usage needed by auxiliary structures that are not maintained as either private or public variables of their data structure class. For this set of experiments, we only test on **Ins** and **Del** since maximum space is used when the entire graph is present in memory.

Fig. 6.13 shows the results of our space-bound experiments. Although PLDS uses more memory than most other implementations, our PLDSOpt uses less memory than Hua and Zhang in most settings (up to $1.34\times$ factor less memory than the minimum space used by either) for *dblp* and up to $1.08\times$ additional space in a few cases; for *livejournal*, it uses up to $1.72\times$ additional space compared to the minimum space used by Hua and Zhang. Sun uses more space than PLDSOpt for most cases; although for a few parameters for deletions in *dblp*, it uses up to $1.9\times$ less space. Since we have a $O(\log^2 n)$ factor in our space usage bound, we expect a slight increase in our space usage compared to algorithms with linear space; however, as we demonstrated, empirically our space usage is not much greater, and we believe that this small extra space usage is a small price to pay for the large improvement in performance obtained by our algorithms. We provide theoretical space-efficient implementations of our PLDS which may also prove to be more space-efficient in practice.

6.7 Static $(2 + \varepsilon)$ -Approximate k -Core

Due to the P-completeness of k -core decomposition for $k \geq 3$ [17], all known static exact k -core algorithms do not achieve polylogarithmic span. We introduce a linear work and polylogarithmic span $(2 + \varepsilon')$ -approximate k -core decomposition algorithm (with only one-sided error) based on the parallel bucketing-based peeling algorithm for

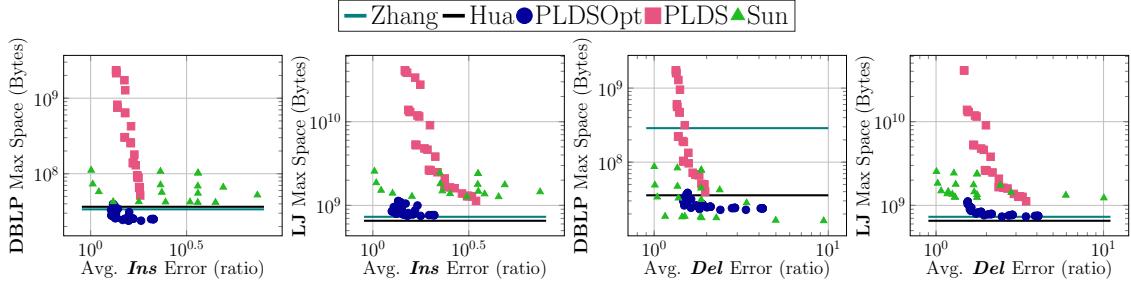


Figure 6.13: Maximum space usage in bytes for PLDSOpt, Hua, Zhang, PLDS, and Sun in terms of the average error. We varied δ and λ and computed the error ratio and space usage for the programs on *dblp* and *livejournal*. We tested against **Ins** and **Del** batches of size 10^5 for *dblp* and batches of size 10^6 for *livejournal*.

static *exact k*-core decomposition of Dhalipala et al. [95]. The algorithm maintains a mapping M from $v \in V$ to a set of *buckets*, with the bucket for a vertex $M(v)$ changing over the course of the algorithm. The algorithm starts at $k = 0$, peels all vertices with degree at most $(2 + \varepsilon)(1 + \varepsilon)^k$ where ε is set to $\frac{\sqrt{4\varepsilon' + 9} - 3}{2}$, increments k , and repeats until the graph becomes empty. The approximate core value of v is $(1 + \varepsilon)^{k-1}$ where we use the value of k when v is peeled. We observe that the dynamic algorithm in this paper can be combined with a peeling algorithm like the above to yield a linear-work approximate *k*-core algorithm with polylogarithmic span.

Algorithm 6.6 shows pseudocode for our approximate *k*-core algorithm, which computes an approximate coreness value for each vertex. The algorithm sets the initial coreness estimates, $C[v]$, of each vertex to its degree (Line 1). Then, it maintains a parallel bucketing data structure M , which maps each vertex to the $\lceil \log_{1+\varepsilon} C[v] \rceil$ 'th bucket (Line 3). It initializes a variable $finished = 0$ to keep track of the number of vertices peeled and a variable $t = 0$ used to compute the approximate core values (Line 2). The rest of the algorithm performs peeling, where the peeling thresholds are powers of $(1 + \varepsilon)$. The peeling loop (Line 4–Line 20) first extracts the lowest non-empty bucket from M (Line 5), which consists of I , a set of vertex IDs of vertices that are being peeled, and the bucket number bkt . If more than $\log_{1+\delta}(n)$ rounds of peeling have occurred at the threshold $(2 + \varepsilon)(1 + \varepsilon)^t$ (where we set $\delta = \frac{2}{\varepsilon}$), the algorithm increments t (Line 6). Next, the algorithm computes in parallel an array R of pairs (v, r_v) , where v is a neighbor of some vertex in I and r_v is the number of neighbors of v in I (Line 8). Finally, the algorithm computes in parallel the new buckets for the affected neighbors v (Line 10–Line 14). The coreness estimate is updated to the maximum of the peeling threshold of the previous level and the current induced degree of v after r_v of its neighbors are removed. Finally, the algorithm updates the buckets using the new coreness estimates for the updated vertices (Line 15), which can be done in parallel using our bucketing data structure.

We provide an example of this algorithm below.

Example 6.8. Fig. 6.14 shows a run of Algorithm 6.6 on an example graph. Given

Algorithm 6.6 – Static Approximate k -core Decomposition

Require: An undirected graph $G(V, E)$.
Ensure: An array of $(2 + \varepsilon')$ -approximate coreness values for any constant $\varepsilon' > 0$.

```

1:  $\forall v \in V$ , let  $C[v] = |N(v)|$ .
2:  $finished \leftarrow 0, t \leftarrow 0, \varepsilon \leftarrow \frac{\sqrt{4\varepsilon'+9}-3}{2}, \delta \leftarrow \frac{2}{\varepsilon}$ .
3: Let  $M$  be a bucketing structure formed by initially assigning each  $v \in V$  to the  $\lceil \log_{1+\varepsilon} C[v] \rceil$ 'th bucket.
4: while ( $finished < |V|$ ) do
5:    $(I, bkt) \leftarrow$  Vertex IDs and bucket ID of next (peeled) bucket in  $M$ .
6:    $t \leftarrow bkt$ .
7:   for iteration  $j \in [\lceil \log_{1+\delta}(n) \rceil]$  do
8:      $R \leftarrow \{(v, r_v) \mid v \in N(I), r_v = |\{(u, v) \in E \mid u \in I\}|\}$ .
9:      $U \leftarrow$  Array of length  $|R|$ .
10:    parfor  $R[i] = (v, r_v), i \in [0, |R|]$  do
11:       $inducedDeg = C[v] - r_v$ 
12:       $C[v] = \max(inducedDeg, \lceil (1 + \varepsilon)^{t-1} \rceil)$ 
13:       $newbkt = \max(\lceil \log_{1+\varepsilon} C[v] \rceil, t)$ 
14:       $U[i] = (v, newbkt)$ 
15:    Update  $M$  for each  $(u, newbkt)$  in  $U$ .
16:     $next-bkt \leftarrow$  bucket ID of the next smallest bucket in  $M$ .
17:    if  $(1 + \varepsilon)^{next-bkt} \leq (2 + \varepsilon)(1 + \varepsilon)^t$  then
18:       $(I, next-bkt) \leftarrow$  Vertex IDs of the next (peeled) bucket in  $M$ .
19:    else
20:      break
21: return  $C$ .

```

the parameters $\varepsilon = \delta = 1$, the two buckets that the vertices of the input graph (shown in (a)) are partitioned into are bucket index 1 (green vertices) and bucket index 2 (purple vertices). Vertices v, w, a , and b have degree 2 so they are put into the bucket with index $\lceil \log_2(2) \rceil = 1$. Since u, x, y , and z have degree ≥ 3 , they are put into the bucket with index $\lceil \log_2(3) \rceil = 2$.

Since the bucket with index 1 has the smaller bucket index, we peel off all the vertices in that bucket (the green vertices) and we assign the core estimate of $(1 + \varepsilon)^1 = 2$ to all vertices in that bucket (shown in (b)). We update the buckets of all neighbors of the peeled vertices; however, since u, x, y , and z all still have degree ≥ 3 , they remain in the bucket with index 2. Finally, we peel bucket index 2 and assign all vertices in that bucket an estimate of $(1 + \varepsilon)^2 = 4$ (shown in (c)). In this example, the estimates produced are 3-approximations of the real coreness values.

We prove below that Algorithm 6.6 finds an $(2 + \varepsilon)$ -approximate k -core decomposition in $O(m)$ expected work and $O(\log^3 m)$ span w.h.p. using $O(m)$ space, as stated in Theorem 6.7. We give the approximation guarantees of our algorithm using lemmas from [144], and use an efficient parallel semisort implementation [154] for our work bounds.

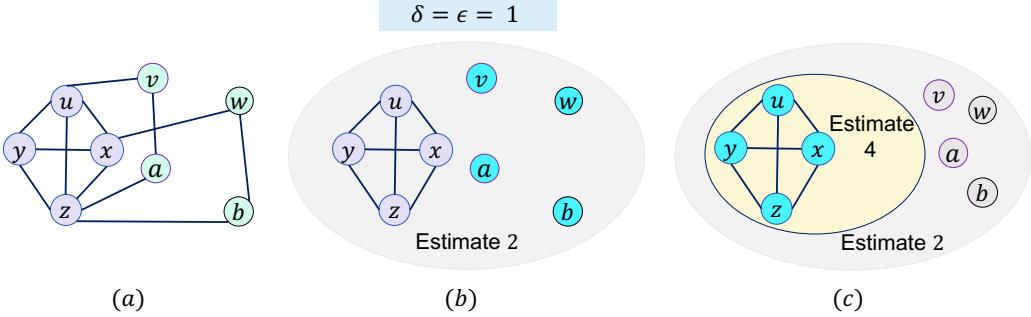


Figure 6.14: Example of a run of Algorithm 6.6 described in Theorem 6.8.

Theorem 6.9. *For a graph with m edges,¹⁰ for any constant $\varepsilon > 0$, there is an algorithm that finds an $(2 + \varepsilon)$ -approximate k -core decomposition in $O(m)$ expected work and $O(\log^3 m)$ span with high probability, using $O(m)$ space.*

Proof. Our approximation guarantee is given by Observation 4 of [144]. Using Observation 4, the number of vertices with core number $(1 + \varepsilon)^t$ after one round of peeling in Algorithm 6.6 shrinks by a factor of $\frac{2}{2 + \varepsilon}$. Let $V_{\leq(1+\varepsilon)^t}$ be the number of vertices with core number at most $(1 + \varepsilon)^t$. After removing all vertices with degree at most $(2 + \varepsilon)(1 + \varepsilon)^t$, the number of vertices with core number $(1 + \varepsilon)^t$ and with degree greater than $(2 + \varepsilon)(1 + \varepsilon)^t$ is at most $\frac{2(1+\varepsilon)^t |V_{\leq(1+\varepsilon)^t}|}{(2+\varepsilon)(1+\varepsilon)^t} = \frac{2|V_{\leq(1+\varepsilon)^t}|}{2+\varepsilon}$. Since $|V_{\leq(1+\varepsilon)^t}| \leq n$, the maximum number of rounds needed to peel all vertices with core number at most $(1 + \varepsilon)^t$ is $\log_{(2+\varepsilon)/2}(n)$. By induction on t (Line 6), after $\log_{(2+\varepsilon)/2}(n)$ rounds, all vertices with core number at most $(1 + \varepsilon)^t$ are removed. Hence, in round $t+1$, all vertices have core number greater than $(1 + \varepsilon)^t$ and have core number at most $(2 + \varepsilon)(1 + \varepsilon)^{t+1}$; hence, we obtain a $(2 + \varepsilon)(1 + \varepsilon) = 2 + \varepsilon'$ approximation (for any constant $\varepsilon' > 0$ and appropriate setting of ε) when we give coreness approximations of $(1 + \varepsilon)^t$ to all vertices peeled for $t + 1$.

Our algorithm uses a number of data structures that we use to obtain our work, span, and space bounds. Our parallel bucketing data structure (Line 3) can be maintained via a sparse set (hash map), or by using the bucketing data structure from [95]. The outer loop iterates for $O(\log n)$ times (Line 4). Within each iteration of the outer loop, we iterate for $O(\log_{(1+\delta)} n) = O(\log n)$ rounds for constant $\delta = \frac{2}{\varepsilon}$. After obtaining a set of vertices, we update the buckets using semisort in $O(\log n)$ span w.h.p. [95]. Thus the overall span of the algorithm is $O(\log^3 m)$ for any constant $\delta > 0$.

The work of the algorithm can be bounded as follows. We charge the work for moving a vertex from its current bucket to a lower bucket within a given round to one of the edges that was peeled from the vertex in the round. Thus the total number of bucket moves done by the algorithm is $O(m)$. Each round of the algorithm also peels a number of edges and aggregates, for each vertex that has a neighbor in the current bucket, the number of edges incident to this vertex that are peeled (the r_v variable in the algorithm). We implement this step using a randomized semisort [154]. Since

¹⁰Our bounds in this paper assume $m = \Omega(n)$ for simplicity, although our algorithms work even if $m = o(n)$.

$2m$ edges are peeled in total, the overall work is $O(m)$ in expectation.

Lastly, we bound the space used by the algorithm. There are a total of $O(\log_{1+\varepsilon} n) = O(\log n)$ buckets for any constant $\varepsilon > 0$. Each vertex appears in exactly one bucket, and thus the overall space of the bucketing structure is $O(n)$. The algorithm also semisorts the edges peeled from the graph in each step. Since all m edges could be peeled and removed within a single step, and thus semisorted the overall space used by the algorithm is $O(m)$. \square

The approximation guarantees provided by our algorithm are essentially the best possible, under widely believed conjectures. Specifically, Anderson and Mayr [17] show that the optimization version of the High-Degree Subgraph problem, namely to compute the largest core number, or *degeneracy* of a graph cannot be done better than a factor of 2. Thus, obtaining a polynomial work and polylogarithmic span $(2 - \varepsilon)$ -approximation to the coreness value of each vertex would yield a $(2 - \varepsilon)$ -approximation to the optimization version of the High-Degree Subgraph problem, and show that $P = NC$, contradicting a widely-believed conjecture in parallel complexity theory.

In recent years, several results have given parallel algorithms that obtain a $(1 + \varepsilon)$ -approximation to the coreness values in distributed models of computation such as the Massively Parallel Computation model [124, 144]. These results work by performing a *random sparsification* of the graph into a subgraph that approximately preserves the coreness values. They then send this subgraph to a single machine, which runs the sequential peeling algorithm on the subgraph to find approximate coreness values. Crucially, this second peeling step on a single machine can have $\Theta(n)$ span, and thus, this approach does not yield a polylogarithmic span algorithm in the work-span model of computation.

6.8 Framework for Batch-Dynamic Graph Algorithms from Low Out-Degree Orientations

In this section, we introduce a framework that we will use in all of our batch-dynamic algorithms that use our batch-dynamic low out-degree orientation algorithm (Section 6.5.4). Our framework assumes three different methods for each of the problems (maximal matching, k -clique counting, and vertex coloring) that we solve. Specifically, these three methods handle batches of insertions and deletions separately; let **BatchFlips**, **BatchInsert**, and **BatchDelete** denote these three methods.

We assume for simplicity that all updates in the batch \mathcal{B} are *unique*, which means that no edge deletion occurs on an inserted edge in the batch and vice versa. Furthermore, we assume that the updates are *valid*, meaning that if an edge insertion (u, v) is in \mathcal{B} , then (u, v) does not exist in the graph, and if an edge deletion (w, x) is in \mathcal{B} , then edge (w, x) exists in the graph. Such assumptions are only *simplifying* assumptions because it is easy to perform preprocessing on \mathcal{B} in $O(|\mathcal{B}| \log n)$ work and $O(\log n)$ span to ensure that these assumptions are satisfied. In fact, our implementations in Section 6.6 do perform this preprocessing on the input batches. To

Algorithm 6.7 – GraphProblemUpdate(G, \mathcal{B})

Require: A graph $G = (V, E)$ and a batch \mathcal{B} of unique and valid updates.

Ensure: A solution to the relevant graph problem.

- 1: $\text{Update}(\mathcal{B})$ [Algorithm 6.1].
 - 2: $A \leftarrow \text{LowOutdegreeOrient}(\mathcal{B})$.
 - 3: Perform parallel filter on \mathcal{B} to obtain a batch of insertions, \mathcal{B}_{ins} , and a batch of deletions, \mathcal{B}_{del} .
 - 4: $\text{BatchFlips}(A, \mathcal{B}_{ins}, \mathcal{B}_{del})$.
 - 5: $\text{BatchDelete}(\mathcal{B}_{del})$.
 - 6: $\text{BatchInsert}(\mathcal{B}_{ins})$.
-

find all unique updates, we perform a parallel sort in $O(|\mathcal{B}| \log n)$ work and $O(\log n)$ span [177, 49, 98]; we first sort on the edge and then the timestamp of the update. Then, we perform a parallel filter in $O(|\mathcal{B}|)$ work and $O(1)$ span [177, 49, 98] where we keep each edge with the latest timestamp. Then, we perform another parallel filter to keep only edge insertions of nonexistent edges and edge deletions of edges that exist in the graph. This preprocessing ensures \mathcal{B} follows our simplifying assumptions and do not exceed the complexity bounds of our PLDS, and hence, we assume all input batches contain unique and valid updates. The work and span for preprocessing are subsumed by the bounds for the algorithms.

Detailed Framework The pseudocode for our framework is shown in Algorithm 6.7. We first update the PLDS by calling the update procedure (Algorithm 6.1) on the batch of updates in Line 1. Afterwards, we call our low out-degree orientation algorithm to obtain the set of edges that were flipped, placed in set A (Line 2). Then, we take the batch of updates \mathcal{B} and split the batch into a batch of insertions, \mathcal{B}_{ins} , and a batch of deletions, \mathcal{B}_{del} (Line 3). We call BatchFlips (Line 4) on the set of flipped edges A , which processes the edge flips accordingly for each problem. Finally, we call the problem specific functions BatchDelete and BatchInsert (Lines 5 and 6) on \mathcal{B}_{del} and \mathcal{B}_{ins} , respectively; we first call BatchDelete and then BatchInsert .

Analysis By Corollary 6.1, our low out-degree orientation algorithm gives a $O(\alpha)$ out-degree orientation. Furthermore, the amortized work of the algorithm indicates that $O(|\mathcal{B}| \log^2 n)$ amortized flips occur with each batch \mathcal{B} . Suppose that $\text{BatchFlips}(A)$ takes $O(|A|W_{flips}(\alpha))$ work and $O(D_{flips})$ span; $\text{BatchInsert}(\mathcal{B}_{ins})$ takes $O(|\mathcal{B}_{ins}|W_{ins}(\alpha))$ work and $O(D_{ins})$ span, and $\text{BatchDelete}(\mathcal{B}_{del})$ takes $O(|\mathcal{B}_{del}|W_{del}(\alpha))$ work and $O(D_{del})$ span; and the update methods require $O(S)$ space in total. Then, we show the following theorem about our framework.

Theorem 6.10. *Algorithm 6.7 takes*

$$O(|\mathcal{B}|W_{flips}(\alpha) \log^2 n + |\mathcal{B}|W_{ins}(\alpha) + |\mathcal{B}|W_{del}(\alpha))$$

amortized work and

$$O(\log^2 n \log \log n + D_{\text{flips}} + D_{\text{ins}} + D_{\text{del}})$$

span w.h.p. in $O(n \log^2 n + m + S)$ space.

Proof. Theorem 6.2 states that updating the PLDS and getting the flipped edges require $O(|\mathcal{B}| \log^2 n)$ amortized work, $O(\log^2 n \log \log n)$ span, and $O(n \log^2 n + m)$ space. Since the calls to the procedures are independent and sequential, the total work, span, and space equal the sum of the work, span, and space of our PLDS algorithm and **BatchFlips**, **BatchDelete** and **BatchInsert**.

Then, the only additional information we need are the sizes of \mathcal{B}_{ins} and \mathcal{B}_{del} . By our algorithm, $|\mathcal{B}_{\text{ins}}|, |\mathcal{B}_{\text{del}}| \leq |\mathcal{B}|$ since $\mathcal{B}_{\text{ins}} \cup \mathcal{B}_{\text{del}} = \mathcal{B}$. By Theorem 6.2, A has $O(|\mathcal{B}| \log^2 n)$ amortized flips; thus, the amortized work of **BatchFlips** is $O(|\mathcal{B}| W_{\text{flips}}(\alpha) \log^2 n)$. Finally, the PLDS uses $O(n \log^2 n + m)$ space; thus, with the additional $O(S)$ space, the total space used is $O(n \log^2 n + m + S)$. \square

In addition, we assume that the algorithms **BatchInsert** and **BatchDelete** correctly maintain the desired properties required by each specific problem after processing \mathcal{B}_{ins} and \mathcal{B}_{del} , respectively. Such an assumption ensures the correctness of the solutions produced by our framework. We show in Section 6.9 that this is true for all of our procedures. Additionally, we can get rid of the $O(n \log^2 n)$ term in space at the expense of an extra $O(\log^2 n)$ factor in span by using our space-efficient structures from [216].

Using this framework (with the PLDS guarantees given in Theorem 6.1), we present batch-dynamic algorithms in Section 6.9 for k -clique counting.

6.9 Clique Counting

Using our framework (Algorithm 6.7) and our problem specific methods, we obtain a k -clique counting algorithm (for constant k) that runs in $O(\alpha^{k-2} |\mathcal{B}| \log^2 n)$ work and $O(\log^2 n)$ span w.h.p. using $O(m\alpha^{k-2} + n \log n)$ space.

6.9.1 Algorithm Overview

Due to the complexity of our algorithm, we first provide some intuition behind the core ideas before we give the specific details. First, we make the simple observation that any clique in a directed acyclic graph has a vertex where all edges in the clique that are adjacent to the vertex are directed out from the vertex. For a particular clique C , we call this vertex the *source* of C .

Observation 6.11. *Provided a directed acyclic orientation of a graph $G = (V, E)$, for any clique $C \in G$, there exists a unique vertex $v \in C$ where all edges from v to all other vertices $w \in C$ are directed from v to w .*

Proof. First, it is easy to see that the source is unique. This is because for any two vertices u and v in the clique, the edge $\{u, v\}$ must be directed either in the (u, v) direction or in the (v, u) direction, one of which makes v (resp. u) no longer the source.

Then, a simple proof by contradiction shows that the source exists. Suppose for contradiction that all vertices in C have at least one out-neighbor and one in-neighbor. We start with vertex v . Suppose that v 's out-neighbor is w and v 's in-neighbor is u . By our assumption, w must have at least one out-neighbor, x . $x \neq u$, otherwise, there exists a 3-cycle in the graph. ($x \neq v$ also since we're only considering simple graphs.) By the same argument, x must have at least one out-neighbor, y . $y \notin \{x, u, v, w\}$, otherwise, by the same argument, there would exist a cycle and we only consider simple graphs. Making the same argument for the k -th unique out-neighbor, we require a $(k+1)$ -st unique vertex in order to not create a cycle. This contradicts the fact that C is a k -clique. \square

We begin our description with an explanation of how to find the newly created cliques resulting from edge insertions. Using Theorem 6.11, we make the second observation that for any edge update (u, v) , we can count the number of k -cliques (for constant k) incident to (u, v) and where u is the source vertex of the clique in $O(\alpha^{k-2})$ work and $O(1)$ span, provided an $O(\alpha)$ acyclic low out-degree orientation. This is because u and v must be in the clique and, thus, there are $\binom{c\alpha}{k-2} = O(\alpha^{k-2})$ additional vertices to choose from among u 's out-neighbors (for some constant c hidden in the $O(\alpha)$). This observation also means that we do not have to worry about finding a clique *until after all edges adjacent to its source vertices* are added. The clique will be found by the last of these source edges when it is added. Thus, the main challenge of our algorithm is how to find the cliques resulting from edge updates to other vertices *aside from those adjacent to the source vertex*.

This leads to our third and final observation: a k -clique can be formed from a $(k-1)$ -clique by attaching a source vertex where all edges from the source vertex are directed into the vertices of the $(k-1)$ -clique. The last crucial observation allows us to count k -cliques inductively by counting $(k-1)$ -cliques, which are in turn counted using $(k-2)$ -cliques, and so on. This means that for any k -clique C , by Theorem 6.11, there exists a set of *unique* source vertices responsible for the set of smaller cliques within C . Specifically, for every clique $C_i \subseteq C$ of size $i \in [2, \dots, k]$, there exists a source for this clique. For every edge insertion, we first determine the possible sets of k vertices which can be *completed* by future edge insertions to form k -cliques. Potential cliques are determined using the above observation by assuming for each edge insertion (u, v) , u is the source of the clique. Suppose C is one such set. We assign the responsibility of counting the potential k -clique C to the *largest incomplete clique*, $C_i \subset C$, *without a source*. This can occur when C_i does not yet have enough edges to determine the source (see (1) in Fig. 6.15 where $\{a, b, d, e\}$ does not yet have a source). The base case, the smallest possible largest incomplete clique without a source, is an edge; once this edge is inserted, the source of the edge counts the clique. The concept of the largest incomplete clique without a source is fundamental to our algorithm.

Given a batch of edge insertions, if a set of edges completes the largest incomplete

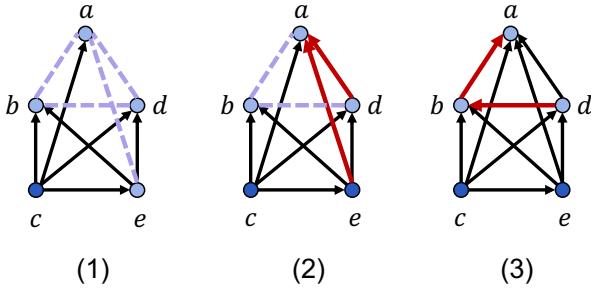


Figure 6.15: Example of incomplete cliques in our counting algorithm for counting $k = 5$ cliques. In (1), c is the source of a potential 5-clique. $\{a, b, c, d, e\}$ represents a potential 5-clique. We do not yet know the source of the 4-clique consisting of $\{a, b, d, e\}$ (the purple edges represent potential edges), and so we store $\{a, b, d, e\}$ in table I_4 . (2) shows a set of edge insertions (indicated by the red edges) which determines a source (e) for the 4-clique. Thus, we insert $\{a, b, d\}$ in table I_3 . Suppose that this is the only clique that would be counted when edges are inserted between all pairs in $\{a, b, d\}$. Thus, we associate with this key, a count of 1 in table I_3 . Finally, (3) shows two edge insertions which completes the triangle; hence, we count the clique using the key $\{a, b, d\}$ and increment the k -clique count using the count associated with it (in this example, the count is 1) in table I_3 .

clique without a source, C_i , of C , then the new source of this clique is responsible for counting C in the clique count. Crucially, C cannot be counted until C_i is completed; furthermore, for any set of k vertices C with a source, but is not a clique, there exists a C_i that can count C . If the batch does not complete the clique but a source has been found for C_i , then, we determine the new largest incomplete clique without a source, C_j (where $j < i$), that will be responsible for counting C . In Fig. 6.15, (2) shows a set of insertions that determines that e is the source of $\{a, b, d, e\}$. Then, the new largest incomplete clique without a source is $\{a, b, d\}$.

This naturally leads to an algorithm for counting k -cliques. We create $k - 2$ parallel hash tables, where for each potential k -clique C , we store the indices of the vertices comprising the largest incomplete cliques, C_i , without a source, for $i \in [2, k - 1]$, in table I_i . The values stored in these hash tables are the numbers of k -cliques C that would be completed if C_i in table I_i is completed. Storing the indices of all vertices allows us to determine the source when the appropriate edges have been inserted. (More details are given in our detailed algorithm below.) Given this set of structures, we increase the k -clique count when a clique from table I_i is completed by a batch of insertions; to increase the k -clique count, we use the value stored for the clique. If there remains any incomplete cliques, we use each table I_j for $j > 2$ to update tables I_i for $i < j$ if any incomplete cliques in j have found sources. We give an example illustration of this part of the algorithm in Fig. 6.15.

Data Structures We maintain the following data structures in our algorithm. We maintain $k - 2$ parallel hash tables, I_i for $i \in [2, k - 1]$. For each I_i , the keys are

Algorithm 6.8 – CliqueCountingBatchFlips($A, \mathcal{B}_{ins}, \mathcal{B}_{del}$)

Require: A set of edge flips A .

Ensure: Updates $\mathcal{B}_{ins}, \mathcal{B}_{del}$.

- 1: **parfor** each flipped edge $(u, v) \in A$ **do** \triangleright The edge is flipped from (u, v) to (v, u) and stored as (u, v) in A .
 - 2: $\mathcal{B}_{del} \leftarrow \mathcal{B}_{del} \cup (u, v)$.
 - 3: $\mathcal{B}_{ins} \leftarrow \mathcal{B}_{ins} \cup (v, u)$.
-

ordered sets of vertices of size i , and the values are counts. The counts represent the number of k -cliques that would form if all edges among the vertices in the keys exist. To prevent over-counting, one edge update incident to the new source of any newly completed clique stored in I_i is responsible for increasing the count by the stored value.

6.9.2 BatchFlips Implementation

Our algorithm uses the framework given in Section 6.8. We first instantiate the algorithm for **BatchFlips**($A, \mathcal{B}_{ins}, \mathcal{B}_{del}$), which creates a set of edge insertions and deletions from the flipped edges in A and appends these edges to \mathcal{B}_{ins} and \mathcal{B}_{del} . The pseudocode is given in Algorithm 6.8. In parallel, for each edge that is flipped from (u, v) to (v, u) (Line 1), we add (u, v) to \mathcal{B}_{del} (Line 2) and add (v, u) to \mathcal{B}_{ins} (Line 3).

6.9.3 BatchInsert Implementation

We now instantiate **BatchInsert** for k -clique counting. The main basis of our **BatchInsert** and **BatchDelete** subroutines is to maintain our parallel hash tables throughout edge insertions and deletions. We first describe **CliqueCountingBatchInsert**, given in Algorithm 6.9. **CliqueCountingBatchDelete** is symmetric and is discussed in Section 6.9.4.

Before we dive into the details of the implementation of Algorithm 6.9, we first provide some intuition for how our algorithm implements our intuitive approach in Section 6.9.1. The key piece of information that our algorithm maintains after all updates is how many sets of k vertices, C , can be a k -clique if a subset of $2 \leq i \leq k-1$ vertices, $C_i \subset C$, has edges between all pairs of vertices in the set. Table i keeps all C_i sets of vertices (as keys). In other words, the entry $I_i[C_i]$ precisely counts the number of unique sets of k distinct vertices, C , where the following two properties hold:

1. $C_i \subset C$.
2. For every $v \in C \setminus C_i$, the directed edge from v to w , (v, w) , exists in G for every $w \in C_i$.

The bulk of Algorithm 6.9 is concerned with updating all of the tables I_i for $i \in [2, \dots, k-1]$ such that the above counts hold for every entry. Using these counts, we can find the number of new k -cliques created by inserting \mathcal{B}_{ins} by checking for each C , whether its largest incomplete clique without a source is completed. We can do this

Algorithm 6.9 – CliqueCountingBatchInsert(\mathcal{B}_{ins})

Require: A batch \mathcal{B}_{ins} of unique and valid insertion updates.

Ensure: An updated k -clique count and updated data structures.

```

1: Let  $count$  be the current count of the number of  $k$ -cliques in the graph.
2: Insert the edges in  $\mathcal{B}_{ins}$  into the graph in the orientation specified by  $\mathcal{B}_{ins}$ . Mark all
   edges in  $\mathcal{B}_{ins}$  in the graph.
3: Let  $R$  be the order of the edges in  $\mathcal{B}_{ins}$ .
4: parfor each edge  $(u, v) \in \mathcal{B}_{ins}$  do                                 $\triangleright$  The edge is oriented from  $u$  to  $v$ .
5:   for  $i \in [1, \dots, k - 2]$  do
6:     parfor each subset  $T$  of  $i$  out-neighbors of  $u$  (excluding  $v$ ) do
7:       Let  $T'$  be the ordered set of  $T \cup \{u, v\}$  sorted by vertex index.
8:       if  $u$  is the source of  $T'$  and  $(u, v)$  is the earliest in  $R$  out of all marked edges
          from  $u$  to  $w \in T'$  then
9:         if all edges between each pair  $x, y \in T'$  exists then
10:           $j \leftarrow 2$ .
11:        else
12:          Find largest incomplete clique without a source,  $C'$ , in  $T'$ .
13:          Let  $j$  be the size of  $C'$ .
14:           $T_{sub} \leftarrow T'$ .
15:          for  $l = i + 1$  to  $j$  do
16:            Find  $s$  the source of  $T_{sub}$ .
17:             $T_{sub} \leftarrow T_{sub} \setminus s$ .
18:            if  $|T_{sub}| = k - 1$  then
19:              if  $l == i + 1$  and  $T'$  is a  $(i + 2)$ -size clique then
20:                 $count \leftarrow count + 1$ .
21:                 $I_l[T_{sub}] \leftarrow I_l[T_{sub}] + 1$ .
22:              else if  $T' \in I_{i+2}$  then
23:                if  $l == i + 1$  and  $T'$  is a  $(i + 2)$ -size clique then
24:                   $count \leftarrow count + I_{i+2}[T']$ .
25:                 $I_l[T_{sub}] \leftarrow I_l[T_{sub}] + I_{i+2}[T']$ .
26: Unmark all marked edges in the graph.

```

efficiently because (1) we do not need to check this individually for every C since our tables I_i already maintain these counts; and (2) if a largest incomplete clique without a source C_i is completed, then it must be incident to an edge update $(u, v) \in \mathcal{B}_{ins}$, where u is the new source of C_i and $v \in C_i$. Knowing (2), we can afford to enumerate sets of out-neighbors of C_i to determine whether C_i is a clique. If C_i is a clique, then we count all C that has it as its largest incomplete clique without a source by adding $I_i[C_i]$ to the cumulative count. The remaining parts of Algorithm 6.9 ensure that we do not over-count newly formed cliques.

We now describe Algorithm 6.9 in detail. For each edge insertion (u, v) where the edge is oriented from u to v (Line 4), we iterate through *all* possible subsets of i out-neighbors of u (*excluding* v , since we know v must be included in the clique) where $i \in [1, \dots, k - 2]$ (Lines 5 and 6). We iterate through $i \in [1, \dots, k - 2]$ because u and v necessarily need to be included in the clique. This is to account for all

possible largest incomplete cliques without a source that are currently in our hash tables. In order to find whether \mathcal{B}_{ins} completes any of these cliques, we must find these cliques by performing this enumeration. Let T be the subset of out-neighbors picked. We consider all cliques of size $i+2$ consisting of the vertices in $T' = T \cup \{u, v\}$. Then, Line 7 provides a canonical order for the vertices in T' so that we can search for T' in I_i . Note that we need to avoid duplicate counting. To avoid duplication, we use the order of the edge insertions in **BatchInsert** (Line 3) and assign the task of updating the clique count to the first insertion in this order, (u, w) , where $w \in T'$. Hence, the if statement in Line 8 checks that all of these above conditions are satisfied. The if statement in Line 9 checks whether the newly inserted edges create a new clique, and if not (Line 11), the algorithm then finds the largest incomplete clique without a source, C' , that contains a subset of the vertices in T' (Line 12). The algorithm then sets a parameter j to be the size of C' (Line 13). If T' is a completed clique, it passes the check on Line 9 and we assign $j = 2$ (Line 11). Now, we consider two possible scenarios.

First, (u, v) along with the other edge insertions in \mathcal{B}_{ins} could complete a largest incomplete clique without a source. In this case, we should increase the k -clique count if u is also the new source of the clique. Furthermore, Lines 19 and 23 check if the clique is completed. If u is a new source, the clique for which it is the source is completed, and $T' \in I_{i+2}$, then we increment the clique count with the value $I_{i+2}[T']$ (Lines 22 and 23). The value stored in $I_{i+2}[T']$ is the number of new cliques that are created if T' is completed. Note that Line 24 is only called if $|T'| < k$ since $l \leq k-1$ and Line 18 handles the case when $|T'| = k$. This is because we do not store size- k sets of vertices in any of the tables; we do not need to store their values because we can enumerate them directly by checking all $(k-1)$ -size subsets of out-neighbors of every u in every edge insertion (u, v) . We denote the ordered set of vertices that gives the key in table I_l by T_{sub} (initially setting $T_{sub} \leftarrow T'$ (Line 14)). If $T' \notin I_{i+2}$ and the size of T_{sub} is $k-1$ (implying the size of T' is k), then we directly increment the clique count by 1 (Lines 18 to 20). As before, in this case, we directly enumerate the new clique for the edge insertion (u, v) without needing to check the tables. This also means that u is the source of the newly created k -clique consisting of the vertices in T' .

After we update the k -clique count, we must then update the $I_i[C_i]$ counts for each $C_i \subset T'$. We need to update these counts because now T' contains a vertex $v \in T' \setminus C_i$ where there exists an edge (v, w) for every $w \in C_i$ (this vertex did not exist previously). Thus, the count for C_i must be incremented by $I_{i+2}[T']$ if $i+2 \leq k-1$, and 1 if $i+2 = k$. This is because, as previously discussed, intuitively, $I_{i+2}[T']$ stores the number of k -cliques that would be created if T' were completed, so we must similarly maintain the number of k -cliques created now that each C_i is completed. We prove this more concretely in Section 6.9.5. When T' is a clique, there exists a C_i for every $2 \leq i < |T'|$ whose entry $I_i[C_i]$ needs to be updated. So, Line 15 loops through each of these possible sizes and the entries are updated by Line 21 or Line 25 depending on whether $|T'| = k$.

The second scenario is that (u, v) and the other edge insertions in \mathcal{B}_{ins} do not complete a clique but create a new source among the vertices in T' . The means that

we need to find a new largest incomplete clique without a source within T' . Again, to avoid duplication, we assign the task to the earliest edge update that is incident to u . Similar to the case when T' is a clique, the algorithm also needs to update tables I_{i+1} to I_j in this case (Line 15). However, we do not update *all* of the tables since T' still has a largest incomplete clique without a source (Line 12). Let $l \in [j, i + 1]$, the table I_j is updated with the number of cliques that would be counted by it if it were the largest incomplete clique without a source. We need to update all of these tables (instead of just table I_j) in order to be able to handle deletions. This is due to the fact that when a smaller clique becomes incomplete due to a batch of deletions, it may cause a larger k -clique to become incomplete. We cannot afford to find all such affected k -cliques; thus, we must store this information in the tables.

To compute the key for table I_l , we need to remove the source of T_{sub} from T_{sub} (Lines 16 and 17). Then, there are two cases we must consider (Line 18 and Line 22). In the first case, when $|T'| = k$, u is a newly created source for a new potential k -clique (Line 18); thus, no entries in the tables have counted T' yet and we increment the count of $I_{k-1}[T_{sub}]$ by 1 (Lines 18 and 21) so that T' will be counted when T_{sub} is completed as the largest incomplete clique without a source. In the second case (Line 22), T' is already an entry in table I_{i+2} ; this means that it already counts the a number of k -cliques that exist if T' is a clique. In this case, we increase the value for $I_l[T_{sub}]$ by $I_{i+2}[T']$ since by definition of the values we store in $I_l[T_{sub}]$, if T_{sub} is a clique, then all the k -cliques that are counted when T' is a clique will now be counted when T_{sub} is completed as the largest incomplete clique without a source (Line 25).

6.9.4 BatchDelete Implementation

Our `CliqueCountingBatchDelete` algorithm is nearly identical to our `CliqueCountingBatchInsert` algorithm; in places where we assign clique counts in the insertion algorithm, we instead remove clique counts in the deletion algorithm. Such changes are expected since deletions of edges remove cliques from the count and also remove assignments of cliques to largest incomplete cliques without a source. The pseudocode is provided in Algorithm 6.10. The few changes to the algorithm are highlighted in blue.

6.9.5 Correctness

To prove the correctness of our algorithm, we first show that Algorithm 6.9 and Algorithm 6.10 accurately store the counts associated with the largest incomplete cliques without sources. For simplicity, we provide separate lemmas for Algorithm 6.9 and Algorithm 6.10, although fundamentally, the proof techniques are the same for both algorithms.

We use the following notation in Lemma 6.3 and Lemma 6.4. Let c_L be the number of sets of k vertices in the graph which do not form a clique, contains a source, and whose largest incomplete cliques without sources is the set of vertices in L *after processing the current, input batch of updates*. We show that after running Algorithm 6.9 or Algorithm 6.10, $I_{|L|}[L] = c_L$. In fact, we show an even stronger lemma; suppose that J is a set of vertices in the graph where $2 \leq |J| \leq k - 1$. If we remove

Algorithm 6.10 – CliqueCountingBatchDelete(\mathcal{B}_{del})

Require: A batch \mathcal{B}_{del} of unique and valid deletion updates.

Ensure: An updated k -clique count and updated data structures.

```

1: Let count be the current count of the number of  $k$ -cliques in the graph.
2: Insert all edges in  $\mathcal{B}_{del}$  into the graph in the orientation specified by  $\mathcal{B}_{del}$ . Mark all edges
   in  $\mathcal{B}_{del}$  in the graph.
3: Let  $R$  be the order of the edges in  $\mathcal{B}_{del}$ .
4: parfor each edge  $(u, v) \in \mathcal{B}_{del}$  do                                $\triangleright$  The edge is oriented from  $u$  to  $v$ .
5:   for  $i \in [k - 2, \dots, 0]$  do
6:     parfor each subset  $T$  of  $i$  out-neighbors of  $u$  do
7:       Let  $T'$  be the ordered set of  $T \cup \{u, v\}$  sorted by vertex index.
8:       if  $u$  is the source of  $T'$  and  $(u, v)$  is the earliest in  $R$  out of all marked edges
         from  $u$  to  $w \in T'$  then
9:         if all edges between each pair  $x, y \in T'$  exists then
10:           $j \leftarrow 2$ .
11:        else
12:          Find largest incomplete clique without a source,  $C'$ , in  $T'$ .
13:          Let  $j$  be the size of  $C'$ .
14:           $T_{sub} \leftarrow T'$ .
15:          for  $l = i + 1$  to  $j$  do
16:            Find  $s$  the source of  $T_{sub}$ .
17:             $T_{sub} \leftarrow T_{sub} \setminus s$ .
18:            if  $|T_{sub}| = k - 1$  then
19:              if  $l == i + 1$  and  $T'$  is a clique then
20:                count  $\leftarrow$  count  $-1$ .
21:                 $I_l[T_{sub}] \leftarrow I_l[T_{sub}] -1$ .
22:              else if  $T' \in I_{i+2}$  then
23:                if  $l == i + 1$  and  $T'$  is a clique then
24:                  count  $\leftarrow$  count  $-I_{i+2}[T']$ .
25:                   $I_l[T_{sub}] \leftarrow I_l[T_{sub}] -I_{i+2}[T']$ .
26: Delete all marked edges in the graph.

```

all edges between pairs of vertices in J , let c_J be the number of sets of k vertices that do not form a clique, contains a source, and whose largest incomplete cliques without sources is the set of vertices in J . We show that $I_{|J|}[J] = c_J$. This stronger form of the lemma is not necessary if we only consider insertion updates; however, under deletion updates, we require this stronger lemma in order to prove correctness. It is sufficient to assume that all data structures are maintained corrected at the beginning of Algorithm 6.9 and Algorithm 6.10 and they remain correct at the end of the algorithms (since by induction, this would prove that the data structures are always correctly maintained).

Lemma 6.3. *After running Algorithm 6.9 on \mathcal{B}_{ins} , $I_{|J|}[J] = c_J$ for every c_J where $2 \leq |J| \leq k - 1$.*

Proof. We prove this lemma via induction on the table index i , starting with $i = k - 1$.

We first prove our base case for $i = k - 1$. In Algorithm 6.9, the value stored in $I_{k-1}[J]$ is only ever incremented in Line 21 since this is the only time when table I_{k-1} can be modified (the condition in Line 22 is never satisfied for any entries in table I_{k-1}). By the condition given in Line 8, $I_{k-1}[J]$ is only incremented when T' has a source s where $J = T' \setminus \{s\}$. Furthermore, the condition that $I_{k-1}[J]$ is incremented by the earliest edge update incident to s ensures that it is incremented at most once by each T' . The number of sets c_J of vertices T' where $J \in I_{k-1}$ is the largest incomplete clique without a source (if all edges in J are removed) is precisely the number of vertices s in the graph with edges directed into all vertices in J . Our argument above shows that $I_{k-1}[J]$ is incremented exactly once for each such vertex s ; furthermore, it is incremented only if s is adjacent to an edge update $(s, x) \in \mathcal{B}_{ins}$ and $x \in J$. This proves our base case.

We assume for our induction step that $I_{|J|}[J] = c_J$ for all tables I_j for $j \in [k - 1, \dots, k - l]$ and prove the lemma holds for table I_{k-l-1} . The value $I_{k-l-1}[J]$ is increased in Line 25. Every T' with k vertices increases the value of $I_{k-l-1}[J]$ by 1 if its largest incomplete clique without a source has size $\leq |J|$. This is easy to see since if all edges from J are removed, then J would be the largest incomplete clique without a source for T' . By our induction hypothesis, the counts of these T' 's are correctly stored in tables I_j for $j \in [k - 1, \dots, k - l]$. Line 8 ensures that only one edge update is responsible for incrementing $I_{k-l-1}[J]$ for each T' ; furthermore, it ensures that $I_{k-l-1}[J]$ is incremented with the value stored in $I_{|C|}[C]$ where $C \subseteq T'$ is the previous largest incomplete clique without a source before the current batch \mathcal{B}_{ins} of insertions. $J \subset C$ by definition, and J is guaranteed to be the largest incomplete clique without a source (after the edge insertions in \mathcal{B}_{ins}) by our argument above. In addition, each T' is counted in at most one $C \subset T'$ in each table I_j where $j \in [k - 1, \dots, k - l]$. This is true by our induction hypothesis since each T' has one unique C where $|C| = j$ which is the largest incomplete clique without a source (if the edges in C are removed).

The last step we need to prove in order to prove our induction hypothesis is that $I_{k-l-1}[J]$ is incremented by 1 for T' with exactly one $I_{|C|}[C]$ where $C \subset T'$. We prove this via contradiction. Suppose there are two subsets $C' \subset C \subset T'$ which are used to increment $I_{k-l-1}[J]$. Let $s' \in C'$ be the source of C' and $s \in C$ be the source of C . This means that in order to satisfy Line 8, s and s' must be incident to some update $(s, x) \in \mathcal{B}_{del}$ (resp. $(s', x') \in \mathcal{B}_{del}$) where $x, x' \in T'$. This means that C was the previous largest incomplete clique without a source for T' and so C' would not contain a count for T' by our induction hypothesis. Since, we process the tables in Line 5 starting with table I_2 in increasing order of table index, $I_{k-l-1}[J]$ cannot be incremented with the count for T' from C' , a contradiction. Thus, $I_{|C|}[C]$ correctly counts all T' and hence, $I_{k-l-1}[J]$ is incremented exactly once for each T' and we have proven our inductive step. \square

The proof of the property for Algorithm 6.10 is almost identical to Lemma 6.3 except to account for the few changes shown in blue in Algorithm 6.10. For simplicity, we present only the parts of the proof that requires more effort than replacing decrement for all mentions of increment in the proof of Lemma 6.3.

Lemma 6.4. *After running Algorithm 6.10 on \mathcal{B}_{del} , $I_{|J|}[J] = c_J$ for every c_J where $2 \leq |J| \leq k - 1$.*

Proof. We prove this lemma via induction on the table index i , starting with $i = k - 1$. The proof of our base case for $i = k - 1$ directly follows from the proof of the base case in Lemma 6.3 when we replace instances of increment with decrement.

We assume for our induction step that $I_{|J|}[J] = c_J$ for all tables I_j for $j \in [k - 1, \dots, k - l]$ and prove the lemma holds for table I_{k-l-1} . The value $I_{k-l-1}[J]$ is increased in Line 25. The proof of the inductive step follows from the proof of the inductive step in the proof of Lemma 6.3 by replacing instances of increase by decrease, except for the last step which we prove below.

The last step we need to prove in order to prove our induction hypothesis is that $I_{k-l-1}[J]$ is decremented by 1 for T' with exactly one $I_{|C|}[C]$ where $C \subset T'$. We prove this via contradiction. The initial setup is the same as the setup in the proof of Lemma 6.3. Suppose there are two subsets $C' \subset C \subset T'$ which are used to decrement $I_{k-l-1}[J]$. Let $s' \in C'$ be the source of C' and $s \in C$ be the source of C . This means that in order to satisfy Line 8, s and s' must be incident to some update $(s, x) \in \mathcal{B}_{\text{ins}}$ (resp. $(s', x') \in \mathcal{B}_{\text{ins}}$) where $x, x' \in T'$.

This means that C is now the largest incomplete clique without a source for T' after processing the deletions in \mathcal{B}_{del} . Thus, because we process the tables in *decreasing order* by table index, starting with table I_2 (Line 5), C satisfies the conditions in Line 8 and by Line 25, C would have deleted the count of T' from $I_{|C'|}[C']$. Thus, C' would not contain a count for T' and $I_{k-l-1}[J]$ cannot be incremented with the count for T' from C' , a contradiction. $I_{|C|}[C]$ correctly counts all T' by our induction hypothesis and hence, $I_{k-l-1}[J]$ is decremented exactly once for each T' and we have proven our inductive step. \square

We are now ready to prove that our algorithms correctly return the k -clique count provided batches of updates.

Theorem 6.12. *Our algorithms, Algorithm 6.9 and Algorithm 6.10, correctly returns the number of k -cliques in a given input graph, $G = (V, E)$, provided batches of updates \mathcal{B}_{ins} and \mathcal{B}_{del} , respectively.*

Proof. Provided Lemma 6.3 and Lemma 6.4, we only need to show the following: given \mathcal{B}_{ins} , each k -clique C completed by \mathcal{B}_{ins} (i.e. contains at least one edge in \mathcal{B}_{ins}), is counted exactly once, and by exactly one update edge incident to the source of its largest incomplete clique without a source (*prior* to the insertions); given \mathcal{B}_{ins} , each k -clique C destroyed by \mathcal{B}_{del} (i.e. contains at least one edge in \mathcal{B}_{del}), is subtracted exactly once, and by exactly one update edge incident to the source of its largest incomplete clique without a source (*after* the deletions).

We first prove the above is true for insertions. The if statement in Line 8 ensures at most one update edge for a set of vertices $C \subset T'$, where T' is a newly formed clique, increments the clique count. Now, we prove that at most one subset of vertices increments the clique count for T' . Suppose for contradiction two sets of vertices $C' \subset C \subset T'$ increments the total clique count by 1 for T' . Then, in order to

pass the if statement in Line 8, the sources of both C' and C must be adjacent to updates in \mathcal{B}_{ins} that point to vertices in C' and C . Since $C' \subset C$, C was the previous largest incomplete clique without a source for T' . By Lemma 6.3, C' does not contain the count for T' and thus, only C increments the total clique count by 1 for T' , a contradiction.

To prove that at least one subset of vertices increments the clique count for T' , suppose that C was the previous largest incomplete clique without a source for T' but C does not increment the clique count. Since T' is a new clique, it must be incident to at least one edge update in \mathcal{B}_{ins} . Since C does not increment the clique count, it must not have found a source (and cannot satisfy Line 9). (It must satisfy Line 19 or Line 23 since T' is a clique and we iterate through all possible i). Since C does not have a source, by Theorem 6.11, it must be missing at least one edge. Then, T' is not a clique, a contradiction.

The proof follows symmetrically for Lemma 6.4 except that instead of the previous largest incomplete clique without a source, we care about the largest incomplete clique without a source *after* processing \mathcal{B}_{del} . Suppose for contradiction two sets of vertices $C' \subset C \subset T'$ decrement the clique count. Then, their sources must both be incident to edge updates. Since, $C' \subset C$, C is processed first by Line 5 using Lines 15, 21 and 25. This means that the count of T' would have been subtracted from $I_{|C'|}[C']$ and it cannot decrement the clique count by 1 for T' , a contradiction. Suppose instead, that C is the largest incomplete clique without a source for T' after processing \mathcal{B}_{del} and it does not decrement the clique count. Either one of two scenarios can occur: either $I_{|C|}[C]$ no longer has the count for T' or the source s of C is not incident to any updates. No C'' where $C \subset C''$ can decrement $I_{|C|}[C]$ for T' since by our assumption, the source of C'' is not incident to any updates directed into vertices in C'' . Thus the first scenario cannot occur and we consider the second scenario where the source s must not be incident to an edge update directed into the vertices in C . Then, s still has all its directed edges to the vertices in C and so is the source of C . This means that C has a source and cannot be the largest incomplete clique *without a source*, a contradiction. \square

Together with the proof of correctness of our framework, Section 6.8, our algorithm correctly provides the k -clique count provided a batch of updates, \mathcal{B} .

6.9.6 Work and Span Analysis

We note for the following result that α is defined as $\max(\alpha_b, \alpha_a)$ where α_b is the arboricity before the current batch of updates is processed and α_a is the arboricity after the current batch of updates is processed.

Theorem 6.13. *We obtain a batch-dynamic k -clique counting algorithm that takes $O(\alpha|\mathcal{B}| \log^2 n)$ amortized work and $O(\log^2 n \log \log n)$ span w.h.p. using $O(m\alpha^{k-2} + n \log^2 n)$ space.*

Proof. We first show the work, span, and space of our algorithms, Algorithms 6.8 to 6.10, and then use Theorem 6.10 to show the bounds for our algorithm. Note

that the increments and decrements to the global k -clique count can be performed in $O(\log n)$ span in parallel by writing each update to an array, and then using parallel reductions at the end to update the global k -clique count. We use the same strategy for updating the hash table counts. Furthermore, our parallel hash table primitives allow us to concurrently modify elements in parallel in $O(\log n)$ span w.h.p. In Algorithm 6.8, the batches \mathcal{B}_{ins} and \mathcal{B}_{del} can be obtained in $O(|\mathcal{B}| \log^2 n)$ work and $O(\log n)$ span. Note that by construction, $|\mathcal{B}_{ins}|, |\mathcal{B}_{del}| = O(|\mathcal{B}| + |A|) = O(|\mathcal{B}| \log^2 n)$. All edges can be checked in parallel (Line 1) and inserted into parallel dynamic arrays; we can also use a simple parallel filter. For the remainder of this proof, we discuss the work and span complexity for a batch \mathcal{B}_{ins} of edge insertions in Algorithm 6.9; the deletion algorithm (Algorithm 6.10) has the same work, span, and space complexity.

All edge insertions are processed in parallel using a parallel loop (Line 4). We then run a sequential for loop of span $O(k)$ (Line 5). Let i be the current index of the sequential for loop. In order to process edge insertions (u, v) , where u is a source, we iterate in parallel over all sets T of $i + 1$ out-neighbors of u including v . Since there are at most $O(\alpha)$ out-neighbors of u , and since v is necessarily included, we have $\binom{\alpha}{i} = O(\alpha^i)$ possible sets T (assuming constant k). For constant k , we perform a constant number of parallel hash table operations and checks for the existence of edges per set T (Lines 9, 10 and 12 to 14). We make $O(k)$ iterations of the for loop in Line 15; updating the hash tables (Lines 21 and 25) require $O(k)$ total work per edge update. Checking for the source of T_{sub} over all T_{sub} requires $O(k^2)$ work per edge update (Line 16). Thus, per edge insertion (u, v) , for constant k , we incur $O(\alpha^i)$ work and $O(\log n)$ span w.h.p. ver all $i \in [0, \dots, k-2]$, this results in $\sum_{i=0}^{k-2} O(\alpha^i) = O(\alpha^{k-2})$ total work over all i , w.h.p. he span is $O(\log^* n)$ w.h.p. ue to the hash table operations and updating the table values by writing to an array and using a parallel reduction for each entry results in $O(\log n)$ span.

Lastly, we update the global k -clique count by writing each update to an array and using a parallel reduction at the end, which maintains the same work and span bound.

Processing the entire batch of insertions in parallel, we have $O(\alpha^{k-2} |\mathcal{B}| \log^2 n)$ amortized work and $O(\log n)$ span w.h.p. hus, in total, our k -clique counting algorithm takes $O(\alpha^{k-2} |\mathcal{B}| \log^2 n)$ amortized work and $O(\log^2 n)$ span w.h.p. y Theorem 6.10.

Our space usage is proportional to the space required to store the contents of the parallel hash tables I_i for $i \in [2, \dots, k - 1]$. By construction, for each edge insertion (u, v) , we create at most $\sum_{j=0}^{k-2} O(\alpha^{j-2}) = O(\alpha^{k-2})$ hash table entries across all I_i . This follows directly from our work analysis. Thus, in total, we use space proportional to $O(m\alpha^{k-2})$. \square

6.9.7 Comparison with Previous Work

The best-known batch-dynamic algorithm for k -clique counting for graphs with low arboricity is given by Dhulipala et al. [104]. They give a $O(|\mathcal{B}| m \alpha^{k-4})$ expected work and $O(\log^{k-2} n)$ span w.h.p. lgorithm using $O(m + |\mathcal{B}|)$ space. Our algorithm improves upon the work of this previous result when $m = \omega(\alpha^2 \log^2 n)$. Note that $\alpha \leq \sqrt{m}$ [72]. Furthermore, in real-world graphs, often $\alpha \ll \sqrt{m}$, since real-world graphs tend

to have small arboricity.

Our algorithm achieves better span for all $k > 4$. For 4-cliques, our span matches the previous algorithm while for larger cliques, we achieve a better span. Finally, we obtain these gains with an increase in space of $O(\alpha^{k-2} + \log^2 n)$ multiplicative factor, but for bounded arboricity graphs, this increase in space is small.

6.10 Discussion

We designed the first shared-memory, multi-core parallel batch-dynamic level data structure that returns a $(2 + \varepsilon)$ -approximation for the k -core decomposition problem, drawing inspiration from the sequential level data structures of Bhattacharya et al. [42] and Henzinger et al. [164] which were used for dynamic densest subgraphs and dynamic low out-degree orientation, respectively. Our algorithm achieves $O(\log^2 m)$ amortized work and has $O(\log^2 m \log \log m)$ span. We also presented a proof of the $(2 + \varepsilon)$ -factor of approximation for our data structure, a new proof that is also applicable (with a simple change) to the original sequential level data structures of Bhattacharya et al. [42] and Henzinger et al. [164].

In addition to our batch-dynamic k -core decomposition results, we also gave a batch-dynamic algorithm for maintaining an $O(\alpha)$ out-degree orientation, where α is the *current* arboricity of the graph. We demonstrated the usefulness of our low out-degree orientation algorithm by presenting a new framework to formally study batch-dynamic algorithms in bounded-arboricity graphs. Our framework obtained new provably-efficient parallel batch-dynamic algorithms for maximal matching, clique counting, and vertex coloring.

We performed extensive experimentation of our parallel batch-dynamic k -core decomposition algorithm on large real-world data sets that show that our PLDS is not only theoretically but also practically efficient. Our experiments tested error vs. runtime, batch size vs. runtime, number of hyper-threads vs. runtime, and space vs. error. We also tested the sensitivity of our implementation to the various tunable parameters of our algorithm. Finally, we tested our algorithm against six other benchmarks on 11 real-world graphs, including graphs orders of magnitude larger than previously studied by other dynamic algorithms. We observed an improvement in performance against all other benchmarks in our experiments. Specifically, we achieved speedups of up to $114.52\times$ against the best parallel implementation, up to $544.22\times$ against the best approximate sequential algorithm, and up to $723.72\times$ against the best exact sequential algorithm. Such speed-ups exceed the expected speed-up gained from parallelism alone (since we only use 60 hyper-threads) and are also due to the theoretical improvements of our algorithm as well as our choice of heuristic optimizations.

An interesting open problem is to design a parallel batch-dynamic algorithm that is space-efficient (uses linear space), without incurring additional costs in span.

Part II

Subgraph Decomposition

Introduction

The discovery of dense substructures, particularly through the use of higher order structures, is a fundamental topic in graph mining. In particular, understanding the distributions of and relationships between dense substructures has important applications in graph visualization tasks [15, 353], gene correlation and DNA motif detection in biological networks [345, 136], motif detection in financial networks [113], and community detection in social networks and web graphs [211, 126]. Constructing hierarchies of dense substructures can form the basis of clustering pipelines, either as a preprocessing step or as a clusterer in itself [205, 146, 32, 321].

However, these dense substructure discovery algorithms often rely first on finding and listing certain subgraphs as a subroutine, and then maintaining these subgraphs through a decomposition algorithm in order to construct these hierarchies. This general process is computationally intensive, with many theoretical and practical barriers, due to P-completeness results and memory limitations, which we explore throughout this chapter. For instance, the previous state-of-the-art serial implementations for butterfly peeling [282] takes over 4 hours on a graph with 5.7 million edges (discogs_style [199]) on a machine with 48 cores and 384 GiB of main memory, and the previous state-of-the-art serial implementations for finding the k -clique densest subgraph [86, 127] take over 5 hours for $k = 8$ on a graph with 200 million edges (comorkut [207]) on a machine with 30 cores and 240 GiB of main memory. Moreover, for the same graph and on the same machine, the previous state-of-the-art parallel implementation for computing the $(3, 4)$ -nucleus decomposition [284, 283] takes over an hour.

We use efficient data structures from GBBS [106] as well as our efficient subgraph counting subroutines in order to provide theoretically efficient and practical parallel decomposition implementations. For bipartite graphs, in Chapter 7, we present new algorithms for the bi-core decomposition, and in Chapter 8, we present new algorithms for butterfly peeling. For general graphs, we present new algorithms for k -clique peeling in Chapter 9, and we generalize even further to nucleus decomposition in Chapters 10 and 11. In order to achieve fast implementations in practice, we present new efficient data structures, notably a new multi-level hash table structure to store information on cliques space-efficiently and a technique for traversing this structure cache-efficiently for the nucleus decomposition problem in Chapter 10. Moreover, we develop various approximation algorithms that take polylogarithmic span and are more scalable alternatives to the exact algorithms, for butterfly peeling in Chapter 8, k -clique peeling in Chapter 9, and nucleus decomposition in Chapter 11.

Our implementations are able to perform butterfly peeling on a graph with 5.7 million edges (discogs_style [199]) in under 0.5 seconds on the previously mentioned machine with 48 core and 384 GiB of main memory. Also, we are able to compute the k -clique densest subgraph on a graph with 200 million edges (com-orkut [207]) in 2.5 hours, and perform (3, 4)-nucleus decomposition on the same graph in under 15 minutes, on the same machine as previously mentioned with 30 cores and 240 GiB of main memory.

The results in this part of the thesis have appeared in the following publications.

- Yihao Huang, Claire Wang, Jessica Shi, and Julian Shun. “Efficient Algorithms for Parallel Bi-core Decomposition”. In: *Proceedings of the SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pp. 17-32, 2023. (Chapter 7)
- Jessica Shi and Julian Shun. “Parallel Algorithms for Butterfly Computations”. In: *Massive Graph Analytics*, pp. 287-330, 2022. (Chapter 8)
Jessica Shi and Julian Shun. “Parallel Algorithms for Butterfly Computations”. In: *Proceedings of the SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pp. 16-30, 2020. (Earlier version)
- Jessica Shi, Laxman Dhulipala, and Julian Shun. “Parallel Clique Counting and Peeling Algorithms”. In: *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*, pp. 135-146, 2021. (Chapter 9)
- Jessica Shi, Laxman Dhulipala, and Julian Shun. “Theoretically and Practically Efficient Parallel Nucleus Decomposition”. In: *Proceedings of the VLDB Endowment*, 15(3), pp. 583-596, 2022. (Chapter 10)
- Jessica Shi, Laxman Dhulipala, and Julian Shun. “Hierarchical and Approximate Parallel Nucleus Decomposition”. In submission. (Chapter 11)

Chapter 7

Bi-core Decomposition

7.1 Introduction

Classic algorithms for dense subgraph discovery, including k -core decomposition [213, 229], k -truss [79], and nucleus decomposition [284], are designed towards general graphs, and do not take into account the specific structure of bipartite graphs. A bipartite graph G consists of two bipartitions of vertices, U and V , where every edge connects a vertex in U to a vertex in V . These graphs model the affiliation between two distinct types of entities, such as in authorship networks [181], group membership networks [286], user-product networks [326], and protein-protein interactions [134]. Directly applying traditional dense substructure analysis techniques designed for general graphs to bipartite graphs does not allow for the bipartitions to be distinguished from one another, which can be important especially if they exhibit different structures. Another approach to discovering dense substructures in bipartite graphs uses graph projection to represent each bipartition as its own graph, and then applies traditional techniques to analyze each of the two resulting graphs; however, such methods still fail to capture important connectivity information and can cause an explosion in the number of edges, making them practically inefficient [282]. Thus, bipartite analogues for classic dense subgraph discovery algorithms are crucial for efficient and accurate dense substructure analysis on bipartite graphs.

Indeed, developing algorithms to apply specifically to bipartite graphs has become a recent popular direction of research [330, 6, 355, 282]. The focus of this work is on the bipartite equivalent of a k -core, known as a bi-core, which was introduced by Ahmed *et al.* [6]. Formally, an (α, β) -core (or a bi-core) is the maximal subgraph where the induced degree of each vertex in the first partition is at least α , and the induced degree of each vertex in the second partition is at least β . The bi-core decomposition has applications in a variety of fields, including spam detection on social networks [41], community search on bipartite networks [331], movie viewership analysis [6], and group recommendation [107]. For example, Beutel *et al.* [41] leveraged the bi-core decomposition to detect spammers on user-post social networks, where spammers and fake accounts often form dense subgraphs by interacting with each others' posts, and Wang *et al.* [331] used the bi-core decomposition as a subroutine

for optimizing community search on bipartite networks, by reducing the search space for dense communities.

Parallel Bi-core Decomposition. Liu *et al.* [215] propose the current state-of-the-art sequential bi-core decomposition algorithm, which computes the (α, β) -core via multiple graph peeling processes. Their sequential algorithm takes $O(\delta m)$ time and $O(m)$ space, where m is the number of edges and δ denotes the degeneracy of the graph. Liu *et al.* also introduce a parallel algorithm; however, their parallel algorithm only parallelizes across different peeling processes and does not parallelize the peeling process itself. As a result, it has long sequential dependencies, which limits its parallel scalability. As the sizes of graphs increase, a bi-core decomposition algorithm with high parallelism and scalability becomes crucial.

We develop an efficient shared-memory parallel bi-core decomposition algorithm that parallelizes the peeling process. In each round of peeling, it removes all vertices with the lowest induced degree from the graph in parallel until the graph is empty. We use the classic *work-span* model to analyze the theoretical complexity of our parallel algorithm, where the *work* is the total number of operations, and the *span* (or the *depth*) is the length of the longest chain of sequential dependencies. We prove that for a graph with m edges our algorithm achieves $O(\delta m)$ work, which matches the best sequential time complexity. Our algorithm achieves $O(\rho \log n)$ span w.h.p., where n is the number of vertices, and ρ denotes the bi-core peeling complexity, which we define as the maximum number of rounds of peeling required to remove all vertices from the graph. Our algorithm uses $O(m)$ space.

Note that ρ is upper bounded by n , so our span is $O(\rho \log n) = O(n \log n)$ w.h.p. Also, the parallel algorithm introduced by Liu *et al.* has a span of $O(m)$, so our parallel algorithm improves upon Liu *et al.*'s algorithm when the number of edges m is $\Omega(\rho \log n)$. Moreover, on real world graphs, we find that $\rho \log n$ is generally 2–3 orders of magnitude smaller than m . We also prove the problem of bi-core decomposition to be P-complete by showing a reduction from the k -core problem to the bi-core decomposition problem. It is as therefore unlikely that there exists a parallel bi-core decomposition algorithm with polylogarithmic span under standard assumptions.

In addition, we develop a parallel index structure, which is an extension of Liu *et al.*'s sequential index structure. The index structure allows for efficient queries of all vertices $x \in (\alpha, \beta)$ -core, for a given α and β , in work linear to the size of the core. Our parallel index structure is able to achieve this with $O(1)$ span. We also introduce an algorithm to construct our index structure in parallel given the bi-core numbers of each vertex in $O(m)$ work and $O(\log n)$ span w.h.p.

Finally, we introduce a practical heuristic that optimizes bi-core peeling by pruning the peeling space of the algorithm. We implement our algorithms and present a comprehensive evaluation on real-world graphs with up to hundreds of millions of edges. We compare our algorithms against Liu *et al.*'s parallel and sequential algorithms, which we use as our baselines. Our parallel bi-core decomposition algorithm achieves up to a 51.4x speedup (average 27.9x) over Liu *et al.*'s sequential algorithm on a machine with 30 cores and two-way hyperthreading. Furthermore, it achieves

up to a 4.9x speedup (average 2.3x) over their parallel algorithm. Our parallel index construction algorithm attains up to a 27.7x speedup (average 18.4x) over Liu *et al.*'s sequential index construction algorithm, and our parallel index query achieves up to a 116.3x speedup (average 43.8x) over Liu *et al.*'s sequential index query. Overall, we show that our implementations demonstrate good scalability over different numbers of threads and over graphs of different sizes.

In summary, the contributions of our work are as follows.

1. We introduce the first theoretically-efficient shared-memory parallel bi-core decomposition algorithm with nontrivial parallelism. We provide an accompanying parallel index construction and query algorithm.
 2. We prove that the problem of bi-core decomposition is P-complete.
 3. We introduce practical optimizations and provide fast implementations of our parallel algorithms that outperform the existing state-of-the-art algorithms.
- Our code is publicly available at <https://github.com/clairebookworm/gbbs>.

The remainder of the chapter is organized as follows. In Section 7.2, we present related work. In Section 7.3, we introduce notation and definitions. In Section 7.4, we introduce our shared-memory parallel bi-core decomposition algorithm and discuss the corresponding practical optimizations. In Section 7.5, we prove the P-completeness of the bi-core decomposition problem, and in Section 7.6, we introduce our parallel index construction and query algorithm. In Section 7.7, we present additional practical optimizations and our experimental results.

7.2 Related Work

k -core Decomposition. The bi-core decomposition problem is an extension of the well-studied k -core decomposition problem; for each vertex v , the k -core decomposition problem asks for the largest integer k such that v is contained within a subgraph where all vertices have induced degree at least k . The first efficient sequential algorithm for k -core decomposition was given by Matula and Beck [229], and there has been much work on parallelizations in both the distributed-memory and shared-memory settings [88, 185, 238, 316, 95].

Other Dense Subgraph Decompositions. k -clique decomposition, k -truss, and (r, s) -nucleus decomposition are all extensions of k -core decomposition that use higher order substructures to discover dense substructures in a graph. k -clique decomposition [317, 296] involves computing the k -clique core number of each vertex v , or the largest c such that there exists an induced subgraph containing v where all vertices are incident upon at least c induced k -cliques. k -truss is a classic extension [79, 353, 350, 324, 186, 283, 13] that asks for the largest k for each edge e such that there exists an induced subgraph containing e where all edges are contained within at least k triangles. Notably, the k -core and k -truss decomposition problems are part of the MIT GraphChallenge [276], demonstrating their practical importance and popularity. The (r, s) -nucleus decomposition [284, 283, 297] further generalizes the k -clique and k -truss decompositions, by asking for the largest c for each r -clique such that there exists an induced subgraph containing the r -clique in which all r -cliques

are contained within at least c induced s -cliques. Notably, k -core decomposition is $(1, 2)$ -nucleus decomposition, k -clique decomposition is $(1, k)$ -nucleus decomposition, and k -truss is $(2, 3)$ -nucleus decomposition.

Generalization of Decomposition Algorithms to Bipartite Graphs. Another direction of work has focused on generalizing these decomposition algorithms to bipartite graphs by focusing on other higher-order structures in bipartite graphs. Zou [355] and Sarıyüce and Pinar [282] defined k -tip and k -wing decomposition on bipartite graphs. k -tip decomposition asks for the largest k for each vertex v such that there exists an induced subgraph in which every vertex is incident to at least k induced $(2, 2)$ -bicliques. Similarly, k -wing decomposition asks for the largest k for each edge e such that there exists an induced subgraph in which every edge is incident to at least k induced $(2, 2)$ -bicliques. Multiple sequential [277, 355, 282, 324, 330] and parallel [298, 202] algorithms have been developed for k -tip and k -wing decomposition.

Ahmed *et al.* proposed the (α, β) -core decomposition problem, or the bi-core decomposition problem and gave the first sequential bi-core algorithm [6]. Ding *et al.* applied bi-core to recommender systems and provided a sequential bi-core algorithm based on the classic k -core peeling algorithm [107]. More recently, Liu *et al.* developed an efficient computation sharing sequential bi-core peeling algorithm and a memory-efficient indexing structure to store the bi-cores for efficient membership queries from vertices [215]. Wang *et al.* extended the problem to weighted bipartite graphs to find the bi-core component with the highest minimum edge weight containing a given query vertex [331].

7.3 Preliminaries

We take every graph to be simple, undirected, and bipartite. A *bipartite graph* is a graph G consisting of two mutually exclusive sets of vertices U and V , such that every edge connects a vertex in U with a vertex in V . In other words, every edge is of the form (u, v) where $u \in U$ and $v \in V$. In the context of the peeling process, we often discuss a subgraph of G consisting of all unpeeled vertices. In that case, we use $N_{\text{in}}(x)$ to refer to the induced neighbors of x and $\deg_{\text{in}}(x)$ to refer to the induced degree of x in the subgraph. We let dmax_v be the maximum degree of all vertices in V , and dmax_u is symmetrically defined for U . We define a bi-core as follows:

Definition 7.1. *A bi-core, or an (α, β) -core, is the maximal induced subgraph $G' = (U', V')$ of G such that for every $u \in U'$, the induced degree $\deg_{\text{in}}(u) \geq \alpha$, and for every $v \in V'$, the induced degree $\deg_{\text{in}}(v) \geq \beta$.*

We define $\max_\alpha(\beta)$ to be the maximum α value, given a value β , such that the (α, β) -core is nonempty. Symmetrically, $\max_\beta(\alpha)$ is defined to be the maximum β value, given a value α , such that the (α, β) -core is nonempty. The degeneracy of the graph, δ , can be equivalently defined as the maximum δ such that the (δ, δ) -core is nonempty. Note that δ is upper bounded by $O(\sqrt{m})$ [213].

See Table 7.1 for a summary of these notations.

We note two additional facts:

| Notation | Definition |
|--------------------------|---|
| G | An undirected, simple, bipartite graph, |
| U | One bipartition of the vertices in G , |
| V | Another bipartition of the vertices in G , |
| x | A vertex in $U \cup V$, |
| u | A vertex in U , |
| v | A vertex in V , |
| $\deg(x)$ | Degree of vertex x , |
| $\deg_{\text{in}}(x)$ | Induced degree of vertex x considering only unpeeled vertices, |
| $N(x)$ | A list of vertex x 's neighbors, |
| $N_{\text{in}}(x)$ | A list of vertex x 's induced neighbors considering only unpeeled vertices, |
| d_{\max_v} | The maximum vertex degree in V , |
| d_{\max_u} | The maximum vertex degree in U , |
| $\alpha_{\max \beta}(v)$ | The maximum α value for a given $v \in V$ and β such that $v \in (\alpha, \beta)$ -core, |
| $\beta_{\max \alpha}(u)$ | The maximum β value for a given $u \in U$ and α such that $u \in (\alpha, \beta)$ -core, |
| $\max_{\alpha}(\beta)$ | The maximum α value such that (α, β) -core is nonempty for fixed β , |
| $\max_{\beta}(\alpha)$ | The maximum β value such that (α, β) -core is nonempty for fixed α , and |
| δ | The maximum δ value such that (δ, δ) -core is nonempty, or the degeneracy. |

Table 7.1: Summary of graph notation.

1. if $x \in (\alpha_1, \beta_1)$ -core, then $x \in (\alpha_2, \beta_2)$ -core if $\alpha_2 \leq \alpha_1$ and $\beta_2 \leq \beta_1$ [215].
2. Every nonempty (α, β) -core must have $\alpha \leq \delta$ and/or $\beta \leq \delta$ [215].

Problem Statement. Now, we formally define the bi-core decomposition problem.

Definition 7.2. For a vertex $v \in V$ and fixed β , we define $\alpha_{\max \beta}(v)$ to be the maximum α such that $v \in (\alpha, \beta)$ -core. Symmetrically, for a vertex $u \in U$ and fixed α , we define $\beta_{\max \alpha}(u)$ to be the maximum β such that $u \in (\alpha, \beta)$ -core.

We call the set of all $\alpha_{\max \beta}(v)$ and $\beta_{\max \alpha}(u)$ values the *bi-core numbers*. The *bi-core decomposition* problem is to compute $\beta_{\max \alpha}(u)$ for every $u \in U$ and every $\alpha \in [1, d_{\max_u}]$, and symmetrically, $\alpha_{\max \beta}(v)$ for every $v \in V$ and every $\beta \in [1, d_{\max_v}]$.

Note that with the bi-core numbers, we can easily determine whether any $u \in U$ is in the (α, β) -core for any α and β . If $\beta_{\max \alpha}(u) \geq \beta$, then $u \in (\alpha, \beta)$ -core. Similarly, for any $v \in V$, if $\alpha_{\max \beta}(v) \geq \alpha$, then $v \in (\alpha, \beta)$ -core.

7.4 Parallel Bi-core Decomposition

In this section, we introduce our parallel bi-core decomposition algorithm, which is inspired by Liu *et al.*'s sequential algorithm [215]. The goal of the algorithm is to compute the $\alpha_{\max \beta}(v)$ values for every β and vertex $v \in V$, and to compute the $\beta_{\max \alpha}(u)$ values for every α and vertex $u \in U$. Note that $\alpha_{\max \beta}(v) = \alpha$ if $v \in (\alpha, \beta)$ -core but $v \notin (\alpha + 1, \beta)$ -core. Symmetrically, $\beta_{\max \alpha}(u) = \beta$ if $u \in (\alpha, \beta)$ -core but $u \notin (\alpha, \beta + 1)$ -core. Thus, a peeling-based subroutine is often used to solve this problem, to discover successive cores [6, 107].

7.4.1 Background

Liu *et al.*'s [215] algorithm computes $\alpha_{\max \beta}(v)$ and $\beta_{\max \alpha}(u)$ values by calling a peeling subroutine PEEL-FIX- β for every $\beta' \in [1, \delta]$, where δ is the maximum k -core number of the graph. They also perform a symmetric subroutine PEEL-FIX- α for every $\alpha' \in [1, \delta]$. Due to the symmetry of these two subroutines, we only discuss PEEL-FIX- β .

PEEL-FIX- β takes as input the fixed β' value, and increases the α value of the (α, β') -core from 1 to $\max_\alpha(\beta')$ while iteratively peeling vertices no longer within the current (α, β') -core. In other words, for each α from 1 to $\max_\alpha(\beta')$, the algorithm iteratively peels vertices not in each successive core. When peeling a vertex v to discover the $(\alpha + 1, \beta')$ -core, they update $\alpha_{\max \beta'}(v) \leftarrow \alpha$, because it is the highest α value for which $v \in (\alpha, \beta')$ -core. Similarly, if they discover that $u \in (\alpha, \beta')$ but $u \notin (\alpha, \beta' + 1)$, they record β' as the value of $\beta_{\max \alpha}(u)$. A symmetric peeling subroutine PEEL-FIX- α is called for every $\alpha' \in [1, \delta]$. Liu *et al.* prove that these peeling subroutines correctly compute all $\alpha_{\max \beta}$ values and $\beta_{\max \alpha}$ values.

7.4.2 Parallel Bi-core Decomposition Algorithm

The sequential nature of Liu *et al.*'s [215] algorithm limits its practical applicability to large graphs. While Liu *et al.* [215] also provide a parallel version of their algorithm, their parallel algorithm only parallelizes across rounds of peeling (calls to subroutines PEEL-FIX- α and PEEL-FIX- β), and does not parallelize the peeling process itself. As a result, their parallel algorithm has a high span of $O(m)$. We present in this section a parallel bi-core decomposition algorithm, and we prove that our algorithm is work-efficient and has span $O(\rho \log n)$ *w.h.p.*, where ρ is the peeling complexity, or the maximum number of rounds needed to empty the graph in any of the peeling subroutines, where in each round, we peel all vertices with the minimum induced degree.

Our parallel algorithm is also based on a peeling paradigm, and in particular, we parallelize the subroutines PEEL-FIX- α and PEEL-FIX- β , which we call PAR-PEEL-FIX- α and PAR-PEEL-FIX- β respectively. Our parallel peeling subroutines compute the exact same $\alpha_{\max \beta}$ and $\beta_{\max \alpha}$ values as Liu *et al.*'s sequential peeling subroutine, so as a result, the correctness of our algorithm follows from the correctness of Liu

et al.'s algorithm. Because the two subroutines are symmetric, we only discuss PAR-PEEL-FIX- β here. PAR-PEEL-FIX- β takes as input a fixed β' . Then, it increases a variable α from 1 to $\max_{\alpha}(\beta')$ while peeling all vertices $u \in U$ where $u \in (\alpha, \beta')$ -core but $u \notin (\alpha + 1, \beta')$ -core in parallel in each iteration when α is increased. The order in which these vertices are peeled does not matter. Thus, this parallel peeling step computes the same result as Liu *et al.*'s sequential peeling step, which peels the vertices sequentially. Note that all such vertices $u \in U$ have induced degree satisfying $\deg_{\text{in}}(u) \leq \alpha$ for the current α value. Then, we peel all vertices $v \in V$ where $v \in (\alpha, \beta')$ -core but $v \notin (\alpha + 1, \beta')$ -core. Every peeled $v \in V$ must have induced degree satisfying $\deg_{\text{in}}(v) < \beta'$, so for each $v \in V$ that is peeled, we update the $\alpha_{\max \beta'}(v)$ value to be the current α value. We then update the α value to be the minimum value α' such that $\alpha' \geq \alpha$ and the (α', β') -core is nonempty.

Two challenges are involved in this process:

1. Finding the minimum α' such that there exists vertices $u \in U$ with induced degree $\deg_{\text{in}}(u) \leq \alpha'$, and returning this set of vertices in parallel.
2. Peeling a set of vertices from one partition and updating their neighbors' degrees in parallel.

To search for the minimum α' such that there exists u with $\deg_{\text{in}}(u) \leq \alpha'$ and to query for this set of vertices in parallel, we store all vertices U in a parallel bucketing structure Julienne by Dhulipala *et al.* [95]. Julienne organizes each $u \in U$ into buckets based on its induced degree, where vertices with $\deg_{\text{in}}(u) > \alpha$ are stored in the bucket indexed by $\deg_{\text{in}}(u)$, and vertices with $\deg_{\text{in}}(u) \leq \alpha$ are stored in a single bucket indexed by α . Finding the minimum α' and its corresponding vertices is then equivalent to finding the lowest-indexed nonempty bucket. Julienne supports this operation $\text{NEXT-BUCKET}(\alpha) \rightarrow (\alpha', U_{\alpha'})$, where α is the bucket index to begin the search from, α' is the next lowest index corresponding to a nonempty bucket, and $U_{\alpha'}$ is the set of vertices inside the bucket with index α' . The operation has $O(\log n)$ span per query and $O(n)$ work over all queries in a subroutine of PAR-PEEL-FIX- β [95]. Julienne also supports updating the degrees of k vertices in $O(k)$ work and $O(\log n)$ span w.h.p. [95].

We now discuss how we resolve the second challenge in more detail. The pseudocode of our parallel bi-core decomposition algorithm is given in Algorithm 7.1.

First, we discuss the subroutine PAR-DEL-UPDATE, which updates the degrees of vertices after peeling a given set of vertices. PAR-DEL-UPDATE takes as input an array of vertices X_{del} and then peels all of these vertices in parallel. On Lines 7–12, we construct an array Y_{update} that stores all neighbors y of X_{del} . Note that if y is incident to multiple vertices in X_{del} , it appears the same number of times in Y_{update} . This array can be constructed in parallel using a parallel PREFIX-SUM on the degrees of the vertices in X_{del} . On Line 12, HISTOGRAM aggregates Y_{update} and returns a sequence of pairs $(y, count)$. For every y , $count$ is the number of its occurrences in Y_{update} . On Lines 13–14, we iterate through each such y and decrease its degree by its corresponding $count$. Note that our pseudocode removes x from G on Line 9 for simplicity, but for our theoretical bounds, it is sufficient to mark removed vertices in a separate array and ignore them when traversing the graph in later iterations.

Now, we discuss the main subroutine, PAR-PEEL-FIX- β . Note that PAR-PEEL-

Algorithm 7.1 – Parallel Bi-core Decomposition

```

1: procedure PAR-BI-CORE( $G$ )
2:   parfor  $\alpha' = 1$  to  $\delta$  do
3:     PAR-PEEL-FIX- $\alpha$ ( $G$ ,  $\alpha'$ )
4:   parfor  $\beta' = 1$  to  $\delta$  do
5:     PAR-PEEL-FIX- $\beta$ ( $G$ ,  $\beta'$ )
6: procedure PAR-DEL-UPDATE( $G$ ,  $X_{\text{del}}$ )
7:    $Y_{\text{update}} \leftarrow []$                                  $\triangleright$  Create an empty array
8:   parfor all  $x$  in  $X_{\text{del}}$  do
9:     remove  $x$  from  $G$ 
10:    parfor all  $y$  in  $N_{\text{in}}(x)$  do
11:      add  $y$  to  $Y_{\text{update}}$                              $\triangleright$  Record  $y$  in parallel for degree update
12:     $Y_{\text{hist}} \leftarrow \text{HISTOGRAM}(Y_{\text{update}})$          $\triangleright$  Count number of occurrence of each vertex
13:    parfor all  $(y, count)$  in  $Y_{\text{hist}}$  do
14:       $\deg_{\text{in}}(y) \leftarrow \deg_{\text{in}}(y) - count$ 
15:    return  $Y_{\text{update}}$ 

16: procedure PAR-PEEL-FIX- $\beta$ ( $G$ ,  $\beta'$ )
17:   PAR-DEL-UPDATE( $G$ ,  $\{v | \deg_{\text{in}}(v) < \beta'\}$ )     $\triangleright$  Remove all vertices in  $V$  with degree
   <  $\beta'$ 
18:   Store vertices in  $U$  into buckets  $\triangleright$  Construct bucketing structure from vertices in  $U$ 
   based on their degrees
19:   while buckets is not empty do
20:      $(\alpha, U_{\text{del}}) \leftarrow \text{buckets.NEXT-BUCKET}(\alpha)$      $\triangleright$  Extract the next set of vertices with
   minimum degree
21:     parfor all  $u$  in  $U_{\text{del}}$  do
22:       parfor  $i = 1$  to  $\alpha$  do
23:         WRITE-MAX( $\beta_{\max i}(u), \beta'$ )                       $\triangleright$  Update  $\beta_{\max i}(u)$ 
24:          $V_{\text{update}} \leftarrow \text{PAR-DEL-UPDATE}(G, U_{\text{del}})$      $\triangleright$  Peel vertices  $u \in U$  with induced degree
    $\leq \alpha$ 
25:          $V_{\text{del}} \leftarrow \text{FILTER}(V_{\text{update}}, \deg_{\text{in}}(v) < \beta')$    $\triangleright$  Determine vertices  $v \in V$  no longer in
    $(\alpha, \beta')$ -core
26:         parfor all  $v$  in  $V_{\text{del}}$  do
27:           WRITE-MAX( $\alpha_{\max \beta'}(v), \alpha$ )                   $\triangleright$  Update  $\alpha_{\max \beta'}(v)$ 
28:            $U_{\text{update}} \leftarrow \text{PAR-DEL-UPDATE}(G, V_{\text{del}})$      $\triangleright$  Remove vertices  $v \in V$  no longer in
    $(\alpha, \beta')$ -core
29:            $\text{buckets.UPDATE-VERTICES}(U_{\text{update}})$              $\triangleright$  Update vertices  $u \in U$  with changed
   degrees in the bucketing structure
30: procedure PAR-PEEL-FIX- $\alpha$ ( $G$ ,  $\alpha'$ )
31:   symmetric to PAR-PEEL-FIX- $\beta$ 

```

FIX- α is symmetric. We refer to Figure 7.1 for an example of the computations in PAR-PEEL-FIX- β , with $\beta' = 2$ and starting with the graph in Step 0.

On Line 17 in PAR-PEEL-FIX- β , we peel all $v \in V$ with degree less than β' using the subroutine PAR-DEL-UPDATE. In Figure 7.1, this removes vertices 0 and 5, resulting

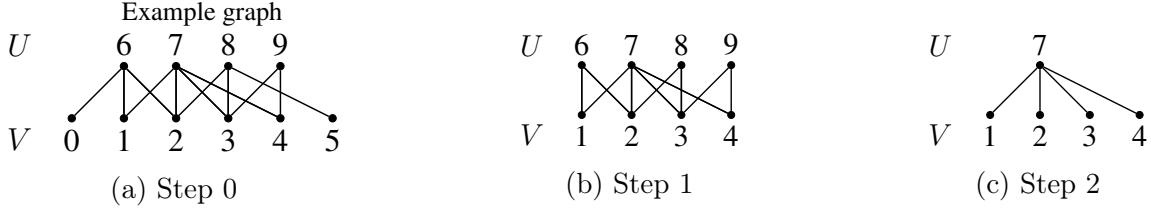


Figure 7.1: An example peeling process of PAR-PEEL-FIX- β where the input β' is 2. From Step 0 to Step 1, all vertices in V with induced degree < 2 are removed. From Step 1 to Step 2, all vertices in U with degree ≤ 2 are removed.

in the graph at Step 1. At this point, all remaining vertices are in the $(0, \beta')$ -core. On Line 18, we initialize the parallel bucketing structure *Julienne* over the vertices in U , which we call *buckets*. We call NEXT-BUCKET on *buckets* on Line 20 to obtain the next nonempty bucket of vertices, which we store into U_{del} . We also update the α value. In our example graph in Figure 7.1, $U_{\text{del}} = \{6, 8, 9\}$ and the new α value is 2. Importantly, U_{del} records all u with induced degree $\text{deg}_{\text{in}}(u) \leq \alpha$. Note that for all $u \in U_{\text{del}}$, $u \in (\alpha, \beta')$ -core. Thus, on Lines 21–23, we can update the $\beta_{\max \alpha}(u)$ values to β' for all $u \in U_{\text{del}}$, which is done in parallel using a WRITE-MAX.

On Line 24, we peel all vertices in the current bucket, U_{del} . In Figure 7.1, this results in the graph in Step 2, where $V_{\text{update}} = \{1, 2, 3, 4\}$. On Line 25, we store in V_{del} all $v \in V_{\text{update}}$, where $\text{deg}_{\text{in}}(v) < \beta'$. These vertices are no longer in (α, β') -core and must be removed. In the example, $V_{\text{del}} = \{1, 2, 3, 4\}$. Because these vertices $v \in V_{\text{del}}$ satisfy $v \in (\alpha, \beta')$ -core but $v \notin (\alpha + 1, \beta')$ -core, we update the $\alpha_{\max \beta'}(v)$ values to α for each vertex $v \in V_{\text{del}}$ on Lines 26–27. Then, Line 28 calls PAR-DEL-UPDATE to peel all vertices in V_{del} . Finally, on Line 29, we use the bucketing structure to update the degrees of vertices in U_{update} , which consists of all $u \in U$ whose degree is affected by peeling V_{del} ; UPDATE-VERTEX moves $u \in U_{\text{update}}$ to new buckets corresponding to their new degrees. For vertices in U_{update} with an updated degree $< \alpha$, we set their degrees to α , so they are peeled in the next round of peeling. In our example, $U_{\text{update}} = \{7\}$, and the degree of vertex 7 is updated to 2. In the second iteration of the while loop, the remaining vertex 7 in the U partition is removed as $\text{deg}_{\text{in}}(7) = 0 \leq 2 = \alpha$, thus completing the peeling procedure.

Analysis. We now show that PAR-PEEL-FIX- β takes $O(m)$ work. First, across all iterations of the while loop, NEXT-BUCKET takes $O(m)$ work [95]. UPDATE-VERTICES takes $O(1)$ work to update a single vertex, and since each vertex x is updated at most $\text{deg}_{\text{in}}(x)$ times, the overall work of all updates is bounded by $O(\sum_{x \in U \cup V} \text{deg}_{\text{in}}(x)) = O(m)$.

Across all calls of PAR-DEL-UPDATE, each vertex is peeled exactly once. Since we traverse the neighbor of each vertex in PAR-DEL-UPDATE once, the total work of PAR-DEL-UPDATE in one call of PAR-PEEL-FIX- β is $O(m)$.

The work of updating the $\alpha_{\max \beta}$ values totals $O(n)$, since each vertex can only appear in V_{del} exactly once. The work of updating the $\beta_{\max \alpha}$ values is bounded by $O(m)$ because for each vertex u , the maximal α value for which $\beta_{\max \alpha}(u) > 0$ is bounded by $\text{deg}_{\text{in}}(u)$. Thus, for each u , Line 23 is only executed $O(\text{deg}_{\text{in}}(u))$

times, which totals to $O(\sum_{u \in U} \deg_{\text{in}}(u)) = O(m)$. FILTER, over all calls, also takes $O(m)$ work. Thus, PAR-PEEL-FIX- β has work $O(m)$, and overall, PAR-BI-CORE takes $O(\delta m) = O(m^{3/2})$ work.

Now, we analyze the span complexity. First, note that PAR-DEL-UPDATE has span $O(\log n)$ w.h.p.; this is because PREFIX-SUM and HISTOGRAM both have $O(\log n)$ span w.h.p. Each iteration of the while loop on Line 19 has span $O(\log n)$ w.h.p. because FILTER, PAR-DEL-UPDATE, UPDATE-VERTICES, and NEXT-BUCKET all take $O(\log n)$ span w.h.p. The number of iterations of the while loop is ρ by definition. The span of PAR-PEEL-FIX- β is therefore $O(\rho \log n)$ w.h.p., and overall, PAR-BI-CORE has span $O(\rho \log n)$ w.h.p.

7.4.3 Peeling Space Pruning Optimization

In this section, we introduce a peeling space pruning optimization to our algorithm, which is also applicable to the sequential bi-core decomposition algorithm by Liu *et al.* [215]. Liu *et al.*'s algorithm performs the peeling subroutine for every α , from $\alpha = 1$ to $\alpha = \max_\alpha(\beta')$, for each $1 \leq \beta' \leq \delta$. Then, it also performs the same subroutine for every β , from $\beta = 1$ to $\beta = \max_\beta(\alpha')$, for each $1 \leq \alpha' \leq \delta$. We observe that, in the process of peeling, all (α, β) -cores with $1 \leq \alpha \leq \delta$ and $1 \leq \beta \leq \delta$ are peeled twice, once when we perform peeling along increasing α values for different β' , and again when we perform peeling along increasing β values for different α' .

To avoid this repetition, we modify Algorithm 7.1. We will discuss the modification considering increasing α values, but the same can be applied to increasing β values. Instead of peeling from the $(1, \beta')$ -core, we modify PAR-PEEL-FIX- $\beta(G, \beta')$ to peel from the (β', β') -core. In other words, the algorithm starts iteratively, increasing the α value from β' to $\max_\alpha(\beta')$ and removing vertices no longer in the current (α, β') -core at the same time. Notably, we confine α to $\beta' \leq \alpha \leq \max_\alpha(\beta')$ as opposed to $1 \leq \beta \leq \max_\alpha(\beta')$ as used in Liu *et al.*'s algorithm and in Algorithm 7.1.

We illustrate this optimization with an example in Figure 7.2, using the graph from Figure 7.1. Each grid intersection in Figure 7.2 represents an (α, β) -core. Edges represent a single-step peeling operation from the (α, β) -core to the $(\alpha, \beta + 1)$ -core (upward), or to the $(\alpha + 1, \beta)$ -core (rightward). The labeled numbers on an edge represent the indices of vertices that would be deleted by that specific peeling operation. The circled nodes represent (α, β) -cores that are nonempty, and the boxed node represents the (δ, δ) -core. Every core corresponding to a grid position that is not drawn is empty. The circled nodes form the boundary of the peeling space.

The peeling operations performed by Algorithm 7.1 can be visualized by the blue highlighted paths in Figure 7.3a. For $\beta' = 1$, we perform α -side peeling from $\alpha = 1$ to $\alpha = 4$. For $\beta' = 2$, we again increase α from 1 to 3 while iteratively removing vertices not within the current bi-core. With the proposed optimization, for $\beta' = 2$, we only perform peeling from $\alpha = 2$ to $\alpha = 3$, starting from the (β', β') -core, or the $(2, 2)$ -core in this case. This is represented by the blue highlighted peeling paths in Figure 7.3b.

To show the correctness of the optimized algorithm, we divide the peeling space into three parts: part A where all of the (α, β) -cores satisfy $\alpha > \beta$, part B where all

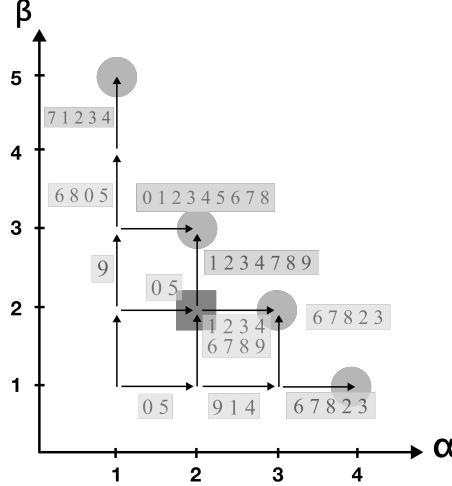


Figure 7.2: This shows the peeling space of the example graph in Figure 7.1, and is discussed in more detail in Section 7.4.3.

of the (α, β) -cores satisfy $\beta > \alpha$, and part C with the diagonal (x, x) -cores. Note that part A of the peeling space corresponds to the section of the peeling space below the diagonal (x, x) -cores, and part B corresponds to the section above the diagonal (x, x) -cores. Thus, for the optimized algorithm, when peeling along increasing α values, it operates in part A of the peeling space; when peeling along increasing β values it operates in part B.

First, we note that the correct $\alpha_{\max \beta}(v)$ values are computed for all vertices v with $(\alpha_{\max \beta}(v), \beta)$ -cores in parts A and C of the peeling space. For a fixed β , if $v \in (\alpha, \beta)$ -core but $v \notin (\alpha + 1, \beta)$ -core where $\alpha \geq \beta$, then $\alpha_{\max \beta}(v)$ is recorded correctly to be α as we perform the peeling process along α values from $\alpha = \beta$ to $\alpha = \max_\alpha(\beta)$. This update is performed on Line 27 of Algorithm 7.1.

Now, we show that the optimized algorithm computes the correct $\alpha_{\max \beta}(v)$ values for all vertices v with $(\alpha_{\max \beta}(v), \beta)$ -cores in part B of the peeling space. When peeling along increasing β values with a fixed α' such that $\alpha' = \alpha_{\max \beta}(v)$, the algorithm removes v at the (α', β') -core, where $\beta' > \alpha'$ and $\beta' \geq \beta$. Consider the update given by Line 23 in subroutine PAR-PEEL-FIX- β of Algorithm 7.1. Using the symmetric update in the subroutine PAR-PEEL-FIX- α , we update $\alpha_{\max \beta}(v) \leftarrow \max(\alpha_{\max \beta}(v), \alpha')$ for all $\beta \in [1, \beta']$. Thus, $\alpha_{\max \beta}(v)$ is set to its correct value, α' .

Because parts A, B, and C form the entire peeling space, we have shown that for all β and v , $\alpha_{\max \beta}(v)$ is correctly computed. Symmetric arguments apply for all $\beta_{\max \alpha}(u)$, to show that the overall optimized algorithm is correct.

7.5 P-completeness of Bi-core Decomposition

The span of our Algorithm 7.1 from Section 7.4.2 is not polylogarithmic. However, we show here that the bi-core decomposition problem in general is P-complete, which means that the problem is inherently sequential and cannot be solved with polyloga-

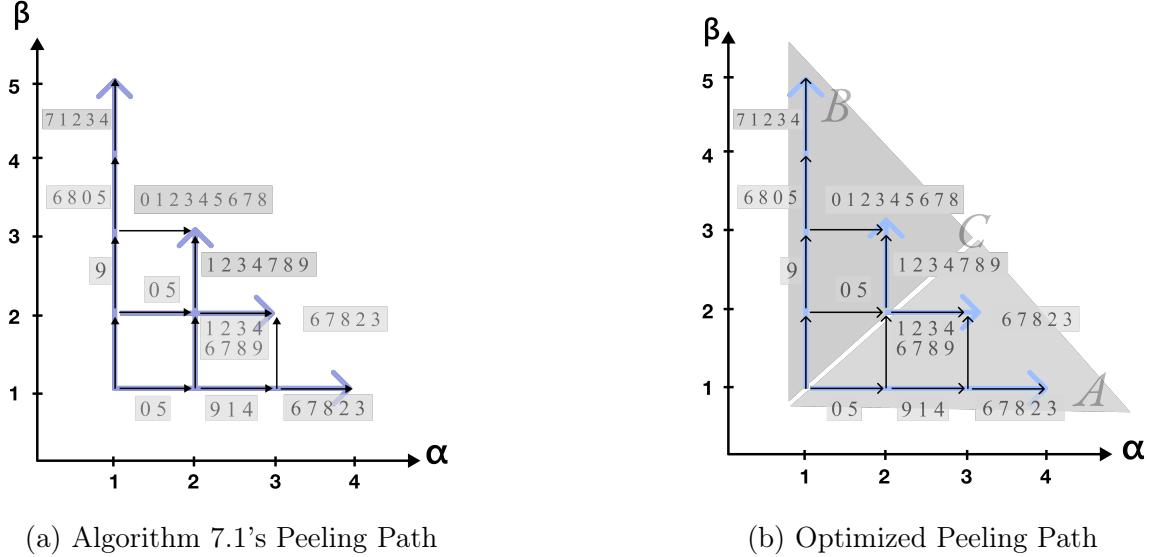


Figure 7.3: This figure compares the unoptimized and optimized peeling paths of the example graph in Figure 7.1. The top figure shows the unoptimized peeling paths, while the bottom figure shows the optimized peeling paths.

rithmic span if we accept the standard assumption that $P \neq NC$ [177]. Note that NC contains problems that can be solved in polylogarithmic span, or more specifically, problems that can be solved in polylogarithmic time on a parallel machine with a polynomial number of processors [198]. We show that for small enough α and β , the bi-core decomposition problem is in NC . More formally, we prove these results for the decision version of the problem, which is, given α and β , and a bipartite graph, decide if the (α, β) -core in the graph is nonempty. This is a generalization of the k -core decision problem on a bipartite graph: given k , decide if the (k, k) -core is nonempty. We show the P-completeness of bi-core decision problem using a reduction from the k -core decision problem. Note that the k -core decision problem is P-complete for $k > 2$ and in NC for $k = 2$ [17].

Theorem 7.1. *The (α, β) -core decomposition problem is P-complete if and only if $\alpha \geq 3$ and $\beta \geq 2$, or $\beta \geq 3$ and $\alpha \geq 2$. Otherwise, it is in NC .*

When $\alpha = 2$ and $\beta = 2$. For $\alpha = 2$ and $\beta = 2$, the $(2, 2)$ -core decomposition problem is equivalent to the k -core decomposition problem on the bipartite graph with $k = 2$, which is in NC [17].

When $\alpha = 1$ or $\beta = 1$. If $\alpha = 1$, then the $(1, \beta)$ -core decomposition problem is equivalent to finding all $v \in V$ such that $\deg(v) \geq \beta$. These vertices and their neighbors in U form the $(1, \beta)$ -core, and can be found in $O(1)$ span. Similarly, we can find the $(\alpha, 1)$ -core for any α in $O(1)$ span.

When $\alpha \geq 3$ and $\beta \geq 3$. We perform the reduction from the k -core decision problem in a graph G (that is not necessarily bipartite) by constructing a bipartite graph G' such that the k -core decision problem on G is equivalent to the (k, k) -core decision problem on G' . Let G' consist of two partitions U and V , where each partition is a

copy of all vertices of G . In other words, each $x \in G$ is copied to $x_u \in U$ and $x_v \in V$ in G' . We form an edge (x_u, y_v) in G' if (x, y) is an edge in G .

Now, we show that for any k , the k -core is nonempty in G if and only if the (k, k) -core is nonempty in G' . If the k -core of G is nonempty and comprises a vertex subset W , then for $w \in W$, there exists at least k edges of the form (w, p) , where $p \in W$. Let W_U be the set of all vertices $x_u \in U$ that represent copies of vertices $x \in W$, and symmetrically, let W_V be the set of all $x_v \in V$ that represent copies of $x \in W$. Consider $W' = W_U \cup W_V$ in G . Since W_U and W_V are copies of W , and each $w \in W$ has at least k edges of the form (w, p) , we know each $w_U \in W_U$ is incident to at least k edges of the form (w_U, p_V) where $p_V \in W_V$ by construction. A similar argument applies for each $w_V \in W_V$. Therefore, W' forms a (k, k) -core on the bipartite graph, and the (k, k) -core is nonempty in G' .

Reversely, if the (k, k) -core in G' is nonempty, we show that the k -core in G is nonempty. Due to the symmetry of the U and V partitions, if $w_U \in (k, k)$ -core, then $w_V \in (k, k)$ -core. Therefore, if the (k, k) -core of G' is W' , then $W' = W_U \cup W_V$, and W_U and W_V are mirror images of each other. Let W be the vertex subset in G that corresponds to W_U and W_V . We show that it is a k -core in G . For each vertex $w_U \in W_U$ incident to edges of the form (w_U, p_V) where $p_V \in W_V$, its corresponding vertex w in W is incident to the corresponding edges of the form (w, p) , and $p \in W$ because $p_V \in W_V$. Since the induced degree $\deg(w_U) \geq k$ for each $w_U \in W_U$ on the subset $W_U \cup W_V$, we must have that the induced degree $\deg(w) \geq k$ for each $w \in W$ on the subset W . Therefore, W forms a k -core of graph G , and since W is nonempty, the k -core of graph G is nonempty.

Thus, we have shown an NC reduction from the k -core problem to the bi-core problem, since constructing G' takes work $O(m)$ and span $O(1)$. Since the k -core decomposition is P-complete for $k \geq 3$ [17], we have shown that the bi-core decomposition is P-complete for $\alpha \geq 3$ and $\beta \geq 3$.

When one of $\alpha, \beta = 2$. We now consider the case where $\alpha = 2$ and $\beta \geq 3$; the reverse where $\beta = 2$ and $\alpha \geq 3$ is symmetric. We show that deciding whether the $(2, \beta)$ -core is nonempty has a reduction from the k -core equivalent with $k = \beta$. Consider a graph G that is not necessarily bipartite, and the k -core problem on this graph. We construct a bipartite graph G' by replacing every edge in G with a path of length 2. In other words, for each edge (x, y) , we add a *middle vertex* z and replace the edge with edges (x, z) and (z, y) . We let U be the set of middle vertices, and V be the set of original vertices in G , and we note that U and V form the bipartitions of G' . Also, note that we can construct G' in $O(1)$ span. Now, deciding if the $(2, \beta)$ -core nonempty in G' is equivalent to deciding if the k -core of G is nonempty where $k = \beta$, because every vertex in V has the same degree as in the original graph, and every vertex in U has degree exactly 2, so they will always be included in a β -core for $\beta \geq 3$. Given this reduction, the problem of bi-core decomposition is P-complete for $\alpha = 2$ and $\beta \geq 3$, and symmetrically for $\beta = 2$ and $\alpha \geq 3$.

Thus, we have shown that the bi-core decomposition problem is in NC if and only if $\alpha = 1$ or $\beta = 1$, or $\alpha = \beta = 2$. Otherwise, it is P-complete.

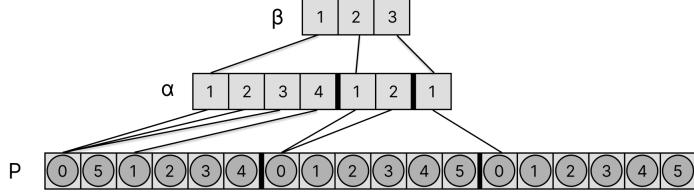


Figure 7.4: This figure shows the index structure $\mathbb{P}\mathbb{I}^V$ for the example graph in Figure 7.1. The first level is indexed by β values, and the second level is indexed by α values, which then point to the corresponding set of vertices in the core. The lines between levels represent pointers in the structure.

7.6 Parallel Bi-core Index Structure

The algorithm in Section 7.4 computes the $\alpha_{\max \beta}(v)$ values for every β and vertex $v \in V$, and the $\beta_{\max \alpha}(u)$ values for every α and vertex $u \in U$. To enable fast queries for the set of vertices inside a particular (α, β) -core, we parallelize the sequential index structure by Liu *et al.* [215]. Their index structure is constructed sequentially using the computed $\alpha_{\max \beta}(v)$ and $\beta_{\max \alpha}(u)$ values, and allows for queries of (α, β) -cores in time proportional to the number of vertices in the core. Our parallel index construction algorithm takes $O(m)$ work and $O(\log n)$ span w.h.p., and our parallel query algorithm takes linear work in the size of the core and $O(1)$ span.

We define $\mathbb{P}\mathbb{I}^U$ and $\mathbb{P}\mathbb{I}^V$ to be the parallel index structures for U and V respectively. Note that $\mathbb{P}\mathbb{I}^V$ is our parallel version of \mathbb{I}^V from Liu *et al.*'s work [215], and symmetrically, $\mathbb{P}\mathbb{I}^U$ is the parallel version of \mathbb{I}^U , where \mathbb{I}^V and \mathbb{I}^U are Liu *et al.*'s sequential index structures. Because they are symmetric, we only discuss $\mathbb{P}\mathbb{I}^V$ here. Liu *et al.* define $\mathbb{I}_{\alpha, \beta}^V$ to be the set containing all vertices $v \in V$ such that $v \in (\alpha, \beta)$ -core but $v \notin (\alpha + 1, \beta)$ -core; notably, each $\mathbb{I}_{\alpha, \beta}^V$ points to a location in memory in their index structure \mathbb{I}^V . In our parallelization, we define $\mathbb{P}\mathbb{I}_{\alpha, \beta}^V$ in the same way, and each $\mathbb{P}\mathbb{I}_{\alpha, \beta}^V$ points to a location in memory in our index structure $\mathbb{P}\mathbb{I}^V$.

A key difference is that in Liu *et al.*'s work, each set $\mathbb{I}_{\alpha, \beta}$ is stored separately. In our parallelization, we store all sets $\mathbb{P}\mathbb{I}_{\alpha, \beta}^V$ contiguously in memory, ordered first by β and then by α . We call this array P . In order to access each set $\mathbb{P}\mathbb{I}_{\alpha, \beta}^V$ efficiently, we define a 2D array M , where each $M[\beta][\alpha]$ corresponds to a set $\mathbb{P}\mathbb{I}_{\alpha, \beta}^V$ and contains the starting index of that set in P . Note that unlike our other data structures, we define M to be 1-indexed, for the sake of clarity in querying for the (α, β) -core. By definition, $\mathbb{P}\mathbb{I}_{\alpha, \beta}^V$ consists of all vertices in P in the range $[M[\beta][\alpha], M[\beta][\alpha + 1] - 1]$. Thus, the range $[M[\beta][\alpha], M[\beta + 1][0] - 1]$ gives precisely the vertices that correspond to $\bigcup_{i=\alpha}^{\max_\alpha(\beta)} \mathbb{P}\mathbb{I}_{i, \beta}^V$, which is the set of all vertices in the (α, β) -core. Figure 7.4 shows an example of the $\mathbb{P}\mathbb{I}^V$ index structure for the example graph from Figure 7.1.

To efficiently query the (α, β) -core, we return all vertices in P in the range $[M[\beta][\alpha], M[\beta + 1][0] - 1]$. This takes $O(|(\alpha, \beta)\text{-core}|)$ work and $O(1)$ span.

Algorithm 7.2 – Parallel Index Construction

```

1: procedure BUILD-V-INDEX( $\alpha_{\max \beta}(v)$  values)
2:    $P \leftarrow$  array of  $(\beta, \alpha_{\max \beta}(v), v)$  for every  $v \in V$ 
3:   RADIX-SORT( $P$ )                                      $\triangleright$  Sorts  $P$  by first  $\beta$ , and then  $\alpha$ 
4:   Initialize  $TPT$                                  $\triangleright$  Creates an empty  $TPT$  of length  $|P|$ 
5:   parfor  $i = 0$  to  $P.size - 1$  do
6:     if  $i = 0$  or  $P[i - 1].\beta \neq P[i].\beta$  or  $P[i - 1].\alpha \neq P[i].\alpha$  then
7:        $TPT[i] \leftarrow i$                           $\triangleright$  Records index where  $\beta$  or  $\alpha$  changes value
8:     FILTER( $TPT$ , element is not empty)            $\triangleright$  Filter out empty indices
9:     Initialize  $FPT$                             $\triangleright$  Creates an empty  $FPT$  of length  $|P|$ 
10:    parfor  $i = 0$  to  $TPT.size - 1$  do
11:      if  $i = 0$  or  $P[TPT[i - 1]].\beta \neq P[TPT[i]].\beta$  then
12:         $FPT[i] \leftarrow i$                           $\triangleright$  Records index of  $TPT$  array where  $\beta$  changes
13:      FILTER( $FPT$ , element is not empty)            $\triangleright$  Filter out empty indices
14:      Initialize  $M$                              $\triangleright$  Creates empty  $M$  array with 1st dimension =  $FPT.size$  and 2nd
         dimension =  $\max_\alpha(\beta)$ 
15:      parfor  $\beta = 1$  to  $FPT.size$  do
16:        parfor  $j = FPT[\beta - 1]$  to  $FPT[\beta] - 1$  do
17:          start  $\leftarrow TPT[j]$                        $\triangleright$  Obtains starting index of  $j^{\text{th}}$  block
18:           $M[\beta][P[start].\alpha] \leftarrow start$   $\triangleright$  Stores starting index;  $P[start].\alpha$  gives  $j^{\text{th}}$  block's
         corresponding  $\alpha$  value
19:         $M[\beta] \leftarrow \text{SUFFIX-MIN}(M[\beta])$ 
20:      return  $M$ 
21: procedure BUILD-U-INDEX( $\beta_{\max \alpha}(u)$  values)
22:   symmetric to BUILD-V-INDEX

```

7.6.1 Parallel Index Construction

In this section, we detail our index construction algorithm for index structure \mathbb{PI}^V . Note that the algorithm for constructing \mathbb{PI}^U is symmetric. The inputs to the construction algorithm are the $\alpha_{\max \beta}(v)$ values for every β and every vertex $v \in V$. The objective is to construct \mathbb{PI}^V , which consists of M and P .

First, we construct P . For every $v \in V$ and a β value such that $\alpha_{\max \beta}(v) > 0$, we store in an array a tuple $(\beta, \alpha_{\max \beta}(v), v)$. We then perform parallel RADIX-SORT on these tuples to obtain P . To construct M , we apply a parallel filter to find the indices of P at which the β or $\alpha_{\max \beta}(v)$ values change. We store these indices in an array, TPT (total pointer table). We also find indices where only the β values change, which we store in FPT (first pointer table). These two tables contain the required information to form M .

The pseudocode for our parallel index construction algorithm is in Algorithm 7.2. We now discuss our algorithm in more detail. First, on Line 2, we store a list of tuples $(\beta, \alpha_{\max \beta}(v), v)$ in P . This is the list of all possible combinations of β and $v \in V$, with the corresponding $\alpha_{\max \beta}(v)$ value, where $\alpha_{\max \beta}(v) > 0$. We perform parallel RADIX-SORT on P based on the ordering of first the β values and then the α values, on Line 3; note that the third value v in the tuple need not be sorted. For our ex-

ample graph in Figure 7.1, the sorted $P = [(1, 3, 0), (1, 3, 5), (1, 4, 1), (1, 4, 2), (1, 4, 3), (1, 4, 4), (2, 2, 0), (2, 2, 1), (2, 2, 2), (2, 2, 3), (2, 2, 4), (2, 2, 5), (3, 1, 0), (3, 1, 1), (3, 1, 2), (3, 1, 3), (3, 1, 4), (3, 1, 5)]$, as illustrated by the example index structure in Figure 7.4. Then, we initialize an empty TPT on Line 4 with the same size as P . The parallel for-loop on Lines 5–8 finds all indices at which either the β value or the α value in P changes. This is accomplished by marking the indices where consecutive values differ on Line 7 and filtering out all of the unmarked index positions on Line 8. Constructing TPT essentially breaks up the array P into blocks where each block has the same β and α values, and corresponds to set $\mathbb{P}\mathbb{I}_{\alpha,\beta}^V$. $TPT[i]$ records the starting index location of the i^{th} block. Lines 9–13 repeat the entire process, but for FPT , to filter out index positions where only the β value of P changes. Note that the indices stored in FPT are not indices of positions in P . Instead, they are indices of positions in TPT ; this is to form the second level of the 2D array M . $[FPT[\beta - 1], FPT[\beta] - 1]$ gives the range of blocks defined by TPT with this particular β value, and it corresponds to set $\bigcup_{0 \leq \alpha \leq \max_\alpha(\beta)} \mathbb{P}\mathbb{I}_{\alpha,\beta}^V$.

Finally, based on FPT and TPT , we create M in the following manner. We obtain $M[\beta]$ for each β value independently. Line 16 iterates in parallel over the blocks that have the particular β value. For each block j , on Lines 17–18, we store its starting position $TPT[j]$ to $M[\beta][\alpha_j]$, where α_j is the α value corresponding with the j^{th} block. Thus, we have constructed $M[\beta][\alpha']$ for all (β, α') pairs such that α' equals $\alpha_{\max_\beta}(v)$ for some $v \in V$. The 2D array M created from our example graph is visually represented by the first two levels of the index structure in Figure 7.4.

For pairs (β, α) where α does not appear in $\alpha_{\max_\beta}(v)$ for some $v \in V$, we point $M[\beta][\alpha]$ to the same location as $M[\beta][\alpha']$ where α' is the smallest value $\geq \alpha$ that appears in $\alpha_{\max_\beta}(v)$ for some $v \in V$. We accomplish this by performing SUFFIX-MIN on $M[\beta]$ on Line 19.

Analysis. Since $P.\text{size} = O(m)$, and since we only need to sort each tuple $(\beta, \alpha_{\max_\beta}(v), v)$ by the first element β and second element $\alpha_{\max_\beta}(v)$, if we compress the pairs $(\beta, \alpha_{\max_\beta})$ into integer keys, the range of these keys is bounded by $O(\sum_{1 \leq \beta \leq \text{dmax}_V} \max_\alpha(\beta)) = O(m)$ [215]. The compression can be done since we know the $\max_\alpha(\beta)$ value for each β . Thus, we can construct a mapping from each pair (β, α) to an index by running a prefix sum over the $\max_\alpha(\beta)$ values for each $\beta \in [1, \text{dmax}_V]$. The mapping can then be stored in a parallel hash table to be used during the RADIX-SORT, which as such takes $O(m)$ work w.h.p. Lines 4–13 also take $O(m)$ work, proportional to the sizes of TPT and FPT . Lines 14–19 take $O(m)$ work, because we loop through M exactly once and the table takes $O(m)$ space, as in its sequential counterpart in [215].

Algorithm 7.2 has span $O(\log n)$ w.h.p., since RADIX-SORT, FILTER, SUFFIX-MIN, and hash table operations [148] are bounded by $O(\log n)$ span w.h.p.

7.7 Experiments

In this section, we provide a comprehensive evaluation of our parallel bi-core decomposition algorithm.

| Name | $ U $ | $ V $ | m | dmax | δ | ρ |
|--------------|-------|-------|--------|--------|----------|--------|
| Orkut | 2.78M | 8.73M | 327M | 318K | 466 | 12100 |
| Web Trackers | 27.7M | 12.7M | 140.6M | 11.57M | 437 | 4542 |
| TREC | 556K | 1.17M | 83.6M | 457K | 508 | 6029 |
| LiveJournal | 3.20M | 7.49M | 112M | 1.05M | 108 | 6831 |
| Reuters | 781K | 284K | 60.6M | 345K | 192 | 4767 |
| Epinions | 120K | 755K | 13.67M | 162K | 151 | 3049 |
| Flickr | 396K | 104K | 8.55M | 35K | 147 | 2300 |

Table 7.2: The graphs used in our experiments, along with the sizes, maximum degree (dmax), degeneracy (δ), and number of rounds required in peeling, or the bi-core peeling complexity, (ρ) are shown.

7.7.1 Experimental Setup

We experiment using real-world bipartite graphs from the KONECT graph database [199], the details of which are given in Table 7.2. As seen in Table 7.2, the bi-core peeling complexities, or ρ , of these real-world graphs are in general 3–4 orders of magnitude smaller than their numbers of edges m . This indicates that the $O(\rho \log n)$ span w.h.p. achieved by our parallel bi-core decomposition algorithm is significantly lower than the $O(m)$ span achieved by the previous state-of-the-art parallel algorithm [215].

We use Google Cloud Platform **c2-standard-60** instances, which are 30-core machines with two-way hyper-threading, with Intel 3.1 GHz Cascade Lake processors and 240 GB of memory; the processors have a maximum turbo clock-speed of 3.8 GHz.

7.7.2 Implementation and Other Optimizations

While our parallel bi-core decomposition algorithm (Algorithm 7.1) is theoretically efficient, it is practically slow due to the overhead incurred by the histogram-based PAR-DEL-UPDATE subroutine. We implemented the fully parallel algorithm and found it to be orders of magnitude slower than Liu *et al.*'s algorithm on certain graphs, and it is overall slower on all of the datasets that we experiment with. Thus, for a practically fast implementation, we do not parallelize the bi-core peeling process as described in Section 7.4. Instead, we only parallelize across different peeling processes, similar to Liu *et al.*'s parallel algorithm [215]. Our parallel implementation differs from Liu *et al.*'s work in that we utilize the Julienne bucketing structure [95] to search for the next set of vertices with minimum induced degree in each peeling iteration. Julienne was originally designed as a parallel bucketing structure, but we use a sequential version, since the additional parallelism does not improve our algorithm's performance. Liu *et al.*'s algorithm, on the other hand, uses a simple sequential search to find the next set of vertices with minimum induced degree. In addition, our parallel implementation includes the peeling space pruning optimization described in Section 7.4.3. We demonstrate in this section that our optimization techniques are effective.

We use Liu *et al.*'s sequential and parallel bi-core decomposition algorithms as baselines. In total, we perform our experimental analysis on the following bi-core decomposition algorithms:

1. LIU-SEQ: Liu *et al.*'s sequential algorithm [215];
2. LIU-PAR: Liu *et al.*'s parallel algorithm [215]; and
3. PAR: Our parallel algorithm, which differs from LIU-PAR in that we use Julianne [95] and the peeling space pruning optimization described in Section 7.4.3.

We also perform experiments on the following bi-core index construction and query algorithms. Note that Liu *et al.* do not provide parallel implementations for their bi-core index construction and query algorithms, so their sequential implementations are the state of the art.

1. LIU-CONS: Liu *et al.*'s sequential bi-core index construction algorithm [215];
2. LIU-QUERY: Liu *et al.*'s sequential bi-core query algorithm [215];
3. IND-CONS: Our parallel bi-core index construction algorithm as detailed in Algorithm 7.2; and
4. QUERY: Our parallel bi-core query algorithm, as described in Section 7.6.

We use the Graph Based Benchmark Suite (GBBS) [106] to implement our algorithms. All of our code is written in C++ and compiled with the `-O3` flag, and we use the work-stealing scheduler from PARLAYLIB by Blelloch *et al.* [46]. We perform each experiment three times and report the average running time.

Because QUERY and LIU-QUERY are extremely fast, we perform query experiments in batches of 10,000 bi-core queries per batch, and report the total time for the batch. For each (α, β) -core query in the batch, the (α, β) -core to be queried is uniformly at random selected from all non-empty bi-cores in the input graph. We test both QUERY and LIU-QUERY on the same set of randomly sampled (α, β) -cores.

We run our parallel algorithms, including PAR, IND-CONS, and QUERY, on 30 threads; using 60 hyper-threads, in general, does not improve their performances. However, we find that LIU-PAR benefits from hyper-threading, and we compare to LIU-PAR using 60 hyper-threads.

7.7.3 Bi-core Decomposition

In this section, we discuss the performance of the bi-core decomposition algorithms LIU-SEQ, LIU-PAR, and PAR.

Comparison to Prior Work. Figure 7.5 shows the comparison of our parallel bi-core decomposition implementation to the state-of-the-art sequential and parallel implementations [215]. Our parallel implementation PAR attains between 18.2–51.4x speedups over LIU-SEQ, with an average speedup of 27.9x. Compared to LIU-PAR running on 60 hyper-threads, PAR also achieves between 1.5–2.0x speedups over LIU-PAR, with an average speedup of 1.8x. However, we note that LIU-PAR runs out of memory when running on 60 hyper-threads for large graphs, specifically Orkut and Web Trackers. This is because LIU-PAR keeps a separate copy of the table storing the bi-core numbers (*i.e.* the $\alpha_{\max} \beta(v)$ and $\beta_{\max} \alpha(u)$ values) for each thread, so it consumes a significant amount of memory when running on a large number of threads. In comparison, our algorithm PAR keeps a single global copy of this table,

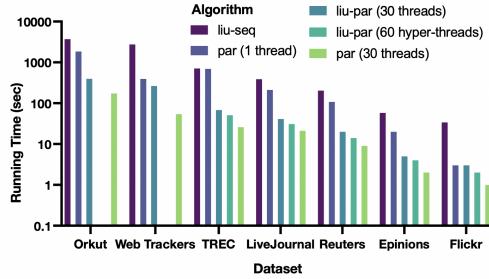


Figure 7.5: This figure compares the running time (in seconds) of various bi-core decomposition algorithms, namely LIU-PAR, LIU-SEQ, and PAR. LIU-PAR on 60 hyper-threads runs out of memory for the Orkut and Web Trackers graphs, hence the missing bars. However, LIU-PAR is able to finish running on all graphs on 30 threads.

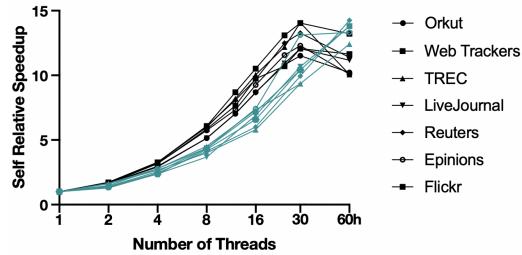


Figure 7.6: This figure compares the parallel self-relative speedups of PAR (in black) and LIU-PAR (in green) over different numbers of threads. LIU-PAR on 60 hyper-threads runs out of memory for the Orkut and Web Trackers graphs. Also, “60h” stands for 60 hyper-threads.

which also reduces the overall memory consumption of PAR compared to LIU-PAR. For Orkut and Web Trackers, the running times reported in Figure 7.5 represent LIU-PAR’s performance on 30 threads, which does not run out of memory on these graphs. Compared to running LIU-PAR on 30 threads, PAR is able to achieve 1.9–4.9x speedups, with an average of 2.7x. Considering the best running times of LIU-PAR for each graph, PAR is between 1.5–4.9x faster than LIU-PAR, with an average of 2.3x.

We outperform LIU-PAR due to our peeling space pruning optimization and our use of Julienne [95]. In particular, LIU-PAR performs more repeated work for cores that are processed by both peeling along increasing α values and peeling along increasing β value, as explained in detail in Section 7.4.3. Additionally, we note that our single-threaded running times achieve between 1.8–10.3x speedups over LIU-SEQ, with an average speedup of 3.9x. This demonstrates the effectiveness of our optimizations, particularly our peeling space pruning optimization, even in the sequential setting.

Analysis of Scalability. Figure 7.6 demonstrates the parallel scalability of our algorithm over different numbers of threads. PAR achieves up to a 14.6x self-relative speedup, with an average of 12.2x across our different input graphs, comparing our running time on 30 threads to our single-threaded running time. In comparison, LIU-PAR, when running on 60 hyper-threads, achieves a similar average self-relative speedup of 13.5x and a maximum of 14.3x. When LIU-PAR is run on 30 threads, it attains an average self-relative speedup of 10.5x, with a maximum of 13.1x.

We note that there is a plateau of the speedup achieved by PAR from 30 threads to 60 hyper-threads due to the strong inherent parallelism of our algorithm; the additional parallelism provided by hyper-threading does not improve its running time.

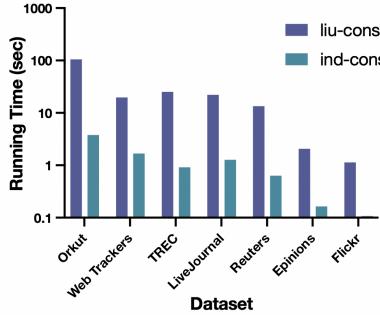


Figure 7.7: This figure shows the running times of Liu et al.’s sequential index construction algorithm, LIU-CONS and our parallel index construction algorithm, IND-CONS.

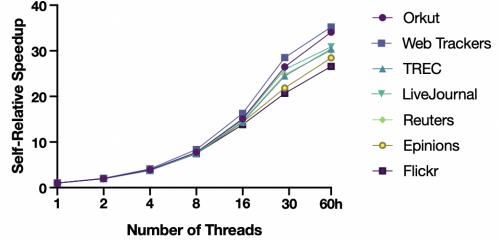


Figure 7.8: This figure shows the parallel self-relative speedup of IND-CONS over different numbers of threads. Note that “60h” stands for 60 hyper-threads.

7.7.4 Bi-core Index

Now, we discuss the performance of the parallel bi-core index construction and query algorithms.

Bi-core Index Construction. As shown in Figure 7.7, our parallel index construction algorithm IND-CONS consistently outperforms the sequential algorithm LIU-CONS across different graphs. Using 30 threads, IND-CONS achieves 10.6–27.7x speedups over LIU-CONS, with an average speedup of 18.4x. We additionally note that constructing the index is not computationally intensive compared to computing the bi-core decomposition, and the running times of IND-CONS constitute 2.2%–8.3% of the running times of PAR.

Figure 7.8 shows the self-relative speedups of IND-CONS across different number of threads and over graphs of different sizes. We observe good scalability, with 20.7–28.5x self-relative speedups of IND-CONS on 30 threads; the average self-relative speedup is 24.7x.

Bi-core Index Query. Our parallel QUERY operation attains between 17.1–116.3x speedups over LIU-QUERY, as demonstrated by Figure 7.9, with an average of 43.8x. Our significant speedups are due to our parallelization and due to our usage of a compact storage format for our index structure, which gives us better cache locality for batches of queries.

Figure 7.10 shows the parallel scalability of QUERY over different numbers of threads. QUERY achieves between 2.6–6.9x self-relative speedup on 30 threads, with an average of 5.3x. Overall, it shows good scalability over most of our input graphs, especially graphs with larger sizes.

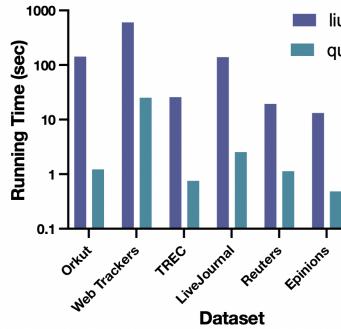


Figure 7.9: This graph shows the running times of Liu et al.’s sequential query algorithm, LIU-QUERY, and our parallel query algorithm, QUERY, on a batch of 10,000 queries.

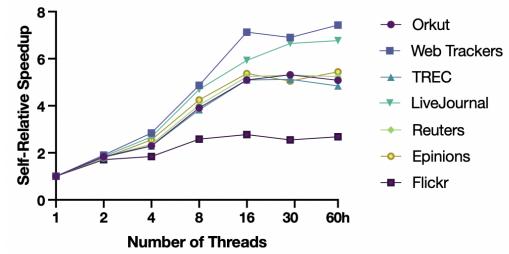


Figure 7.10: This graph shows the parallel self-relative speedups of PAR-QUERY over different numbers of threads, running on a batch of 10,000 queries. Note that “60h” stands for 60 hyper-threads.

7.8 Discussion

In this chapter, we developed a work-efficient shared-memory parallel bi-core decomposition algorithm with improved span bounds. Our parallel algorithm improves the span complexity from the state-of-the-art $O(m)$ to $O(\rho \log n)$ w.h.p. Furthermore, we proved the problem of bi-core decomposition to be P-complete. We also introduced a parallel indexing structure to store the bi-cores, and provided work-efficient parallel index construction and query algorithms. Finally, we introduced a practical optimization reducing the amount of computation in the peeling process, and we provided fast implementations of all of our algorithms. We performed a thorough experimental evaluation of our algorithms on real-world bipartite graphs, and demonstrated that our parallel algorithms outperform the previous best parallel implementation by up to 4.9x for computing the bi-core numbers, and outperform the previous best sequential implementation by up to 27.7x for constructing the index structure and up to 116.3x for querying the bi-cores. We also showed that our bi-core decomposition algorithms are scalable to various real-world graphs with up to hundreds of millions of edges.

Chapter 8

Butterfly Peeling

8.1 Introduction

Butterfly counting naturally lends itself to finding dense subgraph structures in bipartite networks. Zou [355] and Sariyüce and Pinar [282] developed peeling algorithms to hierarchically discover dense subgraphs, similar to the k -core decomposition for unipartite graphs [290, 229]. There has been recent work on designing efficient sequential algorithms for butterfly peeling [355, 282, 329]. Given the high computational requirements of butterfly computations, it is natural to study whether we can obtain performance improvements using parallel machines, and given that all real-world bipartite graphs fit on a multicore machine, we design parallel algorithms for this setting. This chapter presents an extension of the framework for butterfly computations, PARBUTTERFLY, from Chapter 3, that enables us to obtain new parallel algorithms for butterfly peeling. PARBUTTERFLY is a modular framework that enables us to easily experiment with many variations of our algorithms. We not only show that our algorithms are efficient in practice, but also prove strong theoretical bounds on their work and span.

PARBUTTERFLY provides parallel algorithms for peeling bipartite networks based on sequential dense subgraph discovery algorithms developed by Zou [355] and Sariyüce and Pinar [282]. Our peeling algorithms iteratively remove the vertices (tip decomposition) or edges (wing decomposition) with the lowest butterfly count until the graph is empty. Each iteration removes vertices (edges) from the graph in parallel and updates the butterfly counts of neighboring vertices (edges) using the parallel wedge aggregation techniques that we developed for counting. We use a parallel bucketing data structure by Dhulipala *et al.* [95] and a new parallel Fibonacci heap to efficiently maintain the butterfly counts.

We prove theoretical bounds showing that some variants of our peeling algorithms are highly parallel and match the work of the best sequential algorithm. We design a parallel Fibonacci heap that improves upon the work bounds for vertex-peeling from Sariyüce and Pinar’s sequential algorithms, which take work proportional to the maximum number of per-vertex butterflies. For a graph $G(V, E)$ with n vertices and m edges, PARBUTTERFLY gives a vertex-peeling algorithm that takes $O(\min(\max-b_v,$

$\rho_v \log n + \sum_{v \in V} \deg(v)^2$) expected work, $O(\rho_v \log^2 n)$ span w.h.p., and $O(n^2 + \text{max-b}_v)$ additional space, and an edge-peeling algorithm that takes $O(\min(\text{max-b}_e, \rho_e \log n) + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$ expected work, $O(\rho_e \log^2 m)$ span w.h.p., and $O(m + \text{max-b}_e)$ additional space, where max-b_v and max-b_e are the maximum number of per-vertex and per-edge butterflies and ρ_v and ρ_e are the number of vertex and edge peeling iterations required to remove the entire graph. Additionally, given a slightly relaxed work bound, we can improve the space bounds in both algorithms; specifically, we have a vertex-peeling algorithm that takes $O(\rho_v \log n + \sum_{v \in V} \deg(v)^2)$ expected work, $O(\rho_v \log^2 n)$ span w.h.p., and $O(n^2)$ additional space, and we have an edge-peeling algorithm that takes $O(\rho_e \log n + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$ expected work, $O(\rho_e \log^2 m)$ span w.h.p., and $O(m)$ additional space.

We present a comprehensive experimental evaluation of all of the different variants of peeling algorithms in the PARBUTTERFLY framework. On a 48-core machine, our peeling algorithms achieve self-relative speedups of up to 10.7x and due to their improved work complexities, outperform the fastest sequential baseline by up to several orders of magnitude.

In summary, the contributions of this chapter are as follows.

- (1) New parallel algorithms for butterfly peeling.
- (2) A framework PARBUTTERFLY with different ranking and wedge aggregation schemes that can be used for parallel butterfly peeling.
- (3) Strong theoretical bounds on algorithms obtained using PARBUTTERFLY.
- (4) A comprehensive experimental evaluation on a 48-core machine demonstrating high parallel scalability and fast running times compared to the best sequential baselines.

The PARBUTTERFLY code can be found at <https://github.com/jeshi96/parbutterfly>.

8.2 Preliminaries

In this chapter, we take every bipartite graph $G = (U, V, E)$ to be simple and undirected.

A **butterfly** is a set of four vertices $u_1, u_2 \in U$ and $v_1, v_2 \in V$ with edges $(u_1, v_1), (u_1, v_2), (u_2, v_1), (u_2, v_2) \in E$. A **wedge** is a set of three vertices $u_1, u_2 \in U$ and $v \in V$, with edges $(u_1, v), (u_2, v) \in E$. We call the vertices u_1, u_2 **endpoints** and the vertex v the **center**. Symmetrically, a wedge can also consist of vertices $v_1, v_2 \in V$ and $u \in U$, with edges $(v_1, u), (v_2, u) \in E$. We call the vertices v_1, v_2 endpoints and the vertex u the center. We can decompose a butterfly into two wedges that share the same endpoints but have distinct centers.

We store our graphs in compressed sparse row (CSR) format, which requires $O(m + n)$ space. We initially maintain separate offset and edge arrays for each vertex partition U and V , and we assume that all arrays are stored consecutively in memory.

PARBUTTERFLY Framework for Peeling

- (1) *Obtain butterfly counts*: Obtain per-vertex or per-edge butterfly counts from the counting framework.
- (2) *Peel*: Iteratively remove vertices or edges with the lowest butterfly count from the graph until an empty graph is reached.

Figure 8.1: The PARBUTTERFLY framework for peeling.

8.3 PARBUTTERFLY Framework

In this section, we describe the PARBUTTERFLY framework and its components. Section 8.3.1 describes the butterfly peeling procedures. Section 8.4 goes into more detail on the parallel algorithms that can be plugged into the framework, as well as their analysis.

8.3.1 Peeling Framework

Butterfly peeling classifies induced subgraphs by the number of butterflies that they contain. Formally, a vertex induced subgraph is a *k-tip* if it is a maximal induced subgraph such that for a bipartition, every vertex in that bipartition is contained in at least k butterflies and every pair of vertices in that bipartition is connected by a sequence of butterflies. Similarly, an edge induced subgraph is a *k-wing* if it is a maximal induced subgraph such that every edge is contained within at least k butterflies and every pair of edges is connected by a sequence of butterflies.

The *tip number* of a vertex v is the maximum k such that there exists a k -tip containing v , and the *wing number* of an edge (u, v) is the maximum k such that there exists a k -wing containing (u, v) . *Vertex peeling*, or *tip decomposition*, involves finding all tip numbers of vertices in one of the bipartitions, and *edge peeling*, or *wing decomposition*, involves finding all wing numbers of edges.

The sequential algorithms for vertex peeling and edge peeling involve finding butterfly counts and in every round, removing the vertex or edge contained within the fewest number of butterflies, respectively. In parallel, instead of removing a single vertex or edge per round, we remove all vertices or edges that have the minimum number of butterflies.

The peeling framework is shown in Figure 8.1, and supports vertex peeling (tip decomposition) and edge peeling (wing decomposition). Because it also involves iterating over wedges and aggregating wedges by endpoint, it contains similar parameters to those in the counting framework described in Section 3.3.1. However, there are a few key differences between counting and peeling.

First, ranking is irrelevant, because all wedges containing a peeled vertex must be accounted for regardless of order. Also, using atomic add operations to update butterfly counts is not work-efficient with respect to our peeling data structure (see

Section 8.4.1), so we do not have this as an option in our implementation. Finally, vertex or edge peeling can only be performed if the counting framework produces per-vertex or per-edge butterfly counts, respectively.

Thus, the main parameter for the peeling framework is the choice of method for wedge aggregation: sorting, hashing, histogramming, simple batching, or wedge-aware batching. These are precisely the same options described in Section 3.3.1.2.

8.4 PARBUTTERFLY Algorithms

We describe here our parallel algorithms for butterfly peeling in more detail. Note that Sariyüce and Pinar [282] gives a sequential butterfly peeling algorithm, but their algorithm is not work-efficient.

8.4.1 Peeling Algorithms

We describe and analyze our parallel algorithms for butterfly peeling here. Note that this builds directly upon the subroutines for and analysis of butterfly counting, described in Section 3.4.2. In the analysis, we assume that the relevant per-vertex and per-edge butterfly counts are already given by the counting algorithms. The sequential algorithm for butterfly peeling [282] is precisely the sequential algorithm for k -core [290, 229], except that instead of computing and updating the number of neighbors removed from each vertex per round, we compute and update the number of butterflies removed from each vertex or edge per round. Thus, we base our parallel butterfly peeling algorithm on the parallel bucketing-based algorithm for k -core in Julienne [95]. In parallel, our butterfly peeling algorithm removes (peels) all vertices or edges with the minimum butterfly count in each round, and repeats until the entire graph has been peeled.

Zou [355] give a sequential butterfly peeling per edge algorithm that they claim takes $O(m^2)$ work. However, their algorithm repeatedly scans the edge list up to the maximum number of butterflies per edge iterations, so their algorithm actually takes $O(m^2 + m \cdot \text{max-}b_e)$ work, where $\text{max-}b_e$ is the maximum number of butterflies per edge. This is improved by Sariyüce and Pinar's [282] work; Sariyüce and Pinar state that their sequential butterfly peeling algorithms per vertex and per edge take $O(\sum_{u \in U} \deg(u)^2)$ work and $O(\sum_{u \in U} \sum_{v_1, v_2 \in N(u)} \max(\deg(v_1), \deg(v_2)))$ work, respectively. They account for the time to update butterfly counts, but do not discuss how to extract the vertex or edge with the minimum butterfly count per round. In their implementation, their bucketing structure is an array of size equal to the number of butterflies, and they sequentially scan this array to find vertices to peel. They scan through empty buckets, and so the time complexity and additional space for their butterfly peeling implementations is on the order of the maximum number of butterflies per vertex or per edge.

We design a more efficient bucketing structure, which stores non-empty buckets in a Fibonacci heap [137], keyed by the number of butterflies. We have an added $O(\log n)$ factor to extract the bucket containing vertices with the minimum butterfly count.

Algorithm 8.1 – Parallel vertex peeling (tip decomposition)

```

1: procedure GET-V-WEDGES( $G, A, f$ )       $\triangleright f$  is a function used to apply over all
   wedges without storing said wedges; COUNT-V-WEDGES passes  $f$  in when it aggregates
   wedges for butterfly computations
2:   parfor  $u_1 \in A$  do
3:     parfor  $v \in N(u_1)$  where  $v$  has not been previously peeled do
4:       parfor  $u_2 \in N(v)$  where  $u_2 \neq u_1$  and  $u_2$  has not been previously peeled do
5:          $f((u_1, u_2), v)$ 
6: procedure UPDATE-V( $G = (U, V, E), B, A$ )
7:    $B' \leftarrow \text{COUNT-V-WEDGES}(\text{GET-V-WEDGES}(G, A))$ 
8:   Subtract corresponding counts  $B'$  from  $B$ 
9:   return  $B$ 
10: procedure PEEL-V( $G = (U, V, E), B$ )  $\triangleright B$  is an array of butterfly counts per vertex
11:   Let  $K$  be a bucketing structure mapping  $U$  to buckets based on # of butterflies
12:    $f \leftarrow 0$ 
13:   while  $f < |U|$  do
14:      $A \leftarrow$  all vertices in next bucket (to be peeled)
15:      $f \leftarrow f + |A|$ 
16:      $B \leftarrow \text{UPDATE-V}(G, B, A)$                                  $\triangleright$  Update # butterflies
17:     Update the buckets of changed vertices in  $B$ 
18:   return  $K$ 

```

Note that insertion and updating keys in Fibonacci heaps take $O(1)$ amortized time per key, which does not contribute more to our work. To use this in our parallel peeling algorithms, we need to ensure that batch insertions, decrease-keys, and deletions in the Fibonacci are work-efficient and have low span. We present a parallel Fibonacci heap and prove its bounds in Section 8.5. We show that a batch of k insertions takes $O(k)$ amortized expected work and $O(\log n)$ span w.h.p., a batch of k decrease-key operations takes $O(k)$ amortized expected work and $O(\log^2 n)$ span w.h.p., and a parallel delete-min operation takes $O(\log n)$ amortized expected work and $O(\log n)$ span w.h.p.

A standard sequential Fibonacci heap gives work-efficient bounds for sequential butterfly peeling, and our parallel Fibonacci heap gives work-efficient bounds for parallel butterfly peeling. The work of our parallel vertex-peeling algorithm improves over the sequential algorithm of Sariyüce and Pinar [282].

Our actual implementation uses the bucketing structure from Julienne [95], which is not work-efficient in the context of butterfly peeling,¹ but is fast in practice. Julienne materializes only 128 buckets at a time, and when all of the materialized buckets become empty, Julienne will materialize the next 128 buckets. To avoid processing many empty buckets, we use an optimization to skip ahead to the next range of 128 non-empty buckets during materialization.

¹Julienne is work-efficient in the context of k -core.

8.4.1.1 Per Vertex

The parallel vertex peeling (tip decomposition) algorithm is given in PEEL-V (Algorithm 8.1). Note that we peel vertices considering only the bipartition of the graph that produces the fewest number of wedges (considering the vertices in that bipartition as endpoints), which mirrors Sarıyüce and Pinar's [282] sequential algorithm and gives us work-efficient bounds for peeling; more concretely, we consider the bipartition X such that $\sum_{v \in X} \binom{\deg(v)}{2}$ is minimized. Without loss of generality, let U be this bipartition.

Vertex peeling takes as input the per-vertex butterfly counts from the PARBUTTERFLY counting framework. We create a bucketing structure mapping vertices in U to buckets based on their butterfly count (Line 11). While not all vertices have been peeled, we retrieve the bucket containing vertices with the lowest butterfly count (Line 16), peel them from the graph, and compute the wedges removed due to peeling (Line 16). Finally, we update the buckets of the remaining vertices whose butterfly count was affected due to the peeling (Line 17).

The main subroutine in PEEL-V is UPDATE-V (Lines 6–9), which returns a set of vertices whose butterfly counts have changed after peeling a set of vertices. To compute updated butterfly counts, we use the equations in Lemma 3.2 and precisely the same overall steps as in our counting algorithms: wedge retrieval, wedge counting, and butterfly counting. Importantly, in wedge retrieval, for every peeled vertex u_1 , we must gather all wedges with an endpoint u_1 , to account for all butterflies containing u_1 (from Equation (3.1)). We process all peeled vertices u_1 in parallel (Line 2), and for each one we find all vertices u_2 in its 2-hop neighborhood, each of which contributes a wedge (Lines 3–5). Note that there is a subtle point to make here, where we may double-count butterflies if we include wedges containing previously peeled vertices or other vertices in the process of being peeled. We can use a separate array to mark such vertices, and break ties among vertices in the process of being peeled by rank. Then, we ignore these vertices when we iterate over the corresponding wedges in Lines 3 and 4, so these vertices are not included when we compute the butterfly contributions of each vertex.

Finally, we aggregate the number of deleted butterflies per vertex (Line 7), and update the butterfly counts (Line 8). The wedge aggregation and butterfly counting steps are precisely as given in our vertex counting algorithm (Algorithm 3.3).

The work of PEEL-V is dominated by the total work spent in the UPDATE-V subroutine. Since UPDATE-V will eventually process in the subsets A all vertices in U , the total work in wedge retrieval is precisely the number of wedges with endpoints in U , or $O(\sum_{u \in U} \deg(u)^2)$. The work analysis for COUNT-V-WEDGES then follows from a similar analysis as in Section 3.4.2.2. Using our parallel Fibonacci heap, extracting the next bucket on Line 14 takes $O(\log n)$ amortized work and updating the buckets on Line 17 is upper bounded by the number of wedges.

Additionally, the space complexity is bounded above by the space complexity of COUNT-V-WEDGES, which uses a hash table keyed by endpoint pairs. Thus, the space complexity is given by $O(n^2)$.

To analyze the span of PEEL-V, we define ρ_v to be the *vertex peeling complexity*

of the graph, or the number of rounds needed to completely peel the graph where in each round, all vertices with the minimum butterfly count are peeled. Then, since the span of each call of UPDATE-V is bounded by $O(\log m)$ w.h.p. as discussed in Section 3.4.2.2, and since the span of updating buckets is bounded by $O(\log^2 m)$ w.h.p., the overall span of PEEL-V is $O(\rho_v \log^2 m)$ w.h.p.

If the maximum number of per-vertex butterflies is $\Omega(\rho_v \log n)$, which is likely true in practice, then the work of the algorithm described above is faster than Sariyüce and Pinar's [282] sequential algorithm, which takes $O(\max\text{-}b_v + \sum_{u \in U} \deg(u)^2)$ work, where $\max\text{-}b_v$ is the maximum number of butterflies per-vertex.

We must now handle the case where $\max\text{-}b_v$ is $O(\rho_v \log n)$. Note that in order to achieve work-efficiency in this case, we must relax the space complexity to $O(n^2 + \max\text{-}b_v)$, since we use $O(\max\text{-}b_v)$ space to maintain a different bucketing structure. More specifically, while we do not know ρ_v at the beginning of the algorithm, we can start running the algorithm as stated (with the Fibonacci heap), until the number of peeling rounds q is equal to $\max\text{-}b_v / \log n$. If this occurs, then since $q \leq \rho_v$, we have that $\max\text{-}b_v$ is at most $\rho_v \log n$ (if this does not occur, we know that $\max\text{-}b_v$ is greater than $\rho_v \log n$, and we finish the algorithm as described above). Then, we terminate and restart the algorithm using the original bucketing structure of Dhulipala *et al.* [98], which will give an algorithm with $O(\max\text{-}b_v + \sum_{u \in U} \deg(u)^2)$ expected work and $O(\rho_v \log^2 n)$ span w.h.p. The work bound matches the work bound of Sariyüce and Pinar and therefore, our algorithm is work-efficient.

The overall complexity of butterfly vertex peeling is as follows.

Theorem 8.1. *Butterfly vertex peeling can be performed in $O(\min(\max\text{-}b_v, \rho_v \log n) + \sum_{u \in U} \deg(u)^2)$ expected work, $O(\rho_v \log^2 n)$ span w.h.p., and $O(n^2 + \max\text{-}b_v)$ space, where $\max\text{-}b_v$ is the maximum number of per-vertex butterflies ρ_v is the vertex peeling complexity. Alternatively, butterfly vertex peeling can be performed in $O(\rho_v \log n + \sum_{u \in U} \deg(u)^2)$ expected work, $O(\rho_v \log^2 n)$ span w.h.p., and $O(n^2)$ space.*

8.4.1.2 Per Edge

While the bucketing structure for butterfly peeling by edge follows that for butterfly peeling by vertex, the algorithm to update butterfly counts within each round is different. Based on Lemma 3.2, in order to obtain all butterflies containing some edge (u_1, v_1) , we must consider all neighbors $u_2 \in N(v_1) \setminus \{u_1\}$ and then find the intersection $N(u_1) \cap N(u_2)$. Each vertex v_2 in this intersection where $v_2 \neq v_1$ produces a butterfly (u_1, v_1, u_2, v_2) . There is no simple aggregation method using wedges in this scenario; we must find each butterfly individually in order to count contributions from each edge. This is precisely the serial update algorithm that Sariyüce and Pinar [282] use for edge peeling.

The algorithm for parallel edge peeling is given in PEEL-E (Algorithm 8.2). Edge peeling takes as input the per-edge butterfly counts from the PARBUTTERFLY counting framework. Line 13 initializes a bucketing structure mapping each edge to a bucket based on its butterfly count. While not all edges have been peeled, we retrieve the bucket containing vertices with the lowest butterfly count (Line 16), peel them

Algorithm 8.2 – Parallel edge peeling (wing decomposition)

```

1: procedure UPDATE-E( $G = (U, V, E), B, A$ )
2:   Initialize an additive parallel hash table  $B'$  to store updated butterfly counts
3:   parfor  $(u_1, v_1) \in A$  do
4:     parfor  $u_2 \in N(v_1)$  where  $u_2 \neq u_1$  and  $(v_1, u_2)$  has not been previously peeled do
5:        $N \leftarrow \text{INTERSECT}(N(u_1), N(u_2))$ , excepting previously peeled edges
6:       Insert  $((u_2, v_1), |N| - 1)$  in  $B'$ 
7:       parfor  $v_2 \in N$  where  $v_2 \neq v_1$  do
8:         Insert  $((u_1, v_2), 1)$  in  $B'$ 
9:         Insert  $((u_2, v_2), 1)$  in  $B'$ 
10:      Subtract corresponding counts in  $B'$  from  $B$ 
11:      return  $B$ 

12: procedure PEEL-E( $G = (U, V, E), B$ )     $\triangleright B$  is an array of butterfly counts per edge
13:   Let  $K$  be a bucketing structure mapping  $E$  to buckets based on # of butterflies
14:    $f \leftarrow 0$ 
15:   while  $f < m$  do
16:      $A \leftarrow$  all edges in next bucket (to be peeled)
17:      $f \leftarrow f + |A|$ 
18:      $B \leftarrow \text{UPDATE-E}(G, B, A)$                                  $\triangleright$  Update # butterflies
19:     Update the buckets of changed edges in  $B$ 
20:   return  $K$ 

```

from the graph and compute the wedges that were removed due to peeling (Line 18). Finally, we update the buckets of the remaining vertices whose butterfly count was affected due to the peeling (Line 19).

The main subroutine is UPDATE-E (Lines 1–11), which returns a set of edges whose butterfly counts have changed after peeling a set of edges. For each peeled edge (u_1, v_1) in parallel (Line 3), we find all neighbors u_2 of v_1 where $u_2 \neq u_1$ and compute the intersection of the neighborhoods of u_1 and u_2 (Lines 4–5). All vertices $v_2 \neq v_1$ in their intersection contribute a deleted wedge, and we indicate the number of deleted wedges on the remaining edges of the butterfly (u_2, v_1) , (u_1, v_2) , and (u_2, v_2) in an array B' (Lines 6–9). As in per-vertex peeling, we can avoid double-counting butterflies corresponding to previously peeled edges and edges in the process of being peeled using an additional array to mark these edges; we ignore these edges when we iterate over neighbors in Line 4 and perform the intersections in Line 5, so they are not included when we compute the butterfly contributions of each edge. Finally, we update the butterfly counts (Line 10).

The work of PEEL-E is again dominated by the total work spent in the UPDATE-E subroutine. We can optimize the intersection on Line 5 by using hash tables to store the adjacency lists of the vertices, so we only perform $O(\min(\deg(u), \deg(u')))$ work when intersecting $N(u)$ and $N(u')$ (by scanning through the smaller list in parallel and performing lookups in the larger list). This gives us $O(\sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$ expected work.

Additionally, the space complexity is bounded by storing the updated butterfly

counts; note that we do not store wedges or aggregate counts per endpoints. Thus, the space complexity is given by $O(m)$.

As in vertex peeling, to analyze the span of PEEL-E, we define ρ_e to be the *edge peeling complexity* of the graph, or the number of rounds needed to completely peel the graph where in each round, all edges with the minimum butterfly count are peeled. The span of UPDATE-E is bounded by the span of updating buckets, giving us $O(\log^2 m)$ span w.h.p. Thus, the overall span of PEEL-E is $O(\rho_e \log^2 m)$ w.h.p.

Similar to vertex peeling, if the maximum number of per-edge butterflies is $\Omega(\rho_e \log m)$, which is likely true in practice, then the work of our algorithm is faster than the sequential algorithm by Sariyüce and Pinar [282]. The work of their algorithm is $O(\max\text{-}b_e + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$, where $\max\text{-}b_e$ is the maximum number of butterflies per-edge (assuming that their intersection is optimized).

To deal with the case where the maximum number of butterflies per-edge is small, in order to achieve work-efficiency in this case, we must relax the space complexity to $O(m + \max\text{-}b_e)$, since we use $O(\max\text{-}b_e)$ space to maintain a different bucketing structure. More specifically, we can start running the algorithm as stated (with the Fibonacci heap), until the number of peeling rounds q is equal to $\max\text{-}b_e / \log m$. If this occurs, then since $q \leq \rho_e$, we have that $\max\text{-}b_e$ is at most $\rho_e \log m$ (if this does not occur, we know that $\max\text{-}b_e$ is greater than $\rho_e \log m$, and we finish the algorithm as described above). Then, we terminate and restart the algorithm using the original bucketing structure of Dhulipala *et al.* [98], which will give an algorithm with $O(\max\text{-}b_e + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$ expected work and $O(\rho_e \log^2 m)$ span w.h.p. Our work bound matches the work bound of Sariyüce and Pinar and therefore, our algorithm is work-efficient.

The overall complexity of butterfly edge peeling is as follows.

Theorem 8.2. *Butterfly edge peeling can be performed in $O(\min(\max\text{-}b_e, \rho_e \log m) + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$ expected work, $O(\rho_e \log^2 m)$ span w.h.p., and $O(m + \max\text{-}b_e)$ space, where $\max\text{-}b_e$ is the maximum number of per-edge butterflies and ρ_e is the edge peeling complexity. Alternatively, butterfly edge peeling can be performed in $O(\rho_e \log m + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')))$ expected work, $O(\rho_e \log^2 m)$ span w.h.p., and $O(m)$ space.*

8.4.1.3 Per Vertex/Edge Storing All Wedges

We note that if we store all the wedges obtained while counting in the peeling algorithm, we can obtain alternate parallel vertex-peeling and edge-peeling algorithms that take $O(\rho_v \log n + b)$ expected work, $O(\rho_v \log^2 n)$ span w.h.p., and $O(\alpha m)$ space, where b is the total number of butterflies (or, relaxing the space bounds, $O(b)$ expected work, $O(\rho_v \log n)$ span w.h.p., and $O(\alpha m + \max\text{-}b)$ space where $\max\text{-}b$ refers to the maximum number of per-vertex and per-edge butterflies respectively). The parallel edge-peeling algorithm uses similar ideas as the sequential algorithm given by Wang *et al.* [329], which takes $O(b)$ work and $O(\alpha m)$ additional space. We have omitted these algorithms from this chapter.

8.5 Parallel Fibonacci Heap

Fibonacci heaps were first introduced by Fredman and Tarjan [137]. In this section, we show that we can parallelize batches of insertion and decrease-key operations work-efficiently, with logarithmic span. Also, we show that a single work-efficient parallel delete-min can be performed with logarithmic span, which is sufficient for our purposes.

Previous work has also explored parallelism in Fibonacci heaps. Driscoll et al. [112] present relaxed heaps, which achieve the same bounds as Fibonacci heaps, but can be used to obtain a parallel implementation of Dijkstra’s algorithm; however their data structures do not support batch-parallel insertions or decrease-key operations. Huang and Weihl [170] and Bhattacharai [43] present implementations of parallel Fibonacci heaps by relaxing the semantics of delete-min, although no theoretical bounds are given.

A *Fibonacci heap* H consists of heap-ordered trees (maintained using a root list, which is a doubly linked list), with certain nodes marked and a pointer to the minimum element. Each node in H is a key-value pair. Let n denote the number of elements in our Fibonacci heap. The **rank** of a node x is the number of children that x contains, and the **rank** of a heap is the maximum rank of any node in the heap. Note that the rank of a Fibonacci heap is bounded by $O(\log n)$. We also define $t(H)$ to be the number of trees in H , and we define $m(H)$ to be the number of marked nodes in H .

We begin by giving a brief overview of the sequential Fibonacci heap operations:

- **Insert** ($O(1)$ work): To insert node x , we add x to the root list as a new singleton tree and update the minimum pointer if needed.
- **Delete-min** ($O(\log n)$ amortized work): We delete the minimum node as given by the minimum pointer, and add all of its children to the root list. We update the minimum pointer if needed. We then merge trees until no two trees have the same rank; a merge occurs by taking two trees of the same rank, and assigning the larger root as a child of the smaller root.
- **Decrease-key** ($O(1)$ amortized work): To decrease the key of a node x , we first check if decreasing the key would violate heap order. If not, we simply decrease the key. Otherwise, we cut the node x and its subtree from its parent, and add it to the root list. If the parent of x was unmarked, we mark the parent. Otherwise, we cut the parent, add it to the root list, and unmark it; we recurse in the same manner on its parent. Finally, we update the minimum pointer if needed

The potential function for the amortized analysis is $\Phi(H) = t(H) + 2 \cdot m(H)$.

In our parallel Fibonacci heap, instead of keeping marks on nodes as boolean values, each node stores an integer number of marks that it accumulates. Furthermore, instead of maintaining the root list as a doubly-linked list, we use a parallel hash table [148] so that we can retrieve all roots efficiently in parallel. Our parallel Fibonacci heap requires linear space to store, just as in the sequential version.

Algorithm 8.3 – Parallel delete-min

```
1: procedure PAR-DELETE-MIN( $H$ )
2:   Delete the minimum node and add all children to root list
3:   Initialize  $C$  such that  $C[i]$  contains all roots with rank  $i$ 
4:   while  $\exists$  a group in  $C$  with  $> 1$  root do
5:     Initialize  $C'$  to hold updated trees
6:     parfor  $i \leftarrow 0$  to  $|C|$  do
7:       Partition the roots in  $C[i]$  into pairs
8:       If a root is leftover, insert it into  $C'[i]$ 
9:       Merge the trees in every pair and insert the new roots into  $C'[i + 1]$ 
10:       $C \leftarrow C'$ 
11:      Use prefix sum among the root nodes to update the minimum pointer
12:   return  $H$ 
```

8.5.1 Batch Insertion

For parallel batch insertion, let K denote the set of key-value pairs that we are adding to our heap. Let $k = |K|$, and n be the size of our heap before insertion. For each key in K , we create a singleton tree. We resize our parallel hash table if necessary to make space for the new singleton trees, and then add all new singleton trees to the root list. Finally, we update the minimum pointer.

Creating new singleton trees takes $O(k)$ work and constant span. Resizing the hash table takes $O(k)$ amortized expected work and $O(\log(n + k))$ span w.h.p. To update the minimum pointer, we use a prefix sum between the newly added nodes and the previous minimum of the heap, which takes $O(k)$ work and $O(\log k)$ span.

The total complexity of batch insertion is given as follows.

Lemma 8.1. *Parallel batch insertion of k elements into a Fibonacci heap with n elements takes $O(k)$ amortized expected work and $O(\log(n + k))$ span w.h.p.*

8.5.2 Delete-min

The amortized work of delete-min is $O(\log n)$, but we describe how to parallelize delete-min in order to get a high probability bound for the span. The parallel delete-min algorithm is given in Algorithm 8.3. We first delete the minimum node and add all of its children to the root list in parallel (Line 2). Then, the main component of our parallel delete-min operation involves consolidating trees such that no two trees share the same rank (Lines 3–10). We place each tree into a group based on its rank (Line 3), and then merge pairs of trees with the same rank in every round (Lines 7–9) until there are no longer trees with the same rank. The final step in our algorithm is updating the minimum pointer using a prefix sum (Line 11).

After $O(\log n)$ rounds, each group will necessarily contain at most one tree, which can be shown inductively. If we assume that after round i , all groups $\leq i$ each contain at most one tree, we see that when we process round $i + 1$, no merged tree can be added to group $\leq i + 1$ (since the rank of merged trees can only increase). Moreover,

Algorithm 8.4 – Parallel batch decrease-key

```

1: procedure BATCH-DECREASE-KEY( $H, K$ )
2:    $\triangleright K$  is an array of triples, holding the key-value pair to be decreased and the
      updated key
3:   Let  $M$  be an empty array (to later store marked nodes)
4:   parfor  $(k, \_, k') \in K$  do
5:     if changing the key to  $k'$  violates heap order then
6:       Cut  $k$  and add to root list with key  $k'$ 
7:       Add a mark to the original parent of  $k$  and add the parent to  $M$ 
8:     else
9:       Change the key to  $k'$ 
10:     $M \leftarrow$  nodes in  $M$  with  $> 1$  marks
11:    while  $M$  is nonempty do
12:      Let  $M'$  be an empty array (to later store marked nodes)
13:      parfor  $p \in M$  do
14:        Cut  $p$  and add to root list
15:        Set # marks on  $p$  to 0 if # marks on  $p$  is even, and 1 otherwise
16:        Add a mark to  $p$ 's original parent and add the parent to  $M'$ 
17:       $M \leftarrow$  nodes in  $M'$  with  $> 1$  marks
18:    return  $H$ 

```

group $i + 1$ contains at most one leftover root, and all other roots have been merged and inserted into group $i + 2$. Thus, the number of rounds needed to complete our consolidation step is bounded above by the rank of H , which is $O(\log n)$.

We use dynamic arrays to represent each group, which allow the insertion of x elements in $O(x)$ amortized work and $O(1)$ span. The amortized work of our parallel delete-min operation is asymptotically equal to the amortized work of the sequential delete-min operation. Their actual costs are the same, except for the hash table and dynamic array operations in the parallel version. Our actual cost for merging trees and dynamic array operations is $O(t(H))$. If we let H' represent our heap after performing PAR-DELETE-MIN, the change in potential is $\Delta\Phi \leq t(H') - t(H) \leq \text{rank}(H') + 1 - t(H) = O(\log n - t(H))$, because no two trees have the same rank after performing our consolidations. Thus, the amortized work for merging trees and dynamic array operations is $O(\log n)$. The amortized expected work for adding the children of the minimum node to the root list is $O(\log n)$ due to hash table insertions. Updating the minimum pointer also takes $O(\log n)$ work, and thus the total amortized expected work is $O(\log n)$, as desired.

The span of our algorithm is dominated by the span of the while loop. In particular, note that every iteration of our while loop has $O(1)$ span, because we can perform the pairwise merges fully in parallel. As we previously discussed, we have at most $O(\log n)$ iterations of our while loop. Inserting the children of the minimum node to the hash table representing the root list takes $O(\log n)$ span w.h.p. Therefore, the span of parallel delete-min is $O(\log n)$ w.h.p. The total complexity of parallel delete-min is as follows.

Lemma 8.2. *Parallel delete-min for a Fibonacci heap takes $O(\log n)$ amortized expected work and $O(\log n)$ span w.h.p.*

8.5.3 Batch Decrease-key

The parallel batch decrease-key operation is given in BATCH-DECREASE-KEY (Algorithm 8.4).

For each decrease key, we check if these decreases violate heap order (Line 5). If not, we can directly decrease the key (Line 9). Otherwise, we cut these nodes from their trees and mark their parents (Lines 6–7). Then, we recursively cut all parents that have been marked more than once, mark their parents, and repeat (Lines 10–17).

We can maintain the arrays M and M' using a parallel filter in work proportional to the size of our batch and the total number of cuts and $O(\log n)$ span per iteration of the while-loop on Line 11.

On Lines 7 and 16, we record in an array when we would like to mark a parent. Then, we can semisort the array and use prefix sum to obtain the number of marks to be added to each parent. This maintains our work bounds, and has $O(\log n)$ w.h.p. per iteration of the while-loop on Line 11.

We now focus on the amortized work analysis. Let k denote the number of keys in K and let c be the total number of cuts that we perform in this algorithm. Note that decreasing our keys takes $O(k)$ total work, and the rest of the work is given by the total number of cuts, or $O(c)$.

Recall that our potential function is $\Phi(H) = t(H) + 2 \cdot m(H)$. Let H' represent our heap after performing BATCH-DECREASE-KEY. The change in the number of trees is given by $t(H') - t(H) = c$, since every new cut produces a new tree.

The change in the number of marks is $m(H') - m(H) \leq k - (c - k) = 2k - c$. The argument for this is similar to the sequential argument.

For each parent node p that is cut, we arbitrarily set a key in K as having *propagated* the cut as follows. Let $c(p)$ denote the key that propagated the cut to parent p , and let $M(p)$ denote the set of all nodes that marked p in the round immediately before p was cut. Then, we set $c(x) = x$ for each key x in K , and we set $c(p)$ to be an arbitrary key in $C(p) = \{c(p') \mid p' \in M(p)\}$; in other words, $c(p)$ is one of the keys that propagated a node that marked p in the round immediately before p was cut.

Each key x then has a well-defined *propagation path*, which is the maximal path of nodes that x has propagated. The last node ℓ of the propagation path must either be a root node or a node whose parent x has not propagated the cut to. Note that x may mark ℓ 's parent without cutting this parent from its tree; we call this mark an *allowance*. In this sense, each key x in K has one mark in its allowance. We have k marks in the total allowance of our heap.

It remains, then, to count the change in the number of marks on each propagation path. We claim that if we have already counted the k allowances in the change in the number of marks ($m(H') - m(H)$), we can now subtract a mark for each cut parent on a propagation path. There are two cases.

If a cut parent p at the start of our algorithm already contained a mark, then the node that propagated the cut added a mark, canceling out the previous mark. Thus,

$c(p)$ has effectively subtracted a mark from p .

If a cut parent p had no marks at the start of our algorithm, a child node p' such that $c(p') \neq c(p)$ must have marked p within our algorithm. Necessarily, $c(p')$ must have ended its propagation path at p' , so it charged its mark allowance to p . Then, when the node that propagated the cut, $c(p)$, added a mark, this cancels out the mark that p' made. Since we have already counted the mark that p' made in its allowance, we can subtract a mark to account for the cancellation.

In total, we see that we can subtract a mark for each cut parent on a propagation path. The number of cut parents is at least $c - k$, so we have $m(H') - m(H) \leq k - (c - k) = 2k - c$, as desired.

As such, the change in potential is $\Phi(H') - \Phi(H) \leq c + 2(2k - c) = 4k - c$. The actual work of BATCH-DECREASE-KEY is $O(k + c)$, and the amortized work is $O(k + c) + 4k - c = O(k)$ by scaling up the units of potential appropriately.

The span of our algorithm is again dominated by the span of the while loop. We have at most $O(\log n)$ iterations of the while loop, since it is bounded by the maximum tree height of our heap, which is bounded by the rank of the heap. Each iteration of the while loop has span $O(\log n)$ w.h.p. Thus, the span of our algorithm is $O(\log^2 n)$ w.h.p.

The overall complexity of parallel batch decrease-key is as follows.

Lemma 8.3. *Parallel batch decrease-key of k elements in a Fibonacci heap with n elements takes $O(k)$ amortized work and $O(\log^2 n)$ span w.h.p.*

8.5.4 Application to Bucketing

We now discuss more specifically how to apply our batch-parallel Fibonacci heap to bucketing in butterfly peeling. Each bucket is represented as a node in the Fibonacci heap, where the key is the number of butterflies and the value is a parallel hash table [148] containing vertices/edges that contain exactly the given number of butterflies. Updates to the bucketing structure trigger certain operations on the Fibonacci heap.

Throughout the rest of this subsection, we also consider key-value pairs in the context of bucketing operations. Bucketing operations involve processing key-value pairs, where we take the key to be a given number of butterflies and the value to be a single vertex/edge. Notably, the value differs from that in the context of a Fibonacci heap; bucketing operations update single vertices/edges, and we use the heap to move sets of these vertices/edges to their correct buckets. To distinguish the key-value pairs in bucketing operations from the key-value pairs that represent nodes in the Fibonacci heap, we refer to the latter as *heap key-value pairs*.

Moreover, in order to ensure that all vertices/edges with the same key are aggregated into a single bucket, we also need a supplemental parallel hash table [148] that stores pointers to buckets, keyed by the corresponding number of butterflies. The combined parallel hash table T and batch-parallel Fibonacci heap H form our bucketing structure $B = (T, H)$. This following analysis assumes n is the number of vertices or edges in the graph, which upper bounds the size of the Fibonacci heap at

Algorithm 8.5 – Bucketing Update Algorithm

```
1: procedure BUCKETING-UPDATE( $B = (T, H), K$ )
2:    $\triangleright K$  is an array of triples that hold key-value pairs to be decreased and their
      updated key
3:   Let  $n_k$  be # times  $k$  appears in a key-value pair in  $K$ 
4:    $I = \{\}$   $\triangleright$  Array of key-value pairs to re-insert
5:    $K' = \{\}$   $\triangleright$  Updated  $K$ 
6:   parfor  $(k, v, k') \in K$  do
7:     if  $n_k = \text{size of } k\text{'s bucket}$  and  $v$  is the first element in the bucket then
8:       Add  $(k, \{v\}, k')$  to  $K'$ 
9:     else
10:      Add  $(k', v)$  to  $I$  and remove  $v$  from bucket  $k$ 
11:   parfor  $(k, \_, k') \in K'$  do
12:     Remove  $k$  from  $T$  and insert  $k'$  into  $T$ 
13:   BATCH-DECREASE-KEY( $H, K'$ )
14:    $I' = \{\}$   $\triangleright$  Array of heap key-value pairs to insert into  $H$ 
15:   parfor each unique  $k'$  where  $(k', v) \in I$  do
16:     if  $k' \in T$  then
17:       Add  $\{v \mid (k', v) \in I\}$  to the hash table of bucket  $k'$ 
18:     else
19:       Add  $(k', V)$  to  $I'$  where  $V$  contains all  $v$  where  $(k', v) \in I$ 
20:   parfor  $(k', V) \in I'$  do
21:     Add  $k'$  to  $T$ 
22:   BATCH-INSERT( $H, I'$ )
23:   return  $B$ 
```

any time. The work bounds for our operations are amortized, but we can remove the amortization when summing across all rounds of our peeling algorithms.

8.5.4.1 Retrieving the Minimum Bucket

To retrieve the minimum bucket, we perform a delete-min operation on the Fibonacci heap, and given the heap key of the minimum, we remove the corresponding key in the supplemental hash table T . The delete-min operation on the Fibonacci heap dominates the complexity, and so retrieving the minimum bucket takes $O(\log n)$ amortized expected work and $O(\log n)$ span w.h.p.

8.5.4.2 Updating the Bucketing Structure

Updating the bucketing structure involves moving elements to new buckets based on their updated butterfly counts, which can only decrease. This involves moving elements between the hash tables of the buckets in the Fibonacci heap, decreasing the key of some buckets in the Fibonacci heap, and inserting new buckets into the Fibonacci heap. The algorithm is shown in Algorithm 8.5.

We must first check whether a key-value pair triggers a decrease-key operation in the Fibonacci heap. If not all values in the bucket need to be updated, then those

values can simply be removed from the bucket and re-inserted with the updated heap key; the bucket holding the rest of the values can remain with the original heap key. Otherwise, if all values in the bucket need to be updated, we keep only the first element in the bucket and decrease the heap key for that element (Lines 7–8 and 13), and we keep the other values with their updated keys in an array I to be re-inserted (Lines 9–10). We also also update the supplemental hash table T for the buckets that we decrease the key for (Line 11–12). We can determine if all values in a bucket need to be updated by using a semisort to aggregate counts on the number of times each key appears in K . This takes $O(k)$ expected work and $O(\log n)$ span w.h.p. where $k = |K|$.

For all key-value pairs that must be reinserted, we first check for each distinct key whether it appears in our supplemental hash table T . For the heap keys that appear in T (Lines 16–17), we simply add the corresponding set of values to their existing bucket in the heap as a batch (which is stored as a hash table, so this consists of performing insertions to the hash table corresponding to the bucket). For heap keys that do not appear in the supplemental hash table T , we keep them along with their heap values (a hash table containing the set of vertices/edges associated with that heap key) in an array I' (Lines 18–19). We also add these new heap keys to T (Lines 20–21). Then we perform a batch insertion in the Fibonacci heap with the set of heap key-value pairs in I' (Line 22). We can perform the hash table operations and insertions into the Fibonacci heap in $O(k)$ amortized expected work and $O(\log n)$ span w.h.p.

The batch decrease-key operation on the Fibonacci heap dominates the complexity, and so updating the bucketing structure for k elements takes $O(k)$ amortized expected work and $O(\log^2 n)$ span w.h.p.

8.6 Experiments

8.6.1 Environment

We run our experiments on an m5d.24xlarge AWS EC2 instance, which consists of 48 cores (with two-way hyper-threading), with 3.1 GHz Intel Xeon Platinum 8175 processors and 384 GiB of main memory. We use Cilk Plus’s work-stealing scheduler [53, 206] and we compile our programs with g++ (version 7.3.1) using the `-O3` flag. We test our algorithms on a variety of real-world bipartite graphs from the Koblenz Network Collection (KONECT) [199]. We remove self-loops and duplicate edges from the graph. Table 8.1 describes the properties of these graphs, including sizes, number of butterflies, and peeling complexities.

We compare our algorithms against Sariyüce and Pinar’s [282] work, which is the state-of-the-art sequential butterfly peeling implementation.

Notationally, when discussing wedge and butterfly aggregation methods, we use the prefix “A” to refer to using atomic adds for butterfly aggregation, and we take a lack of prefix to mean that the wedge aggregation method was used for butterfly aggregation. “BatchS” is the simple version of batching and “BatchWA” is the wedge-

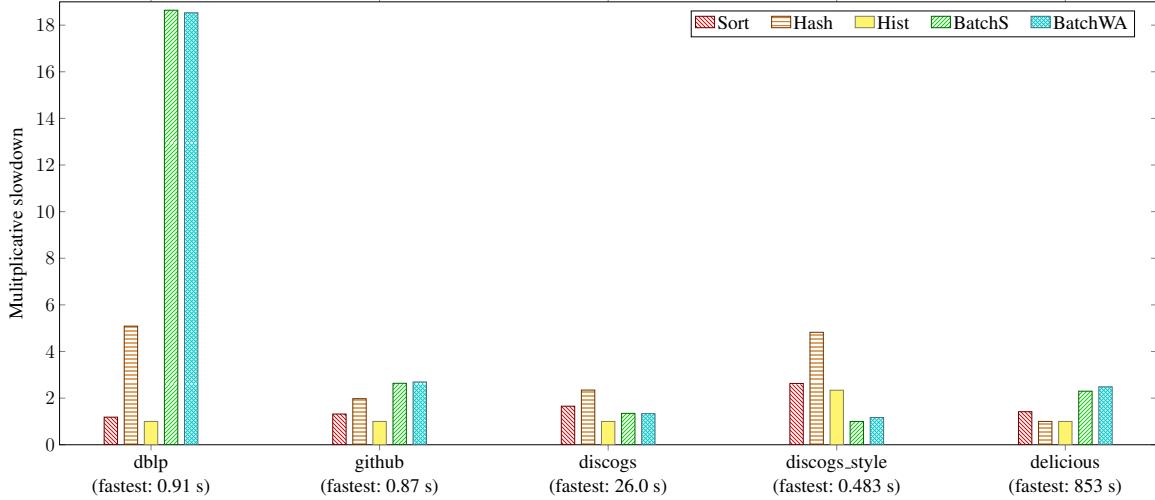


Figure 8.2: These are the parallel runtimes for butterfly vertex peeling with different wedge aggregation methods (these runtimes do not include the time taken to count butterflies). All times are scaled by the fastest parallel time, as indicated in parentheses. Also, note that the runtimes for discogs_style represent single-threaded runtimes; this is because we did not see any parallel speedups for discogs_style, due to the small number of vertices that were peeled.

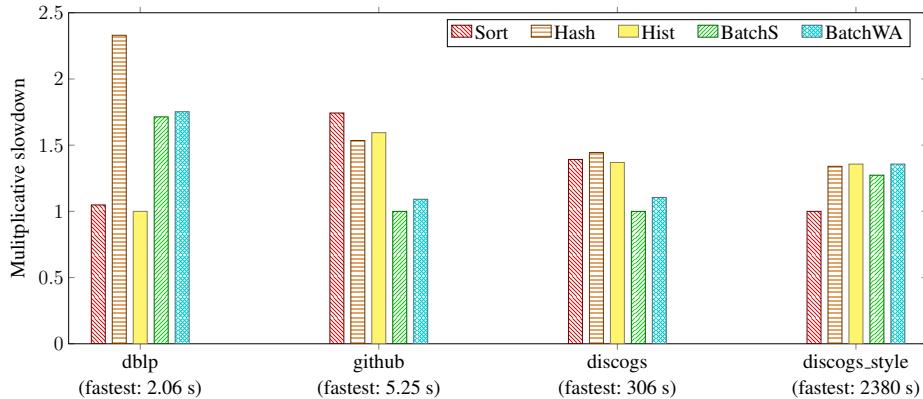


Figure 8.3: These are the parallel runtimes for butterfly edge peeling with different wedge aggregation methods (these runtimes do not include the time taken to count butterflies). All times are scaled by the fastest parallel time, as indicated in parentheses.

aware version of batching that dynamically assigns tasks to workers so they have a roughly equal number of wedges to process.

8.6.1.1 Butterfly Peeling

Figures 8.2 and 8.3 show the runtimes over different wedge aggregation methods for vertex peeling and edge peeling, respectively (the runtimes do not include the time for counting butterflies). We only report times for the datasets for which finished within

| Dataset | Abbreviation | $ U $ | $ V $ | $ E $ | # butterflies | ρ_v | ρ_e |
|----------------------|---------------|------------|------------|-------------|--------------------|----------|----------|
| DBLP | dblp | 4,000,150 | 1,425,813 | 8,649,016 | 21,040,464 | 4,806 | 1,853 |
| Github | github | 120,867 | 56,519 | 440,237 | 50,894,505 | 3,541 | 14,061 |
| Wikipedia edits (it) | itwiki | 2,225,180 | 137,693 | 12,644,802 | 298,492,670,057 | — | — |
| Discogs label-style | discogs | 270,771 | 1,754,823 | 5,302,276 | 3,261,758,502 | 10,676 | 123,859 |
| Discogs artist-style | discogs_style | 383 | 1,617,943 | 5,740,842 | 77,383,418,076 | 374 | 602,142 |
| LiveJournal | livejournal | 7,489,073 | 3,201,203 | 112,307,385 | 3,297,158,439,527 | — | — |
| Wikipedia edits (en) | enwiki | 21,416,395 | 3,819,691 | 122,075,170 | 2,036,443,879,822 | — | — |
| Delicious user-item | delicious | 33,778,221 | 833,081 | 101,798,957 | 56,892,252,403 | 165,850 | — |
| Orkut | orkut | 8,730,857 | 2,783,196 | 327,037,487 | 22,131,701,213,295 | — | — |
| Web trackers | web | 27,665,730 | 12,756,244 | 140,613,762 | 20,067,567,209,850 | — | — |

Table 8.1: These are relevant statistics for the KONECT [199] graphs that we experimented on. Note that we only tested peeling algorithms on graphs for which Sarıyüce and Pinar’s [282] serial peeling algorithms completed in less than 5.5 hours. As such, there are certain graphs for which we have no available ρ_v and ρ_e data, and these entries are represented by a dash.

| Dataset | Vertex Peeling | | | Edge Peeling | | |
|---------------|----------------|--------|--------------------------|--------------|--------|--------------------------|
| | PB | | Sariyüce and Pinar [282] | PB | | Sariyüce and Pinar [282] |
| | T_{48h} | T_1 | T_1 | T_{48h} | T_1 | T_1 |
| dblp | 0.91° | 2.40° | 2.06 | 2.06° | 16.90° | 6.93 |
| github | 0.87° | 1.03° | 1.15 | 5.25* | 18.00* | 18.82 |
| discogs | 26° | 53.10* | 157.14 | 306* | 2160* | 2149.54 |
| discogs_style | 0.48* | 0.48* | 14826.16 | 2380# | 15600* | 16449.56 |
| delicious | 853° | 1900* | 2184.27 | — | — | — |

Table 8.2: These are runtimes in seconds for parallel and single-threaded butterfly peeling from PARBUTTERFLY (PB) and serial butterfly peeling from Sariyüce and Pinar [282]. Note that these runtimes do not include the time taken to count butterflies. For the runtimes from PARBUTTERFLY, we have noted the aggregation method used; * refers to simple batching, # refers to sorting and ° refers to histogramming.

5.5 hours. We find that for vertex peeling, aggregation by histogramming largely gives the best runtimes, while for edge peeling, all of our aggregation methods give similar results.

We compare our parallel peeling times to our single-threaded peeling times and serial peeling times from Sariyüce and Pinar’s [282] implementation, which we ran in our environment and which are shown in Table 8.2. Compared to Sariyüce and Pinar [282], we achieve speedups between 1.3–30696x for vertex peeling and between 3.4–7.0x for edge peeling. Our speedups are highly variable because they depend heavily on the peeling complexities and the number of empty buckets processed. Our largest speedup of 30696x occurs for vertex peeling on discogs_style where we are able to efficiently skip over many empty buckets, while the implementation of Sariyüce and Pinar sequentially iterates over the empty buckets.

Moreover, comparing our parallel peeling times to their corresponding single-threaded times, we achieve speedups between 1.0–10.7x for vertex peeling and between 2.3–10.4x for edge peeling. We did not see self-relative parallel speedups for vertex peeling on discogs_style, because the total number of vertices peeled (383) was too small.

8.7 Related Work

There have been several sequential algorithms designed for butterfly peeling. Zou [355] develop the first algorithm for butterfly peeling per edge, with $O(m^2 + m \cdot \text{max-b}_e)$ work. Sariyüce and Pinar [282] give algorithms for butterfly peeling over vertices and over edges, which take $O(\text{max-b}_v + \sum_{u \in U} \deg(u)^2)$ work and $O(\text{max-b}_e + \sum_{u \in U} \sum_{v_1, v_2 \in N(u)} \max(\deg(v_1), \deg(v_2)))$ work, respectively. Very recently, Wang *et al.* [329] present a sequential algorithm for butterfly edge peeling that improves over the algorithm by Sariyüce and Pinar [282] in practice, and uses an index that takes

$O(\alpha m)$ space.

8.8 Discussion

We have designed in this chapter a framework PARBUTTERFLY that provides efficient parallel algorithms for butterfly peeling (by vertex and by edge). We have also shown strong theoretical bounds in terms of work and span for these algorithms. The PARBUTTERFLY framework is built with modular components that can be combined for practical efficiency. PARBUTTERFLY outperforms the fastest sequential baseline by up to several orders of magnitude for butterfly peeling.

Chapter 9

k -clique Densest Subgraph

9.1 Introduction

In this chapter, we build upon the k -clique counting algorithms given in Chapter 5, and introduce new parallel algorithms for the k -clique densest subgraph problem, a generalization of the densest subgraph problem that was first introduced by Tsourakakis [317]. This problem admits a natural $1/k$ -approximation by peeling vertices in order of their incident k -clique counts. We present a work-efficient parallel peeling algorithm for this problem that peels all vertices with the lowest k -clique count in each round. The span of the algorithm is proportional to the number of peeling rounds needed, which can be more than polylogarithmic, but we also prove the P-completeness of this peeling process for $k > 2$, which indicates that it is unlikely admit a polylogarithmic-span solution. Tsourakakis also shows that naturally extending the parallel algorithm by Bahmani et al. [26] algorithm for approximate densest subgraph gives an $1/(k(1 + \epsilon))$ -approximation in $O(\log n)$ parallel rounds, although he does not describe a work-efficient algorithm. We present a linear-work and polylogarithmic-span algorithm for obtaining a $1/(k(1 + \epsilon))$ -approximation to the k -clique densest subgraph problem. Danisch et al. [86] use their k -clique counting algorithm as a subroutine to implement these two approximation algorithms for k -clique densest subgraph, although their implementations do not have provably-efficient bounds.

We perform a thorough experimental study of these algorithms on a 60-core machine with two-way hyper-threading and compare them with prior work. We study our two parallel approximation algorithms for k -clique densest subgraph and show that our we are able to outperform KCLIST by up to 29.59x and achieve 1.19–13.76x self-relative speedup.

We summarize the main contributions of this chapter below:

- (1) A work-efficient parallel algorithm for computing a $1/k$ -approximation to the k -clique densest subgraph problem, and a work-efficient polylogarithmic-span algorithm for computing a $1/(k(1 + \epsilon))$ -approximation.
- (2) A proof that the algorithm for computing a $1/k$ -approximation to the k -clique densest subgraph problem is P-complete for $k > 2$.
- (3) Highly-optimized implementations of our algorithms that achieve significant speedups

over existing state-of-the-art methods.

Our code is publicly available at: <https://github.com/ParAlg/gbbs/tree/master/benchmarks/CliqueCounting>.

9.2 Related Work

Vertex Peeling and k -clique Densest Subgraph. An important application of k -clique counting is its use as subroutine in computing generalizations of approximate densest subgraph. A seminal work of Charikar gave a polynomial time algorithm for exactly computing the densest subgraph, and also presented a simple $1/2$ -approximation algorithm [66]. The algorithm works by iteratively removing (peeling) the vertex with minimum degree, and one of the intermediate subgraphs is a $1/2$ -approximation to the densest subgraph. Unfortunately, this peeling process is exactly the peeling process used to compute the k -core of the graph, which we know to be P -complete.

In this paper, we study parallel algorithms for the k -clique densest subgraph, a generalization of the densest subgraph problem that was first introduced by Tsourakakis [317]. Tsourakakis presents a sequential $1/k$ -approximation algorithm based on iteratively peeling the vertex with minimum k -clique-count, as well as a parallel $1/(k(1 + \epsilon))$ -approximation algorithm based on a parallel $1/(2(1 + \epsilon))$ -approximation for the densest subgraph presented by Bahmani et al. [26]. We design parallel algorithms based on these two algorithms that are work-efficient and have polylogarithmic span.

Recently, Fang et al. [127] propose algorithms for finding the largest (j, Ψ) -core of a graph, which is the largest subgraph such that all vertices have at least j subgraphs Ψ incident on them. Ψ can be a k -clique or any other subgraph. In particular, they propose an algorithm for Ψ being a k -clique that peels vertices with larger clique counts first and show that their algorithm gives a $1/k$ -approximation to the k -clique densest subgraph.

Nucleus Decomposition. The peeling process used to solve the densest subgraph problem, and its generalization to k -cliques is an instance of the nucleus decomposition problem. A c - (r, s) **nucleus** is a maximal subgraph H of an undirected graph where each K_r in H has induced K_s degree at least c . The (r, s) **nucleus decomposition** problem is to compute all non-empty (r, s) nuclei. Note that the k -core of a graph corresponds to a k - $(1, 2)$ nucleus, and the vertex peeling problem used to solve k -clique densest subgraph corresponds to computing all of the c - $(1, k)$ nuclei. Although it is known that computing the $(1, 2)$ nucleus is P -complete for $k > 2$, nothing is known about the parallel complexity of computing higher $(1, s)$ nuclei for $s > 2$, and any value of k .

9.3 k -Clique Densest Subgraph

We present our new work-efficient parallel algorithms for the k -clique densest subgraph problem, as well as results for the vertex peeling, or the $(1, k)$ -nucleus de-

Algorithm 9.1 – Parallel vertex peeling (k -clique densest subgraph)

```
1: procedure UPDATE( $G = (V, E), k, DG, C, A$ )
2:   Initialize  $T$  to store  $k$ -clique counts per vertex in  $A$ 
3:   parfor  $v$  in  $A$  do
4:      $I \leftarrow \{u \mid u \in N_G(v) \text{ and } u \text{ has not been previously peeled}\}$ 
5:      $t' \leftarrow \text{REC-COUNT-CLIQUE-V}(DG, I, k - 1, C)$ 
6:     Store  $t'$  in  $T$ 
7:      $t \leftarrow \text{REDUCE-ADD}(T)$                                  $\triangleright$  Sum  $k$ -clique counts in  $T$ 
8:   return  $t$ 

9: procedure PEEL-CLIQUE(G = (V, E), k, DG, C, t)
10:    $\triangleright C$  is an array of  $k$ -clique counts per vertex and  $t$  is the total # of  $k$ -cliques
11:   Let  $B$  be a bucketing structure mapping  $V$  to buckets based on # of  $k$ -cliques
12:    $S^* \leftarrow G, d^* \leftarrow t/|V|$ 
13:    $f \leftarrow 0$ 
14:   while  $f < |V|$  do
15:      $A \leftarrow$  vertices in next bucket in  $B$  (to be peeled)
16:      $f \leftarrow f + |A|$ 
17:      $t' \leftarrow \text{UPDATE}(G, k, DG, C, A)$                        $\triangleright$  Update number of  $k$ -cliques
18:     Update the buckets of changed vertices in  $C$ , peeling  $A$ 
19:     if  $t'/(|V| - f) > d^*$  then
20:        $d^* \leftarrow t'/(|V| - f)$                                  $\triangleright$  Update maximum density
21:   return  $d^*$ 
```

composition problem, which is used as a sub-routine in one of our k -clique densest subgraph algorithms. Note that we use our k -clique counting algorithms from Section 5.3 throughout this section.

9.3.1 Vertex Peeling

Algorithm. Algorithm 9.1 describes our parallel algorithm for vertex peeling, which also gives a $1/k$ -approximate to the k -clique densest subgraph problem. The algorithm relies on Algorithm 5.3 from Section 5.3 to compute the initial per-vertex clique counts (C), which are given as an argument to the algorithm. The algorithm first initializes a parallel bucketing structure that stores buckets containing sets of vertices, where all vertices in the same bucket have the same clique count (Line 11). Then, while not all of the vertices have been peeled, it repeatedly extracts the vertices with the lowest induced k -clique count (Line 15), updates the count of the number of peeled vertices (Line 16), and updates the k -clique counts of vertices that are not yet finished that participate in k -cliques with the peeled vertices (Line 17). UPDATE also returns the number of k -cliques that were removed. Lastly, the algorithm checks if the new induced subgraph has higher density than the current maximum density, and if so updates the maximum density (Line 20). The UPDATE procedure performs the bulk of the work in the algorithm. It takes each vertex in A (vertices to be peeled), builds its induced neighborhood, and counts all $(k - 1)$ -cliques in this neighborhood using Algorithm 5.3, as these $(k - 1)$ -cliques together with a peeled vertex form a k -clique.

The algorithm presented above computes a *density* approximating the density of the k -clique densest subgraph. If computing a subgraph with this density is required, one can simply re-run the algorithm and emit a subgraph with density equal to the maximum density computed in the first run.

Correctness. It suffices to show that Algorithm 9.1 peels vertices in exactly the same order as Tsourakakis' sequential k -clique densest subgraph algorithm [317]. Although this statement is not true (our algorithm removes multiple vertices in the same bucket in parallel, whereas the sequential algorithm removes them one at a time), a slight modification of it is. In particular, we can show that ignoring the ordering of peeled vertices within the same core, our algorithm peels vertices in exactly the same order as the sequential peeling algorithm. One can easily show this fact using induction on the rounds of the algorithm (on each peeling step), and using the correctness of our k -clique counting methods from Section 5.3, which suffices to show the correctness of our peeling method. This gives us the desired $1/k$ -approximation of the k -clique densest subgraph.

Cost. We prove the following theorem regarding the complexity of our algorithm. Note that to analyze the span of our algorithm, we define $\rho_k(G)$ to be the $(1, k)$ -nucleus peeling complexity of G , or the number of rounds needed to peel the graph where in each round, all vertices with the minimum k -clique count are peeled.

Theorem 9.1. *There is a parallel algorithm computing a $1/k$ -approximation to the k -clique densest subgraph problem on an undirected graph G that runs in $O(m\alpha^{k-2} + \rho_k(G) \log n)$ expected amortized work and $O(\rho_k(G) \log^{k-2} n)$ span w.h.p., where $\rho_k(G)$ is the $(1, k)$ -nucleus peeling complexity of the graph G .*

Proof. The proof is similar in spirit to that of Theorem 5.2 but there are some subtle, and important differences. First, unlike in our k -clique counting algorithm (Algorithm 5.3), our peeling algorithm does not have the luxury of only finding k -cliques directed “upward” from a peeled vertex v . Instead, it must find *all* k -cliques that v participates in and decrement the counts of these k -cliques. Arguing that this does not cost a prohibitive amount of work is the main challenge of the proof. Importantly, our peeling algorithm calls the recursive subroutine REC-COUNT-CLIQUE-V of our k -clique counting algorithm directly, on a different input than used in the full k -clique counting algorithm, so the analysis of the work and span differs from the analysis given in Section 5.3.2.

We first account for the work and span of extracting and updating the bucketing structure. The overall work of inserting vertices into the bucketing structure is $O(n)$. Each vertex can have its bucket decremented at most once per k -clique, and since there are at most $O(m\alpha^{k-2})$ k -cliques, the overall cost for updating buckets of vertices is also the same. Lastly, removing the minimum bucket can be done in $O(\log n)$ amortized expected work and $O(\log n)$ span w.h.p., which costs a total of $O(\rho_k(G) \log n)$ amortized expected work, and $O(\rho_k(G) \log n)$ span w.h.p.

Next, to bound the cost of finding all k -cliques incident to a peeled vertex v , we rely on the Nash-Williams theorem [242], which provides a bound on the size of induced subgraphs in an arboricity α graph. More explicitly, the Nash-Williams

theorem states that a graph G has arboricity α if and only if for every $U \subseteq V$, $|G[U]| \leq \alpha(|U| - 1)$. Notably, the first call to REC-COUNT-CLIQUE-V performs intersect operations that essentially compute the induced subgraph on the neighbors of each peeled vertex v ; this is because during this first call, we intersect the directed neighbors of each vertex in $N_G(v)$ (that has not been previously peeled) with $N_G(v)$ itself, producing a pruned version of the induced subgraph of $N_G(v)$ on G . We have that for each $v \in V$, the induced subgraph on its neighbors has size $|G[N(v)]| \leq \alpha(|N(v)| - 1) = \alpha(d(v) - 1)$. Assuming for now that we can construct the induced subgraph on all vertex neighborhoods in work linear in their size, summed over all vertices, the overall cost is just

$$\sum_{v \in V} |G[N(v)]| \leq \alpha \sum_{v \in V} d(v) - 1 = O(m\alpha) \quad (9.1)$$

How do we build these subgraphs in the required work and span? Our approach is to do so using an argument similar to the elegant proof technique proposed in Chiba-Nishizeki's original k -clique listing algorithm. Because the first call to REC-COUNT-CLIQUE-V takes each vertex $u \in N_G(v)$ and intersects the directed neighbors $N_{DG}(u)$ with $N_G(v)$, we use $O(\min(d(u), d(v)))$ work to build the induced subgraph on v 's neighborhood. Observe that each edge in the graph is processed by an intersection in this way exactly once in each direction, once when each endpoint is peeled. By Lemma 2 of [72], we know that $\sum_{e=(u,v) \in E} \min(d(u), d(v)) = O(m\alpha)$ and therefore the overall work of performing all intersections is bounded by $O(m\alpha)$, and the per-vertex induced subgraphs can therefore also be built in the same bound. The span for this step is just $O(\log n)$ w.h.p. using parallel hash tables [148].

Lastly, we account for the remaining cost of performing k -clique counting within each round. We now recursively call REC-COUNT-CLIQUE-V $k-1$ times in total, as ℓ ranges from 1 to $k-1$, but the final recursive call returns the size of I immediately, and we have already discussed the work of the first call to REC-COUNTS-V. Considering the remaining $k-3$ recursive steps with non-trivial work, we have $O(m'\alpha^{k-3})$ work and $O(\log^{k-3} n)$ span where m' is the size of the vertex's induced neighborhood. Considering the work first, summed over all vertices induced neighborhoods, the total work is:

$$\sum_{v \in V} O(|G[N(v)]|\alpha^{k-3}) = O(m\alpha^{k-2})$$

which follows from Equation 9.1. The span follows, since adding in the span of the first recursive call, we have $O(\log^{k-2} n)$ span to update k -clique counts per peeled vertex, and there are $\rho_k(G)$ rounds by definition. \square

Discussion. To the best of our knowledge Tsourakakis presents the first sequential algorithm for this problem. His algorithm does not provide strong bounds on its work, or on the cost of bucketing. Considering Algorithm 4 from Tsourakakis' paper [317], by using a heap to store vertices based on their induced clique counts, the algorithm requires $O(n \log n)$ work for bucketing, not including the cost of counting k -cliques incident to removed vertices, which no serial algorithm prior to our work gave non-

trivial bounds for.

Sariyuce et al. [282] present a sequential algorithm for the more general (r, k) -nucleus decomposition problem. We observe that computing a $(1, k)$ -nucleus decomposition is equivalent to vertex peeling. Their fastest algorithm for this problem runs in $O(R(G, k))$ work and $O(C(G, k))$ space, where $R(G, k)$ is the cost of an arbitrary k -clique counting algorithm and $C(G, k)$ is the number of k -cliques in G . This algorithms works by essentially building a bipartite graph where one side of the bipartition is the vertices, and the other side are vertices representing k -cliques, and vertices are connected to k -cliques that contain them. The algorithm is to simply run k -core on this bipartite graph. They provide another algorithm which runs in $O(m + n)$ space, but requires $O(\sum_v d(v)^k)$ work, based on enumerating the cliques incident to peeled vertices. This could be as costly as $O(n^k)$ work. Our bounds are asymptotically better than theirs in all but the highly degenerate case where $C(G, k) = o(\rho \log n)$, since they can perform sequential bucketing in space proportional to the number of k -cliques in G . Note that Sariyuce et al. [283] also give a parallel (r, k) -nucleus decomposition algorithm, which is similarly not work-efficient.

9.3.2 Approximate Vertex Peeling

We present a $1/(k(1 + \epsilon))$ -approximate algorithm to the k -clique densest subgraph problem based on approximate peeling. The algorithm is similar to the peeling algorithm (Algorithm 9.1), and we describe the main differences in words in what follows. The approximate peeling algorithm also iteratively peels the graph until it becomes empty, maintaining the density of the current induced subgraph. Instead of peeling all vertices with the *minimum bucket*, in each round the algorithm sets a threshold $t = k(1 + \epsilon)\tau(S)$ where $\tau(S)$ is the density of the current subgraph S , and removes all vertices with current k -clique count at most τ . This can be done without using a bucketing structure at all by simply scanning all remaining vertices and filtering out vertices to be peeled. Tsourakakis [317] describes this procedure, which is analogous to the Bahmani et al. [26] $O(\log n)$ round $1/(2(1 + \epsilon))$ -approximation to the densest subgraph problem, and shows that when applied to the k -clique densest subgraph problem, computes a $1/(k(1 + \epsilon))$ -approximation in $O(\log n)$ rounds of peeling. Although the round-complexity in Tsourakakis' implementation is low, nothing non-trivial was known about its work.

Our implementation of Tsourakakis' algorithm is almost identical, except that we utilize the fast, parallel k -clique counting methods introduced in this paper. Using the same technique as in Algorithm 9.1 to decrement the local k -clique counts of peeled vertices, and by an identical analysis of the cost of peeling vertex neighborhoods, we have the following theorem:

Theorem 9.2. *There is a parallel algorithm computing a $1/(k(1 + \epsilon))$ -approximation to the k -clique densest subgraph problem on an undirected graph G that runs in $O(m\alpha^{k-2})$ expected work and $O(\log^{k-1} n)$ span w.h.p..*

Proof. The correctness and approximation guarantees of this algorithm follows from [317]. The work bound follows similarly to the proof of Theorem 9.1. We note that filtering

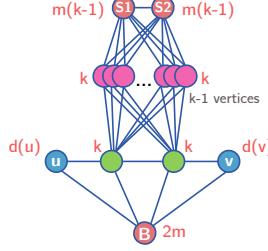


Figure 9.1: Gadget used for k -(1, 3) nucleus reduction. Note that the figure uses the notation k -(r, s) to refer to nucleii, whereas the text uses the notation c -(r, k). The red values show the initial clique-degree (K_s degree in this terminology, K_k degree in the main text) of each vertex before peeling.

the remaining vertices in each round can be done in just $O(n)$ total work, since a constant fraction of vertices are peeled in each round. The span bound follows from the fact that there are $O(\log n)$ rounds in total, and since each round runs in $O(\log^{k-2} n)$ span, again using the same argument as given in Theorem 9.1. \square

9.3.3 Practical Optimizations

In practice, we use the same optimizations as described in Section 5.3.4, for updating k -clique counts in every round. Also, we use the bucketing structure designed by Dhulipala et al. [95]. In particular, this structure keeps an array corresponding k -clique counts to the relevant vertices, but does not densely materialize sections of this array until enough vertices have been peeled to necessitate lookups into larger k -clique counts. Moreover, if large ranges of k -clique counts contain no vertices, this structure efficiently skips over such ranges. This allows for fast retrieval of vertices to be peeled in every round using linear space.

9.3.4 Parallel Complexity of c -(1, k)-nucleii

We consider the problem of computing the c -(1, k) nuclei for all values of c and k . We note that this problem is exactly the problem solved by the k -clique densest subgraph vertex peeling algorithm, from earlier in this section. Based on the work of Anderson and Mayr, we know that for the c -(1, 2) nucleii (which corresponds to the k -cores of the graph), this problem is in NC for $c = \{1, 2\}$ and is P-complete for $c > 2$. The proof by Anderson and Mayr does not immediately imply anything about the complexity of c -(1, k) nucleii for $k > 2$. In this chapter, we show the following theorem:

Theorem 9.3. *The c -(1, k) nucleus problem is P-complete for all constants $c > 2$, and $k > 2$.*

Proof. We first observe that since the number of k -cliques incident to each vertex can be efficiently computed in NC by both known results, and Theorem 5.2, the $1 - (1, k)$ decomposition problem is in NC for constant k (indeed it is in NC for any k s.t. $m\alpha^{k-2} \in \text{poly}(n)$)).

c – $(1, k)$ -nucleii when $c > 2$. Next, we study the parallel complexity of computing c – $(1, k)$ nucleii for $c > 2$. We will show that there is an NC reduction from the problem of whether the c – $(1, k)$ nucleii is non-empty, to the problem of deciding whether the c – $(1, k)$ nucleii is non-empty. We first discuss the reduction at a high level. The input is a graph, some c , and the problem is to decide whether the c – $(1, 2)$ nucleus is non-empty. The idea is to map the original peeling process to compute the c – $(1, 2)$ nucleus to a peeling process to compute the c – $(1, k)$ nucleus.

The reduction works as follows. will break up each edge in the graph into three vertices connected in a path, and create gadgets to increase each of these new ‘edge vertices’ k -degrees to c . The gadgets are constructed so that if a vertex on either endpoint of the path (an original vertex) has its k -degree go below c and is not in the c – $(1, k)$ nucleus, then the path corresponding to this edge will unravel, and the other original vertex will have its s -degree decremented by one, exactly as in the $(1, 2)$ peeling process. Formally, the gadgets constructs $c - 1$ K_k ’s between the two middle vertices in the path, and a set of gadget vertices for this edge. It also constructs one K_k between each original edge endpoint, its neighboring ‘edge vertex’, and a specially designated set of base vertices. The last part of the construction ensures that the gadget vertices have large enough K_k degree by creating $c - 1$ K_k between them and a set of special vertices, S_i ’s. Figure 9.1 shows an illustration of the reduction for $k = 3$, and marks the initial K_k degrees of each vertex in red.

To argue that this reduction is correct, it suffices to show that the c – $(1, k)$ -nucleus is non-empty if and only if the c -core of the original graph is non-empty. We only argue the reverse direction, since the proof for the forward direction is almost identical. Suppose the input graph has a non-empty c -core, C . Then, observe that all of the original vertices corresponding to C in the reduction graph, G' , will have degree at least c . Furthermore, all of the ‘edge vertices’ corresponding to edges in the c -core initially have K_k degree exactly c , the gadget vertices corresponding to these edges have K_k degree exactly c . It remains to argue that the special vertices, and the base vertices have sufficient K_k degree. Observe that the special vertices connected to all gadget vertices for the edges form K_k cliques with all gadget vertices, and thus have K_k degree $(c - 1)|E(G[C])|$ where $E(G[C])$ is the set of edges in the induced subgraph on C . Since a c core on C vertices must have at least $c|C|$ edges, $(c - 1)|E(G[C])| \geq (c - 1)c|C| > c$. Similarly for the base vertices, they form a single K_k for each edge in the c -core, and so the base vertices have K_k degree at least $c|C|$. Thus the subgraph corresponding to the original vertices, edge-vertices, gadget-vertices for these edges, and the special and base vertices all have sufficient K_k degree to form a non-empty c – $(1, k)$ nucleus.

By the discussion above, we have shown that computing c – $(1, k)$ -nucleii is P-complete for $c > 2$, a strengthening of the original P-completeness result of Anderson and Mayr. \square

$(1, k)$ -nucleii when $c = 2$. An interesting question is to understand the parallel complexity of computing $2 - (1, k)$ nucleii for any constant k . For $k = 2$, Anderson and Mayr observed that this problem is actually in NC. The idea of their algorithm is to apply list-contraction to the graph to remove all paths in parallel. Concretely, the

algorithm flips coins on all degree two vertices. Vertices with degree higher than two do not flip, but can be contracted into. If a vertex flips heads, and its two neighbors flip tails (or can be contracted into), the vertex splices itself out of the graph. If the vertex has degree one, it simply decreases the degree of its neighbor and deletes itself since it cannot be part of the 2-(1, 2) core. It is easy to see that this process terminates after $O(\log n)$ rounds w.h.p. and computes the 2-(1, 2) core of the graph. We leave it for future work to determine whether a similar algorithm can find the 2-(1, k) nucleii in NC for $k > 2$.

9.4 Experiments

Environment. We run most of our experiments on a c2-standard-60 Google Cloud instance, which consist of 60 cores (with two-way hyper-threading), with 3.8GHz Intel Xeon Scalable (Cascade Lake) processors and 240 GiB of main memory. For our large compressed graphs, we instead use a m1-ultramem-160 Google Cloud instance, which consists of 80 cores (with two-way hyper-threading), with 2.6GHz Intel Xeon E7 (Broadwell E7) processors and 3844 GiB of main memory. We use the work-stealing scheduler PARLAYLIB by Blelloch *et al.* [46] for our k -clique peeling runtimes (approximate and exact). Note that we used OpenMP for our k -clique counting experiments in Section 5.4. While OpenMP’s scheduling tends to be sufficiently fast for k -clique counting, we find that PARLAYLIB gives better performance for peeling algorithms. Finally, we compile our programs with g++ (version 7.3.1) using the -O3 flag. We terminate any experiment that takes over 5 hours, except for experiments on compressed graphs.

We test our algorithms on real-world undirected graphs from the Stanford Network Analysis Project (SNAP) [207], namely autonomous systems by Skitter (as-skitter), DBLP communities (com-dblp), Orkut communities (com-orkut), Friendster communities (com-friendster), and LiveJournal communities (com-lj). Note that Table 5.1 in Section 5.4 describes the sizes and k -clique counts for these graphs. Table 9.1 describes the peeling rounds, k -clique core size and maximum k -clique densities that we obtained from our algorithms.

Also, for our peeling algorithms, we test different orientations, including the Goodrich-Pszona and Barenboim-Elkin orientations discussed in Section 5.3.1, both with $\varepsilon = 1$. We additionally test other orientations that do not give work-efficient and polylogarithmic-span bounds for counting, but are fast in practice, including the orientation given by ranking vertices by increasing degree and directing low-degree vertices to high-degree vertices, the orientation given by k -core ordering, and the orientation given by the original ordering of vertices in the graph as given.

Moreover, we compare our algorithms against Danisch et al.’s KCLIST algorithm [86], which contains the state-of-the-art parallel and sequential k -clique peeling implementations. We include a simple modification to their k -clique listing code to support faster k -clique counting; we forego the final iteration over completed k -cliques and simply return the number of these k -cliques, to be added to a total clique count. Note that KCLIST also offers the option of node or edge parallelism, but only offers a k -core

| | | $k = 4$ | $k = 5$ | $k = 6$ | $k = 7$ | $k = 8$ | $k = 9$ |
|-------------------|------------------------------------|-----------|-------------|------------------------|---------------------|------------------------|------------------------|
| as-skitter | ρ_k | 11,669 | 17,720 | 22,091 | 23,988 | 23,095 | 21,538 |
| | k -clique core | 109,398 | 1,749,976 | 20,018,637 | 182,528,468 | 1.350×10^9 | 9.355×10^9 |
| | Max density (k -clique peeling) | 297,096 | 5,598,323 | 75,372,336 | 782,071,056 | 6.183×10^9 | 4.080×10^{10} |
| com dblp | ρ_k | 441 | 343 | 240 | 166 | 127 | 103 |
| | k -clique core | 234,136 | 6,438,740 | 140,364,532 | 2.527×10^9 | 3.862×10^{10} | 5.117×10^{11} |
| | Max density (k -clique peeling) | 234,136 | 6,438,740 | 140,364,532 | 2.527×10^9 | 3.862×10^{10} | 5.117×10^{11} |
| com orkut | ρ_k | 94,931 | 160,577 | 210,966 | 236,623 | 241,330 | — |
| | k -clique core | 117,182 | 2,115,900 | 29,272,988 | 312,629,724 | 2.741×10^9 | — |
| | Max density (k -clique peeling) | 340,997 | 4,882,477 | 73,696,814 | 883,634,847 | 8.332×10^9 | — |
| as skitter | ρ_k | 140,705 | 249,605 | 339,347 | — | — | — |
| | k -clique core | 349,377 | 11,001,375 | 274,901,025 | — | — | — |
| | Max density (k -clique peeling) | 428,928 | 12,762,919 | 363,676,399 | — | — | — |
| com dblp | ρ_k | 29,514 | 42,994 | 50,159 | — | — | — |
| | k -clique core | 7,660,975 | 679,343,769 | 4.796×10^{10} | — | — | — |
| | Max density (k -clique peeling) | 9,031,923 | 839,813,448 | 6.199×10^{10} | — | — | — |

Table 9.1: Relevant k -clique peeling statistics for the SNAP graphs that we experimented on. We do not have statistics for certain graphs for large values of k , because the corresponding k -clique peeling algorithms did not terminate in under 5 hours; these entries are represented by a dash.

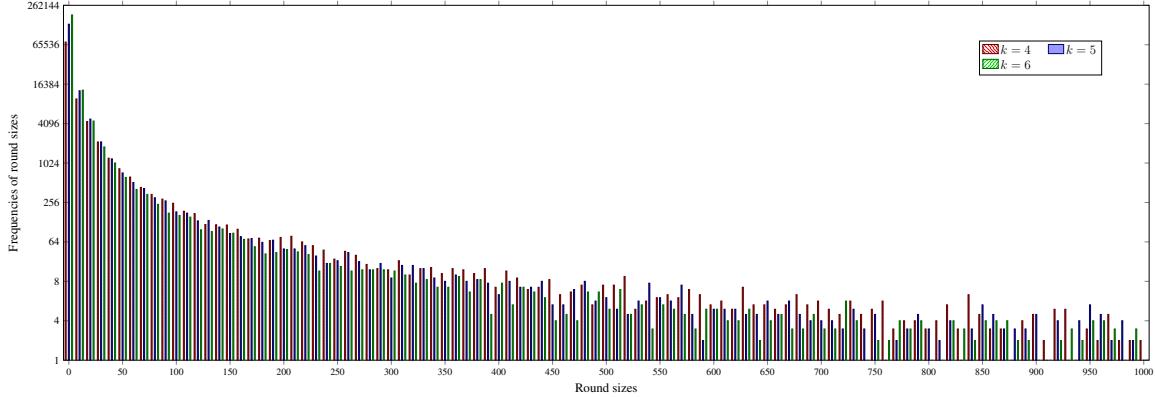


Figure 9.2: These are the frequencies of the number of vertices peeled in a parallel round using PEEL-CLIQUEs, for k -clique peeling on com-orcut ($4 \leq k \leq 6$). Note that rounds with more than 1000 vertices peeled have been truncated; these truncated round frequencies are very low, most often consisting of 0 rounds. Also, note that the frequencies are given in a log scale.

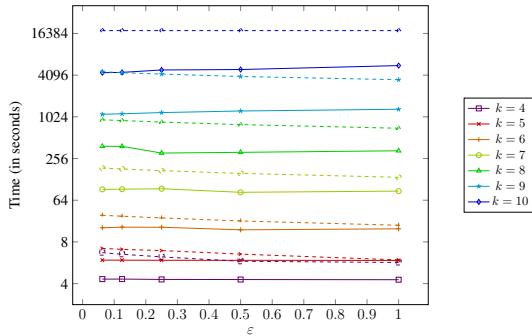


Figure 9.3: Parallel runtimes for approximate k -clique peeling using APPROX-PEEL-CLIQUEs (solid lines) and KCLIST (dashed lines). These runtimes were obtained on com-orcut, varying over ϵ , giving a $1/(k(1 + \epsilon))$ -approximation of the k -clique densest subgraph. Note that these runtimes were obtained using the orientation given by degree ordering, and the runtimes are given in a log scale. Moreover, we cut off KCLIST's runtimes at 5 hours, which occurred for $k = 10$ over all ϵ .

ordering to orient the input graphs.

Peeling results. Table 9.2 shows the best parallel and sequential runtimes for k -

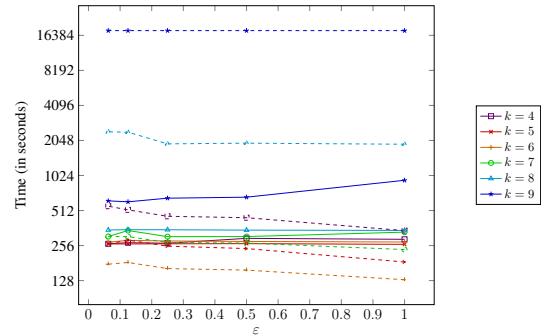


Figure 9.4: These are the parallel runtimes for approximate k -clique peeling using APPROX-PEEL-CLIQUEs (solid lines) and KCLIST (dashed lines). These runtimes were obtained on com-friendster, varying over ϵ , giving a $1/(k(1 + \epsilon))$ -approximation of the k -clique densest subgraph. Note that these runtimes were obtained using the orientation given by degree ordering, and the runtimes are given in a log scale. Moreover, we cut off KCLIST's runtimes at 5 hours, which occurred for $k = 9$ over all ϵ .

| | | $k = 4$ | $k = 5$ | $k = 6$ | $k = 7$ | $k = 8$ | $k = 9$ |
|----------------|-----------------------|----------------|--------------------------|--------------------------|-----------------------------|-----------------------------|-----------------------------|
| as-skitter | PEEL-CLIQUEs T_{60} | 4.85 | 11.42 | 47.17 | 282.40 | 1750.72 | 10460.60 |
| | PEEL-CLIQUEs T_1 | 7.13 | 14.78 | 56.47 | 338.70 | 2123.42 | > 5 hrs |
| | KCLIST T_1 | 6.34 | 15.24 | 68.35 | 417.75 | 2644.21 | > 5 hrs |
| com dblp | PEEL-CLIQUEs T_{60} | 0.21 | 0.23 [°] | 1.29 [°] | 18.77 | 276.69 [°] | 3487.09 [°] |
| | PEEL-CLIQUEs T_1 | 0.37 | 1.378 | 17.99 | 258.24 | 3373.05 | > 5 hrs |
| | KCLIST T_1 | 0.25 | 1.10 | 14.98 | 221.98 | 2955.87 | > 5 hrs |
| com orkut | PEEL-CLIQUEs T_{60} | 76.91 | 221.28 | 721.73 | 2466.99 [°] | 9062.99 [°] | > 5 hrs |
| | PEEL-CLIQUEs T_1 | 184.28 | 422.20 | 1032.19 | 3123.72 | > 5 hrs | > 5 hrs |
| | KCLIST T_1 | 218.94 | 587.24 | 2029.43 | 7414.77 | > 5 hrs | > 5 hrs |
| com friendster | PEEL-CLIQUEs T_{60} | 1747.92 | 4144.96 | 6870.06 | > 5 hrs | > 5 hrs | > 5 hrs |
| | PEEL-CLIQUEs T_1 | 11540.73 | 12932.28 | 14112.95 | > 5 hrs | > 5 hrs | > 5 hrs |
| | KCLIST T_1 | 3216.92 | 4325.73 | 6933.32 | > 5 hrs | > 5 hrs | > 5 hrs |
| com lj | PEEL-CLIQUEs T_{60} | 26.36 | 324.77 | 12920.08 | > 5 hrs | > 5 hrs | > 5 hrs |
| | PEEL-CLIQUEs T_1 | 70.12 | 822.10 | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs |
| | KCLIST T_1 | 42.16 | 839.13 | > 5 hrs | > 5 hrs | > 5 hrs | > 5 hrs |

Table 9.2: Best runtimes in seconds for our parallel and single-threaded k -clique peeling algorithm (PEEL-CLIQUEs), as well as the best sequential runtimes from previous work (KCLIST) [86]. The fastest runtimes for each experiment are bolded and in green. All runtimes are from tests in the same computing environment, and include only time spent peeling. For our parallel runtimes, we have chosen the fastest orientations per experiment, while for our serial runtimes, we have fixed the orientation given by degree ordering. For the parallel runtimes from COUNT-CLIQUEs, we have noted the orientation used; [°] refers to the Goodrich-Pszona orientation, and no superscript refers to the orientation given by degree ordering.

clique peeling over the SNAP datasets, from our implementation and from `KCLIST` (note that `KCLIST` only includes sequential k -clique peeling). Overall, our parallel implementation obtains between 1.01–11.83x speedups over `KCLIST`'s sequential runtimes. The higher speedups occur in graphs that require proportionally fewer parallel peeling rounds ρ_k compared to its size; notably, `com dblp` requires few parallel peeling rounds, and we see between 4.60–11.83x speedups over `KCLIST` on `com dblp` for $k \geq 5$. As such, our parallel speedups are constrained by ρ_k .

Also, we note that while we generally do not reach large enough k in our peeling algorithms for different orientations to have a significant effect, we do see that on `com dblp` and `com orkut` for large k , the Goodrich-Pszona orientation results in slightly faster k -clique peeling runtimes. Similarly to k -clique counting, the orientation obtained by degree ordering gives the fastest runtimes for small k .

Moreover, we see between 1.19–13.76x self-relative speedups from our single-threaded runtimes. Our single-threaded runtimes are generally slower than `KCLIST`'s sequential runtimes owing to the parallel overhead necessary to aggregate k -clique counting updates between rounds. The single-threaded runtimes are particularly slow for large graphs and small k , where there is greater parallel overhead for aggregation in rounds that do not peel many vertices. Figure 9.2 shows the frequencies of the different numbers of vertices peeled in each parallel round for `com orkut`, for $4 \leq k \leq 6$. A significant number of rounds contain fewer than 50 vertices peeled, and by the time we reach the tail of the histogram, there are very few parallel rounds with a large number of vertices peeled.

We also tested Tsourakakis's [317] triangle densest subgraph implementation. Unfortunately this implementation requires too much memory to run for `as skitter`, `com orkut`, `com friendster`, and `com lj` on our machines. It completes 3-clique peeling on `com dblp` in 0.86 seconds, while our parallel `COUNT CLIQUES` takes 0.26 seconds.

Approximate peeling results. Figures 9.3 and 9.4 show parallel runtimes for approximate k -clique peeling on `com orkut` and `com friendster`, respectively. Note that we tested both our implementation (`APPROX PEEL CLIQUES`) and `KCLIST`, both of which are parallel algorithms. We see that in general, there is not a significant difference in runtimes over different ϵ for both implementations. Also, our parallel algorithm is over 29.59x faster than `KCLIST` for large k . We note that our parallel algorithm is slower on `com friendster` for small k ; this is because `KCLIST` uses a serial heap to recompute vertices that must be peeled, which is more amenable over small rounds, while our implementation incurs additional parallel overhead to recompute peeled vertices in parallel, which is mitigated over larger k and smaller ϵ .

In terms of percentage error in the maximum k -clique density obtained compared to the density obtained from k -clique peeling, we see between 48.58–77.88% error on `com orkut` and between 5.95–80.83% error on `com friendster`. Note that we see lower percentage error for the larger graph, `com friendster`, and the lowest percentage errors were obtained on small k and small ϵ .

9.5 Discussion

We have presented new work-efficient parallel algorithms for k -clique peeling with low span. We have shown experimentally that our implementations achieve good parallel speedups and significantly outperform state-of-the-art implementations.

Chapter 10

Nucleus Decomposition

10.1 Introduction

Recently, Sariyüce *et al.* [284] introduced the nucleus decomposition problem, which generalizes the influential notions of k -cores and k -trusses to k -(r, s) nuclei, and can better capture higher-order structures in the graph. Informally, a k -(r, s) nucleus is the maximal induced subgraph such that every r -clique in the subgraph is contained in at least k s -cliques. The goal of the (r, s) nucleus decomposition problem is to identify for each r -clique in the graph, the largest k such that it is in a k -(r, s) nucleus.

Solving the (r, s) nucleus decomposition problem is a significant computational challenge for several reasons. First, simply counting and enumerating s -cliques is a challenging task, even for modest s . Second, storing information for all r -cliques can require a large amount of space, even for relatively small graphs. Third, engineering fast and high-performance solutions to this problem requires taking advantage of parallelism due to the computationally-intensive nature of listing cliques. There are two well-known parallel paradigms for approaching the (r, s) nucleus decomposition problem, a global peeling-based model and a local update model that iterates until convergence [283]. The former is inherently challenging to parallelize due to sequential dependencies and necessary synchronization steps [283], which we address in this chapter, and we demonstrate that the latter requires orders of magnitude more work to converge to the same solution and is thus less performant.

Lastly, it is unknown whether existing sequential and parallel algorithms for this problem are theoretically efficient. Notably, existing algorithms perform more work than the fastest theoretical algorithms for k -clique enumeration on sparse graphs [72, 296], and it is open whether one can solve the (r, s) nucleus decomposition problem in the same work as s -clique enumeration.

In this chapter, we design a novel parallel algorithm for the nucleus decomposition problem. We address the computational challenges by designing a theoretically efficient parallel algorithm for (r, s) nucleus decomposition that nearly matches the work for s -clique enumeration, along with new techniques that improve the space and

cache efficiency of our solutions. The key to our theoretical efficiency is combining a combinatorial lemma bounding the total sum over all k -cliques in the graph of the minimum degree vertex in this clique [116] with a theoretically efficient k -clique listing algorithm [296], enabling us to provide a strong upper bound on the overall work of our algorithm. As a byproduct, we also obtain the most theoretically-efficient serial algorithm for (r, s) nucleus decomposition. We provide several new optimizations for improving the practical efficiency of our algorithm, including a new multi-level hash table structure to space efficiently store data associated with cliques, a technique for efficiently traversing this structure in a cache-friendly manner, and methods for reducing contention and further reducing space usage.

Finally, we experimentally study our parallel algorithm on various real-world graphs and (r, s) values, and find that it achieves between 3.31–40.14x self-relative speedup on a 30-core machine with two-way hyper-threading. The only existing parallel algorithm for nucleus decomposition is by Sariyüce *et al.* [283], but their algorithm requires much more work than the best sequential algorithm. Our algorithm achieves between 1.04–54.96x speedup over the state-of-the-art parallel nucleus decomposition of Sariyüce *et al.*, and our algorithm can scale to larger (r, s) values, due to our improved theoretical efficiency and our proposed optimizations. We are able to compute the (r, s) nucleus decomposition for $r > 3$ and $s > 4$ on several million-scale graphs for the first time. We summarize our contributions below:

- The first theoretically-efficient parallel algorithm for the nucleus decomposition problem.
- A collection of practical optimizations that enable us to design a fast implementation of our algorithm.
- Comprehensive experiments showing that our new algorithm achieves up to a 55x speedup over the state-of-the-art algorithm by Sariyüce *et al.*, and up to a 40x self-relative parallel speedup on a 30-core machine with two-way hyper-threading.

Our code is publicly available at: <https://github.com/jeshi96/arb-nucleus-decomp>.

10.2 Related Work

The nucleus decomposition problem is inspired by and closely related to the k -core problem, which was defined independently by Seidman [290], and by Matula and Beck [229]. The k -core of a graph is the maximal subgraph of the graph where the induced degree of every vertex is at least k . The *coreness* of a vertex is the maximum value of k such that the vertex participates in a k -core. Matula and Beck provided a linear time algorithm based on peeling vertices that computes the coreness value of all vertices [229].

In subsequent years, many concepts capturing dense near-clique substructures were proposed, including k -trusses (or triangle-cores), k -plexes [291], and n -clans and n -clubs [236]. In particular, k -trusses were proposed independently by Cohen [79], Zhang *et al.* [350], and Zhou *et al.* [353] with the goal of efficiently obtaining dense clique-like substructures. Unlike other near-clique substructures like k -

plexes, n -clans, and n -clubs, which are computationally intractable to enumerate and count, k -trusses can be efficiently found in polynomial-time. Many parallel, external-memory, and distributed algorithms have been developed in the past decade for k -cores [238, 128, 189, 185, 95, 337] and k -trusses [324, 71, 355, 186, 305, 70, 83, 218, 45], and computing all trussness values of a graph is one of the challenge problems in the yearly MIT GraphChallenge [275]. A related problem is to compute the k -clique densest subgraph [317] and (k, Ψ) -core [127], for which efficient parallel algorithms have been recently designed [296]. The concept of a (r, s) nucleus decomposition was first proposed by Sariyüce *et al.* as a principled approach to discovering dense substructures in graphs that generalizes k -cores and k -trusses [284]. They also proposed an algorithm for efficiently finding the hierarchy associated with a (r, s) nucleus decomposition [281]. Sariyüce *et al.* later proposed parallel algorithms for nucleus decomposition based on local computation [283]. Recent work has studied nucleus decomposition in probabilistic graphs [123].

Clique counting and enumeration are fundamental subproblems required for computing nucleus decompositions. A trivial algorithm enumerates c -cliques in $O(n^c)$ work, and using a thresholding argument improves the work for counting to $O(m^{c/2})$ [14]. The current fastest combinatorial algorithms for c -clique enumeration for sparse graphs are based on the seminal results of Chiba and Nishizeki [72], who show that all c -cliques can be enumerated in $O(m\alpha^{c-2})$ where α is the arboricity of the graph. We defer to the survey of Williams for an overview of theoretical algorithms for this problem [322]. The current state-of-the-art practical algorithms for k -clique counting are all based on the Chiba-Nishizeki algorithm [86, 296, 210].

Researchers have also studied k -core-like computations in bipartite graphs [298, 329, 202, 282, 215], as well as how to maintain k -cores and k -trusses in dynamic graphs [209, 337, 349, 279, 11, 171, 18, 214, 224, 169, 310, 184, 222, 351, 172]. Very recently, Sariyüce proposed a motif-based decomposition, which generalizes the connection between r -cliques and s -cliques in nucleus decomposition to any pair of subgraphs [278].

10.3 Preliminaries

A c - (r, s) **nucleus** is a maximal subgraph H of an undirected graph formed by the union of s -cliques C_s , such that each r -clique C_r in H has induced s -clique degree at least c (i.e., each r -clique is contained within at least c induced s -cliques). The (r, s) **nucleus decomposition problem** is to compute all non-empty (r, s) -nuclei. Our algorithm outputs the (r, s) -**clique core number** of each r -clique C_r , or the maximum c such that C_r is contained within a c - (r, s) nucleus.¹ The k -core and k -truss problems correspond to the k - $(1, 2)$ and k - $(2, 3)$ nucleus, respectively.

¹The original definition of (r, s) nucleus decomposition is stricter, in that it additionally requires any two r -cliques in the maximal subgraph H to be connected via s -cliques [284, 281]. This requires additional work to partition the r -cliques, which the previous parallel algorithm [283] does not perform, and is also out of our scope of this chapter. Note that we discuss this in more detail in Chapter 11.

10.4 (r, s) Nucleus Decomposition

We present here our parallel work-efficient (r, s) nucleus decomposition algorithm. Importantly, we introduce new theoretical bounds for (r, s) nucleus decomposition, which also improve upon the previous best sequential bounds. We discuss in Section 10.4.1 a key s -clique counting subroutine, and we present ARB-NUCLEUS, our parallel (r, s) nucleus decomposition algorithm, in Section 10.4.2.

10.4.1 Recursive s -clique Counting Algorithm

We first introduce an important subroutine, REC-LIST-CLIQUEs, based on previous work from Shi *et al.* [296], which recursively finds and lists c -cliques in parallel. This subroutine is based on a state-of-the-art c -clique listing algorithm, which in practice balances performance and memory efficiency, outperforming other baselines, particularly for large graphs with hundreds of billions of edges [296]. This subroutine has been modified from previous work to integrate in our parallel (r, s) nucleus decomposition algorithm, to both count the number of s -cliques incident on each r -clique, and update the s -clique counts after peeling subsets of r -cliques. The main idea for REC-LIST-CLIQUEs is to iteratively grow each c -clique by maintaining at every step a set of candidate vertices that are neighbors to all vertices in the c -clique so far, and prune this set as we add more vertices to the c -clique.

The pseudocode for the algorithm is shown in Algorithm 10.1. REC-LIST-CLIQUEs takes as input a directed graph DG , a set I of potential neighbors to complete the clique (which for c -clique listing is initially V), the recursive level $r\ell$ (which for c -clique listing is initially set to c), a set C of vertices in the clique so far (which for s -clique listing is initially empty), and a function f to apply to each discovered c -clique. The directed graph DG is an $O(\alpha)$ -orientation of the original undirected graph, which allows us to reduce the work required for computing intersections. REC-LIST-CLIQUEs then uses repeated intersections on the set I and the directed neighbors of each vertex in I to find valid vertices to add to the clique C (Line 8), and recurses on the updated clique (Line 9). At the final recursive level, REC-LIST-CLIQUEs applies the user-specified function f on each discovered clique (Line 5). Assuming that DG is an $O(\alpha)$ -oriented graph, and excluding the time required to obtain DG , REC-LIST-CLIQUEs can perform c -clique listing in $O(m\alpha^{c-2})$ work and $O(c \log n)$ span w.h.p. [296].

10.4.2 (r, s) Nucleus Decomposition Algorithm

We now describe our parallel nucleus decomposition algorithm, ARB-NUCLEUS. ARB-NUCLEUS computes the (r, s) nucleus decomposition by first computing and storing the incident s -clique counts of each r -clique. It then proceeds in rounds, where in each round, it peels, or implicitly removes, the r -cliques with the minimum s -clique counts. It updates the s -clique counts of the remaining unpeeled r -cliques, by decrementing the count by 1 for each s -clique that the unpeeled r -clique shares peeled r -cliques

Algorithm 10.1 – Parallel c -clique listing algorithm.

```

1: procedure REC-LIST-CLIQUE(DG, I, r $\ell$ , C, f)
2:    $\triangleright$  DG is the directed graph, I is the set of potential neighbors to complete the
      clique, r $\ell$  is the recursive level, C is the set of vertices in the clique so far, and f is the
      desired function to apply to  $c$ -cliques.
3:   if r $\ell$  = 1 then
4:     parfor v in I do
5:       Apply f on the  $c$ -clique  $C \cup \{v\}$ 
6:   return
7:   parfor v in I do
8:      $I' \leftarrow \text{INTERSECT}(I, N_{DG}(v))$             $\triangleright$  Intersect I with directed neighbors of v
9:     REC-LIST-CLIQUE(DG, I', r $\ell$  - 1, C  $\cup$  {v}, f)     $\triangleright$  Add v to the  $c$ -clique and
      recurse

```

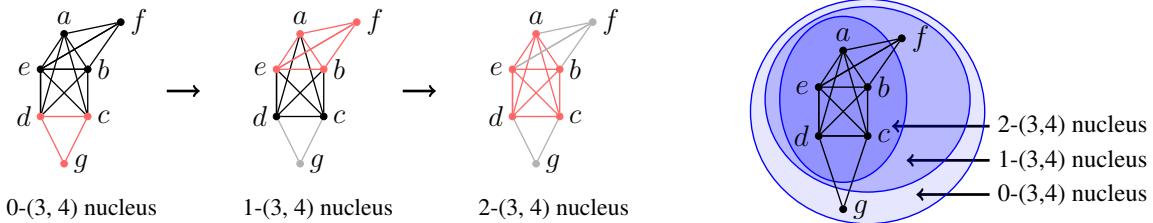


Figure 10.1: An example of our parallel nucleus decomposition algorithm ARB-NUCLEUS for $(r, s) = (3, 4)$. At each step, we peel in parallel all triangles (3-cliques) with the minimum 4-clique count; the vertices and edges that compose of these triangles are highlighted in red. We then recompute the 4-clique count on the remaining triangles. Vertices and edges that no longer participate in any active triangles, due to previously peeled triangles, are shown in gray. Each step is labeled with the k -(3,4) nucleus discovered, where k is the 4-clique count of the triangles highlighted in red. The figure below labels each k -(3,4) nucleus.

with. ARB-NUCLEUS uses REC-LIST-CLIQUE as a subroutine, to compute and update s -clique counts.

Example. An example of our algorithm for $(r, s) = (3, 4)$ is shown in Figure 10.1. There are 14 total triangles in the example graph, namely those given by any three vertices in $\{a, b, c, d, e\}$, and the additional triangles abf , bef , aef , and $c当地$. At the start of the algorithm, $c当地$ is incident to no 4-cliques, while abf , bef , and aef are each incident to one 4-clique. Also, abe is incident to three 4-cliques, and the rest of the triangles are incident to two 4-cliques. Thus, only $c当地$ is peeled in the first round, and has a (3, 4)-clique-core number of 0. Then, abf , bef , and aef are peeled simultaneously in the second round, each with a 4-clique count of one, which is also their (3, 4)-clique-core number. Peeling these triangles updates the 4-clique count of abe to two, and in the third round, all remaining triangles have the same 4-clique count (and form the 2-(3, 4) nucleus) and are peeled simultaneously, completing the algorithm.

Our Algorithm. We now provide a more detailed description of the algorithm.

Before rounds:

T : key (3-clique), value (4-clique count)

| | | | | | | | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| $abf, 1$ | $aef, 1$ | $bef, 1$ | $abc, 2$ | $abd, 2$ | $abe, 3$ | $acd, 2$ | $ace, 2$ | $ade, 2$ | $bcd, 2$ | $bce, 2$ | $bde, 2$ | $cde, 2$ | $cdg, 0$ |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|

| | | | | |
|-------|----------|-----------------|----------|---|
| B : | bucket 0 | cdg | bucket 2 | $abc, abd, acd, ace, ade, bcd, bce, bde, cde$ |
| | bucket 1 | abf, aef, bef | bucket 3 | abe |

Round 1: $A = \text{bucket } 0 = \{cdg\}$, finished = 1

$U : \emptyset$

T :

| | | | | | | | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| $abf, 1$ | $aef, 1$ | $bef, 1$ | $abc, 2$ | $abd, 2$ | $abe, 3$ | $acd, 2$ | $ace, 2$ | $ade, 2$ | $bcd, 2$ | $bce, 2$ | $bde, 2$ | $cde, 2$ | $cdg, 0$ |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|

| | | | | |
|-------|----------|---|----------|-------|
| B : | bucket 1 | abf, aef, bef | bucket 3 | abe |
| | bucket 2 | $abc, abd, acd, ace, ade, bcd, bce, bde, cde$ | | |

Round 2: $A = \text{bucket } 1 = \{abf, aef, bef\}$, finished = 4

$U : abe$

T :

| | | | | | | | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| $abf, 0$ | $aef, 0$ | $bef, 0$ | $abc, 2$ | $abd, 2$ | $abe, 2$ | $acd, 2$ | $ace, 2$ | $ade, 2$ | $bcd, 2$ | $bce, 2$ | $bde, 2$ | $cde, 2$ | $cdg, 0$ |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|

| | | |
|-------|----------|--|
| B : | bucket 2 | $abc, abd, abe, acd, ace, ade, bcd, bce, bde, cde$ |
|-------|----------|--|

Round 3: $A = \text{bucket } 2$, finished = 14

Figure 10.2: An example of the data structures in ARB-NUCLEUS during each round of (3, 4) nucleus decomposition, on the graph in Figure 10.1.

Algorithm 10.2 presents the pseudocode for ARB-NUCLEUS. We refer to Figure 10.2, which shows the state of each data structure after each round of ARB-NUCLEUS, for an example of (3, 4) nucleus decomposition on the graph in Figure 10.1. ARB-NUCLEUS first directs the graph G such that every vertex has out-degree $O(\alpha)$ (Line 20), using an efficient low out-degree orientation algorithm by Shi *et al.* [296]. Then, it initializes a parallel hash table T to store s -clique counts, keyed by r -clique counts, and calls the recursive subroutine REC-LIST-CLIQUEs to count and store the number of s -cliques incident on each r -clique (Lines 21–22). It uses COUNT-FUNC (Lines 2–4) to atomically increment the count for each r -clique found in each discovered s -clique. As shown in Figure 10.2, before any rounds of peeling, T contains the 4-clique count incident to each triangle. The algorithm also initializes a parallel bucketing structure B that stores sets of r -cliques with the same s -clique counts (Line 23). We have four initial buckets in Figure 10.2. The first bucket contains cdg , which is incident to no 4-cliques, and the second bucket contains abf , aef , and bef , which are incident to exactly one 4-clique. The third bucket contains the triangles incident to exactly two 4-cliques, abc , abd , acd , ace , ade , bcd , bce , bde , and cde . The final bucket contains abe , which is incident to exactly three 4-cliques.

²When COUNT-FUNC and UPDATE-FUNC are invoked on lines 17 and 22, all arguments except the last argument S are bound to each function. This is because these functions are then called in REC-LIST-CLIQUEs, where they take as input an s -clique S , which is precisely the last argument.

Algorithm 10.2 – Parallel (r, s) nucleus decomposition algorithm

```

1: Initialize  $r, s$                                  $\triangleright r$  and  $s$  for  $(r, s)$  nucleus decomposition
2: procedure COUNT-FUNC( $T, S$ ) 2
3:   parfor all size  $r$  subsets  $R \subset S$  do
4:     Atomically add 1 to the  $s$ -clique count  $T[R]$ 
5: procedure UPDATE-FUNC( $U, A, T, S$ ) 2
6:   Let  $U' \leftarrow \{R \subset S \mid |R| = r \text{ and } R \notin A\}$ 
7:   Let  $a$  be the # of size  $r$  subsets  $R \subset S$  such that  $R \in A$ 
8:   if any  $R \in U'$  has been previously peeled then
9:     return
10:    parfor  $R$  in  $U'$  do
11:      Atomically subtract  $1/a$  from the  $s$ -clique count  $T[R]$ 
12:      Add  $R$  to  $U$ 
13: procedure UPDATE( $G = (V, E), DG, A, T$ )
14:   Initialize  $U$  to be a parallel hash table to store  $r$ -cliques with updated  $s$ -clique counts
        after peeling  $A$ 
15:   parfor  $R$  in  $A$  do
16:      $I \leftarrow \text{INTERSECT}(N_G(v) \text{ for } v \in R)$   $\triangleright$  Intersect the undirected neighbors of  $v \in R$ 
17:     REC-LIST-CLIQUE(S)( $DG, I, s - r, R, \text{UPDATE-FUNC}(U, A, T)$ )
18:   return  $U$ 
19: procedure ARB-NUCLEUS( $G = (V, E), \text{ORIENT}$ )
20:    $DG \leftarrow \text{ORIENT}(G)$             $\triangleright$  Apply a user-specified orientation algorithm
21:   Initialize  $T$  to be a parallel hash table with  $r$ -cliques as keys, and  $s$ -clique counts as
        values
22:   REC-LIST-CLIQUE(S)( $DG, V, s, \emptyset, \text{COUNT-FUNC}(T)$ )            $\triangleright$  Count  $s$ -cliques
23:   Let  $B$  be a bucketing structure mapping each  $r$ -clique to a bucket based on # of
         $s$ -cliques
24:    $\text{finished} \leftarrow 0$ 
25:   while  $\text{finished} < |T|$  do
26:      $A \leftarrow r$ -cliques in the next bucket in  $B$  (to be peeled)
27:      $\text{finished} \leftarrow \text{finished} + |A|$ 
28:      $U \leftarrow \text{UPDATE}(G, DG, A, T)$   $\triangleright$  Update # of  $s$ -cliques and return  $r$ -cliques with
        changed  $s$ -clique counts
29:     Update the buckets of  $r$ -cliques in  $U$ , peeling  $A$ 
30:   return  $B$ 

```

While not all of the r -cliques have been peeled, the algorithm repeatedly obtains the r -cliques incident upon the lowest number of induced s -cliques (Line 26), updates the count of the number of peeled r -cliques (Line 27), and updates the s -clique counts of r -cliques that participate in s -cliques with peeled r -cliques (Line 28). In Figure 10.2, we see that in the first round, the bucket with the least s -clique count is bucket 0, and **finished** is updated to the size of the bucket, or one triangle. No 4-cliques are involved, so no updates are made to T , and U remains empty. ARB-NUCLEUS then updates the buckets for r -cliques with changed s -clique counts (Line 29), and repeats until all r -cliques have been peeled. At the end, the algorithm returns the

bucketing structure, which maintains the (r, s) -core number of each r -clique. In the second round in Figure 10.2, the bucket with the least 4-clique count is bucket 1, containing abf , aef , and bef . We add 3 to `finished`, making its value 4. The sole 4-clique incident to these triangles is $abef$, so when these triangles are peeled, there is one fewer 4-clique incident on abe . Thus, the 4-clique count stored on abe in T is decremented by one, and abe is returned in the set U as the only triangle with a changed 4-clique count. Note that the $(3, 4)$ -clique core number of abf , aef , and bef is thus 1, which is implicitly maintained upon their removal from B . The bucket of abe is updated to 2, since abe now participates in only two 4-cliques. Then, in the final round, the rest of the triangles are all in the peeled bucket, bucket 2, and the `finished` count is updated to 14, or the total number of triangles. The implicitly maintained $(3, 4)$ -clique-core number of these triangles is 2, and since no triangles remain unpeeled, there are no further updates to the data structure and the algorithm returns.

The subroutine `UPDATE` (Lines 13–18) is the main subroutine of `ARB-NUCLEUS`, used on Line 28. It takes as additional input the directed graph DG , a set A of r -cliques to peel, and the parallel hash table T that stores current s -clique counts, and updates incident s -clique counts affected by peeling the r -cliques in A . It also returns the set of r -cliques that have not yet been peeled with their decremented s -clique counts. `UPDATE` first initializes a parallel hash table U to store the set of r -cliques with changed s -clique counts (Line 14).³ It then considers each r -clique $R \in A$ being peeled, and computes the intersection of the undirected neighbors of the vertices in R , which are stored in set I (Line 16). The vertices in I are candidate vertices to form s -cliques from R , and `UPDATE` uses I with the subroutine `REC-LIST-CLIQUE`s to find the remaining $s - r$ vertices to complete the s -cliques incident to R (Line 17). In the second round of Figure 10.2, abf , aef , and bef are being peeled. The intersection of the neighbors of a , b , and f adds vertex e to the discovered 4-clique. We thus find only one 4-clique incident to abf , and similarly, we find the same 4-clique incident to aef and bef .

For each s -clique S found, `REC-LIST-CLIQUE`s calls `UPDATE-FUNC` (Lines 6–12), which first checks if S contains r -cliques that were previously peeled (Line 8). If not, `UPDATE-FUNC` atomically subtracts $1/a$ from each s -clique count for each r -clique in S , where a is the number of size r subsets in S that are also being peeled (Line 11). This is to prevent over-counting—if r -cliques that participate in the same s -clique are peeled simultaneously, then they will each subtract $1/a$ from the s -clique count, and the total subtraction will sum to 1 for this s -clique. `UPDATE-FUNC` also adds each r -clique with updated counts to U (Line 12). In the second round of Figure 10.2, since abf , aef , and bef each discover the 4-clique $abef$, which consists of $a = 3$ triangles that are simultaneously being peeled, abf , aef , and bef each decrement $1/3$ from the 4-clique count of abe in T . In total, 1 is decremented from abe 's 4-clique count. Then, abe is added to U , since its 4-clique count has been updated. In the third

³Note that it is inefficient to initialize U here in the `UPDATE` in every round, although we do so in the pseudocode for simplicity. Instead, U should be initialized following Line 24, with size equal to the total number of r -cliques. Then, after Line 29, U can be efficiently cleared for the next round by rerunning the `UPDATE` subroutine solely to clear previously modified entries in U .

round, because all remaining triangles are being peeled in A , the set U' defined on Line 6, is either empty or contains a previously peeled triangle, by definition. Thus, UPDATE returns without performing any modifications to U or T , and no buckets are updated. Afterwards, the `finished` variable equals the total number of triangles, and the algorithm finishes.

We now discuss the theoretical efficiency our parallel nucleus decomposition algorithm. To show that our algorithm improves upon the best existing work bounds for the sequential nucleus decomposition algorithm, we use the following key lemma that upper bounds the sum of the minimum degrees of vertices over all c -cliques. We note that Chiba and Nishizeki [72] first proved this lemma for $c = 2$, and Eden *et al.* first proved this lemma for larger c [116]. We provide a similar proof here.

Lemma 10.1 ([116]). *For a graph G with arboricity α , over all c -cliques $C_c = \{v_1, \dots, v_c\}$ in G where $c \geq 1$, $\sum_{C_c} \min_{1 \leq i \leq c} \deg(v_i) = O(m\alpha^{c-1})$.*

Proof. First, the $c = 1$ case is easy, since $\sum_{v_i \in V} \deg(v_i) = 2m$, and the $c = 2$ case is proven by Chiba and Nishizeki [72]. We consider $c \geq 3$ for the rest of the proof.

We begin by rewriting the sum $\sum_{C_c} \min_{1 \leq i \leq c} \deg(v_i)$ to be taken over all edges $e \in E$ instead of over all c -cliques C_c . Importantly, we must assign each unique c -clique C_c to a unique edge e contained within C_c , to avoid double counting. We perform this assignment using an $O(\alpha)$ -orientation of the graph, where each c -clique C_c is an ordered list of vertices v_i for $1 \leq i \leq c$, with the ordering given by the $O(\alpha)$ -orientation. We assign each C_c to the edge e given by its first two vertices v_1 and v_2 .

Then, rewriting the sum $\sum_{C_c} \min_{1 \leq i \leq c} \deg(v_i)$ to be taken over all edges, we obtain $\sum_{e \in E} \sum_{C_c \text{ assigned to } e} \min_{1 \leq i \leq c} \deg(v_i)$. For each edge e , the quantity inside the inner sum is upper bounded by the minimum of the degrees of the endpoints of e , so we have $\sum_{e=(u,v) \in E} \min(\deg(u), \deg(v)) \cdot (\# \text{ of } C_c \text{ assigned to } e)$.

We now claim that the number of c -cliques C_c assigned to each edge $e = (u, v)$ is upper bounded by $O(\alpha^{c-2})$. This fact follows by virtue of the $O(\alpha)$ -orientation used earlier to assign c -cliques C_c to edges. Namely, if we let DG denote the directed graph given by the $O(\alpha)$ -orientation, then by construction for each C_c , vertices $v_i \in C_c$ for $i > 2$ must be out-neighbors of v_1 and v_2 . There are at most $O(\alpha)$ out-neighbors in the intersection of $N_{DG}(u)$ and $N_{DG}(v)$ that could potentially complete a directed c -clique with the vertices u and v . With $c - 2$ additional vertices necessary to complete a directed c -clique, and $O(\alpha)$ vertices to choose from, there are at most $O(\alpha^{c-2})$ such directed c -cliques.

Thus, our desired sum is given by $\sum_{e=(u,v) \in E} \min(\deg(u), \deg(v)) \cdot (\# \text{ of } C_c \text{ assigned to } e) = O(\alpha^{c-2} \sum_{e=(u,v) \in E} \min(\deg(u), \deg(v)))$. By Lemma 2 of [72], we know that $\sum_{e=(u,v) \in E} \min(\deg(u), \deg(v)) = O(m\alpha)$. Therefore, in total, we have $\sum_{C_c} \min_{1 \leq i \leq c} \deg(v_i) = O(m\alpha^{c-1})$, as desired. \square

Using this key lemma, we prove the following complexity bounds for our parallel nucleus decomposition algorithm. $\rho_{(r,s)}(G)$ is defined to be the **(r, s) peeling complexity** of G , or the number of rounds needed to peel the graph where in each round,

all r -cliques with the minimum s -clique count are peeled. $\rho_{(r,s)}(G) \leq O(m\alpha^{r-2})$, since at least one r -clique is peeled in each round.

Theorem 10.1. ARB-NUCLEUS computes the (r, s) nucleus decomposition in $O(m\alpha^{s-2} + \rho_{(r,s)}(G) \log n)$ amortized expected work and $O(\rho_{(r,s)}(G) \log n + \log^2 n)$ span w.h.p., where $\rho_{(r,s)}(G)$ is the (r, s) peeling complexity of G .

Proof. First, the work and span of counting the number of s -cliques per r -clique is given directly by Shi *et al.* [296], since we use this s -clique enumeration algorithm, REC-LIST-CLIQUEs, as a subroutine. Because we hash each s -clique count per r -clique in parallel, this takes $O(m\alpha^{s-2})$ work and $O(\log^2 n)$ span w.h.p. for a constant s .

We now discuss the work and span of obtaining the set of r -cliques with minimum s -clique count and updating the s -clique counts in our bucketing structure B . We make use of the fact that the total number of c -cliques in G is bounded by $O(m\alpha^{c-2})$, which follows from the c -clique enumeration algorithm [296]. The overall work of inserting r -cliques into B is given by the number of r -cliques in G , or $O(m\alpha^{r-2})$. Each r -clique has its bucket decremented at most once per incident s -clique, and because there are at most $O(m\alpha^{s-2})$ s -cliques, the work of updating buckets is given by $O(m\alpha^{s-2})$. Finally, extracting the minimum bucket can be done in $O(\log n)$ amortized expected work and $O(\log n)$ span w.h.p., which in total gives $O(\rho_{(r,s)}(G) \log n)$ amortized expected work, and $O(\rho_{(r,s)}(G) \log n)$ span w.h.p.

Finally, it remains to discuss the work and span of obtaining updated s -clique counts after peeling each set of r -cliques, or the work and span of the UPDATE subroutine. For each r -clique R , UPDATE first computes the intersection of the neighbors of each vertex $v \in R$, and stores them in set I . Notably, the total work of intersecting the neighbors of each vertex $v \in R$ over all r -cliques R is given by $O(\sum_R \min_{1 \leq i \leq r} \deg(v_i)) = O(m\alpha^{r-1})$ w.h.p., which follows from Lemma 10.1, and the span across all intersections is $O(\log n)$ w.h.p. for a constant r .

UPDATE then calls the REC-LIST-CLIQUEs subroutine to complete s -cliques from R , taking as input the sets I computed in the previous step. Each successive recursive call to REC-LIST-CLIQUEs takes a multiplicative $O(\alpha)$ work w.h.p. due to the intersection operation, and REC-LIST-CLIQUEs requires $s - r$ recursive levels to complete each s -clique. This results in a total multiplicative factor of $O(\alpha^{s-r-1})$ work w.h.p., because the final recursive level (as shown in the $r\ell = 1$ case in REC-LIST-CLIQUEs) does not involve any intersection operations, and simply iterates through the discovered s -cliques. Thus, including the initial cost of setting up the call to REC-LIST-CLIQUEs, and using the fact that the number of potential neighbors in the sets I across all r -cliques is $O(m\alpha^{r-1})$, the total work of REC-LIST-CLIQUEs across all r -cliques is given by $O(m\alpha^{r-1} \cdot \alpha^{s-r-1}) = O(m\alpha^{s-2})$ w.h.p. The span of each intersection on each recursive level is $O(\log n)$ w.h.p., and there are $\rho_{(r,s)}(G)$ rounds by definition, which contributes $O(\rho_{(r,s)}(G) \log n)$ overall. \square

10.4.3 Discussion of Theoretically Efficient Bounds

We discuss in this section the proof that the complexity bounds for our (r, s) nucleus decomposition algorithm, ARB-NUCLEUS, improve upon that in prior work. Specifi-

cally, ARB-NUCLEUS is work-efficient with respect to the best sequential algorithm, and improves upon the best sequential algorithm that uses sublinear space in the number of s -cliques. This is because the previous best sequential bounds were given by Sariyüce *et al.* [284], in terms of the number of c -cliques containing each vertex v , or $ct_c(v)$, and the work of an arbitrary c -clique enumeration algorithm, or RT_c . Assuming space proportional to the number of s -cliques and the number of r -cliques in the graph, they compute the (r, s) nucleus decomposition in $O(RT_r + RT_s) = O(RT_s)$ work, and assuming space proportional to only the number of r -cliques in the graph, they compute the (r, s) nucleus decomposition in $O(RT_r + \sum_{v \in V(G)} ct_r(v) \cdot \deg(v)^{s-r})$ work. ARB-NUCLEUS uses space proportional to the number of r -cliques due to the space required for T and B , and is work-efficient with respect to the best sequential algorithm that uses the same space, improving upon the corresponding bound given by Sariyüce *et al.* [284]. If more space is permitted, a small modification of our algorithm yields a sequential nucleus decomposition algorithm that performs $O(m\alpha^{s-2})$ work, matching the corresponding bound given by Sariyüce *et al.* [284]. The idea is to use a dense bucketing structure with space proportional to the total number of s -cliques, since finding the next bucket with the minimum s -clique count involves a simple linear search rather than a heap operation. In the parallel setting, ARB-NUCLEUS can work-efficiently find the minimum non-empty bucket by performing a series of steps, where at each step i , ARB-NUCLEUS searches in parallel the region $[2^i, 2^{i+1}]$ for the next non-empty bucket. This search procedure takes logarithmic span, and is work-efficient with respect to the sequential algorithm.

In more detail, we directly compare here our bound to those given by Sariyüce *et al.* [284] for their sequential (r, s) nucleus decomposition algorithms. Assuming a work-efficient c -clique listing algorithm, specifically Shi *et al.*'s work-efficient c -clique listing algorithm [296], we see that Sariyüce *et al.*'s bounds are $O(m\alpha^{s-2})$ and $O(m\alpha^{r-2} + \sum_{v \in V(G)} ct_r(v) \cdot \deg(v)^{s-r})$ assuming space proportional to the number of s -cliques and r -cliques and space proportional to only the number of r -cliques, respectively.

We note that Sariyüce *et al.* do not include the work required to retrieve the r -clique with the minimum s -clique count in their complexity bound. Importantly, the time complexity of this step is non-negligible in the theoretical bounds of (r, s) nucleus decomposition. There are two possibilities depending on space usage. If space proportional to the number of s -cliques is allowed, it is possible to use an array proportional to the maximum number of s -cliques per vertex to allow for efficient retrieval of the r -clique with the minimum s -clique count in any given round, which would take $O(n_s)$ work in total. However, if the space is limited to the number of r -cliques, a space-efficient heap must be used, which would add an additional $O(n_r \log n)$ term to the work.

Adding the time complexity of retrieving the r -clique with the minimum s -clique count, assuming space proportional to the number of s -cliques and r -cliques, Sariyüce *et al.*'s algorithm takes $O(m\alpha^{s-2} + n_s) = O(m\alpha^{s-2})$ work, and assuming space proportional to only the number of r -cliques, Sariyüce *et al.*'s algorithm takes $O(m\alpha^{r-2} + \sum_{v \in V(G)} ct_r(v) \cdot \deg(v)^{s-r} + n_r \log n)$ work.

Space proportional to the number of s -cliques and r -cliques. We first compare to Sariyüce *et al.*'s work considering space proportional to the number of s -cliques and r -cliques.

We note that assuming space proportional to the number of s -cliques, we can replace the $O(\rho_{(r,s)}(G) \log n)$ term in ARB-NUCLEUS's work bound with $O(n_s)$ while maintaining the same span by replacing the batch-parallel Fibonacci heap [298] with an array of size proportional to the total number of s -cliques (to store and retrieve the r -cliques with the minimum s -clique count in any given round).

In more detail, let our bucketing structure be represented by an array of size $x = O(\text{poly}(y))$, where each array cell represents a bucket, and there are y elements in the bucket structure. Then, we can repeatedly pop the minimum bucket until the structure is empty using $O(x)$ total work and $O(\rho \log y)$ span, assuming it takes $O(\rho)$ rounds to empty the structure. This is done by splitting the array into regions $r_i = [2^i, 2^{i+1}]$ for $i = [0, \log x]$. Let us denote the number of buckets in each region r_i by $|r_i|$. We start at the first region r_0 and search in parallel for the first non-empty bucket, using a parallel reduce. We return the first non-empty bucket if it exists, and otherwise, we repeat with the next region r_1 , and so on. We note that we will never revisit the previously searched region r_0 once it has been found to be empty. We repeat the entire process of splitting the array into regions and searching each region in order for each subsequent pop query, starting from the previously popped bucket. In this manner, we maintain $O(\log y)$ span to pop each non-empty bucket, while incurring $O(x)$ total work.

Thus, allowing space proportional to the number of s -cliques, ARB-NUCLEUS is work-efficient, taking $O(m\alpha^{s-2} + n_s) = O(m\alpha^{s-2})$ work.

Space proportional to the number of r -cliques. We now discuss Sariyüce *et al.*'s work considering space proportional to only the number of r -cliques. We claim that ARB-NUCLEUS improves upon Sariyüce *et al.*'s algorithm.

In order to show this claim, we discuss the work of each step of ARB-NUCLEUS in more detail, as we do in the proof of Theorem 10.1.⁴ First, we consider the work of inserting r -cliques into our bucketing structure B , which takes $O(m\alpha^{r-2})$ work. Then, because each r -clique has its bucket decremented at most once per incident s -clique, we incur work proportional to the total number of s -cliques, or $O(ct_s(v))$. Extracting the minimum bucket takes $O(\rho_{(r,s)}(G) \log n)$ amortized expected work using the batch-parallel Fibonacci heap.

Finally, it remains to discuss the work of obtaining updated s -clique counts after peeling each set of r -cliques, in the UPDATE subroutine. As discussed in the proof of Theorem 10.1, it takes $O(\sum_R \min_{1 \leq i \leq r} \deg(v_i))$ work w.h.p. to intersect the neighbors of each vertex $v \in R$ over all r -cliques R . ARB-NUCLEUS then considers the set of vertices in this intersection I , and performs successive intersection operations with the oriented neighbors of each vertex in I . These intersection operations are bounded by the arboricity $O(\alpha)$, but they are also bounded by $O(\min_{1 \leq i \leq r} \deg(v_i))$

⁴We provide a closed form work bound for ARB-NUCLEUS, whereas Sariyüce *et al.*'s bound is not closed. As a result, directly comparing our closed form bound to Sariyüce *et al.*'s bound is difficult; instead, we provide here a more detailed comparison of the exact work required by ARB-NUCLEUS with Sariyüce *et al.*'s work bound.

as the maximum size of I . Thus, this step takes $O(\sum_R \min_{1 \leq i \leq r} \deg(v_i)^{s-r}) = O(\sum_{v \in V(G)} ct_r(v) \cdot \deg(v)^{s-r})$ work.

In total, ARB-NUCLEUS incurs as an upper bound $O(m\alpha^{r-2} + ct_s(v) + \sum_{v \in V(G)} ct_r(v) \cdot \deg(v)^{s-r} + \rho_{(r,s)}(G) \log n) = O(m\alpha^{r-2} + \sum_{v \in V(G)} ct_r(v) \cdot \deg(v)^{s-r} + \rho_{(r,s)}(G) \log n)$ work, and thus does not exceed Sariyüce *et al.*'s work bound.

Note that ARB-NUCLEUS improves upon Sariyüce *et al.*'s work in the UPDATE subroutine. Notably, Sariyüce *et al.*'s work assumes that for each vertex v participating in an r -clique, the work incurred is $O(\deg(v)^{s-r})$. However, our UPDATE subroutine considers only the minimum vertex degree of each r -clique. Moreover, after finding the set I of candidate vertices to extend each r -clique into an s -clique, ARB-NUCLEUS uses the $O(\alpha)$ -oriented neighbors of these candidate vertices in the intersection subroutine. The actual work incurred in the intersection is thus the minimum of the size of I and of the arboricity oriented out-degrees of the vertices in I , which further improves upon the $O(\deg(v)^{s-r})$ bound in Sariyüce *et al.*'s work. This improvement is particularly evident if there are a few vertices of high degree involved in many r - and s -cliques, in which the use of an $O(\alpha)$ -orientation significantly reduces the number of out-neighbors that must be traversed. We additionally note that $O(\alpha)$ tightly bounds the maximum out-degree in any acyclic orientation of a graph G ,⁵ and in this sense, our closed-form work bound for ARB-NUCLEUS is never asymptotically worse than the work bound using any other acyclic orientation.

10.5 Practical Optimizations

We now introduce the practical optimizations that we use for our parallel (r, s) nucleus decomposition implementation.

10.5.1 Number of Parallel Hash Table Levels

ARB-NUCLEUS uses a single parallel hash table T to store the s -clique counts, where the keys are r -cliques. However, this storage method is infeasible in practice due to space limitations, particularly for large r , since r vertices must be concatenated into a key for each r -clique. We observe that space can be saved by introducing more levels to our parallel hash table T . For example, one option is to instead use a two-level combination of an array and a parallel hash table, which consists of an array of size n whose elements are pointers to individual hash tables where the keys are $(r - 1)$ -cliques. The s -clique count for a corresponding r -clique $R = \{v_1, \dots, v_r\}$ (where the vertices are in sorted order) is stored by indexing into the v_1^{th} element of the array, and storing the count on the key corresponding to the $(r - 1)$ -clique given by $\{v_2, \dots, v_r\}$ in the given hash table. Space savings arise because v_1 does not have to be repeatedly stored for each $(r - 1)$ -clique in its corresponding hash table, but

⁵This follows because if we let d denote the degeneracy of the graph G , then the arboricity α tightly bounds the degeneracy in that $\alpha \leq d \leq 2\alpha - 1$. Moreover, d is the degeneracy of G if and only if G can be acyclically directed such that the maximum out-degree in the directed graph is d [74].

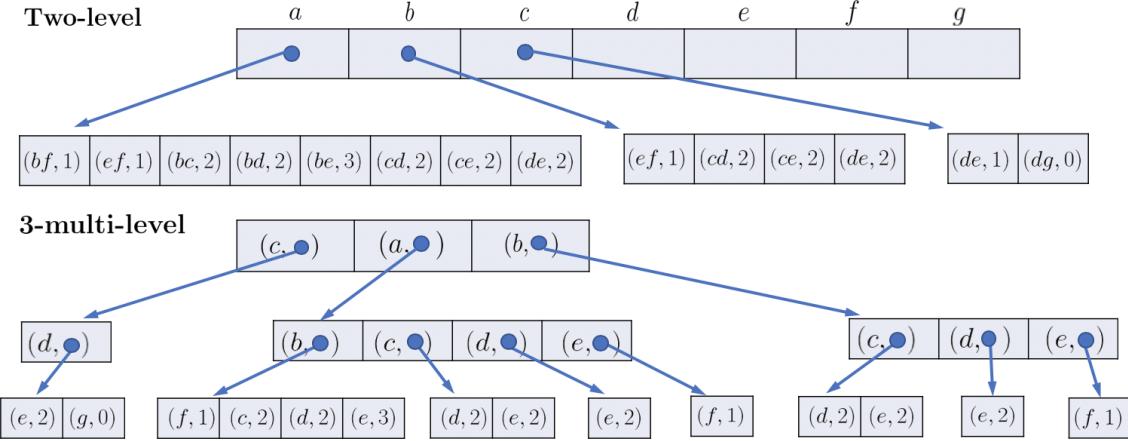


Figure 10.3: An example of the initial parallel hash table T for $(3, 4)$ nucleus decomposition on the graph from Figure 10.1, considering different numbers of levels. Note that Figure 10.2 shows a one-level parallel hash table T . If we consider each vertex and each pointer to take a unit of memory, the one-level T takes 42 units, while the two-level T takes 35 units, thus saving memory. However, the 3-multi-level T takes 50 units, because $r = 3$ is too small to give memory savings for this graph.

rather is only stored once in the array.

A more general option, particularly for large r , is to use a multi-level parallel hash table, with $\ell \leq r$ levels in which each intermediate level consists of a parallel hash table keyed by a single vertex in the r -clique whose value is a pointer to a parallel hash table in the subsequent level, and the last level consists of a parallel hash table keyed by $(r - \ell + 1)$ -cliques. Given an r -clique $R = \{v_1, \dots, v_r\}$ (where the vertices are in sorted order), each of the vertices in the clique in order is mapped to each level of the hash table, except for the last $(r - \ell + 1)$ vertices which are concatenated into a key for the last level of the hash table. Thus, the location in the hash table corresponding to R can be found by looking up the key v_j in the hash table at each level for $j < \ell$, and following the pointer to the next level. At the last level, the key is given by the $(r - \ell + 1)$ -clique corresponding to $\{v_\ell, \dots, v_r\}$. Again, space savings arise due to the shared vertices on each intermediate level, which need not be repeatedly stored in the keys on the subsequent levels of the parallel hash table, and for $r \geq \ell > 2$, these savings may exceed those found in the previous combination of an array and a parallel hash table.

In considering different numbers of levels, we differentiate between the *two-level* combination of an array and a parallel hash table, and the ℓ -*multi-level* option of nested parallel hash tables, where ℓ is the number of levels, and notably, we may have $\ell = 2$. Figures 10.3 and 10.4 show examples of T using different numbers of levels. As shown in these examples, there are cases where increasing the number of levels does not save space, because the additional pointers required in increasing the number of levels exceeds the overlap in vertices between r -cliques, particularly for small r .

Moreover, note that there are no theoretical guarantees on whether additional

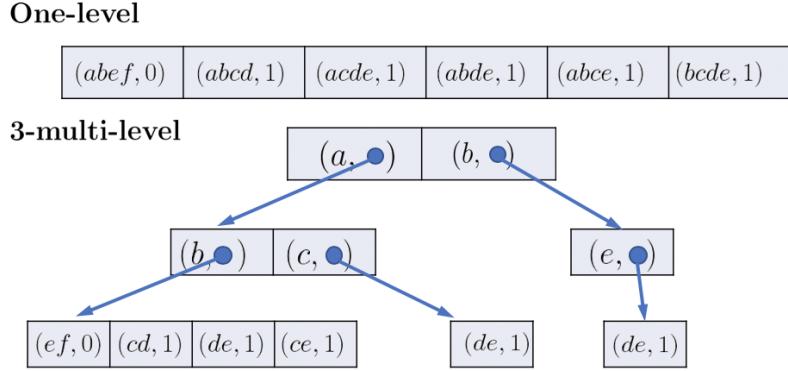


Figure 10.4: An example of the initial parallel hash table T for $(4, 5)$ nucleus decomposition on the graph from Figure 10.1, considering different numbers of levels. If we consider each vertex and each pointer to take a unit of memory, the one-level T takes 24 units, while the 3-multi-level T takes 22 units, thus saving memory. We see memory savings with more levels in T compared to in Figure 10.3, because $r = 4$ is sufficiently large.

levels will result in a memory reduction, since a memory reduction depends on the number of overlapping r -cliques and the size of the overlap in r -cliques in any given graph. In other words, the skew in the constituent vertices of the r -cliques directly impacts the possible memory reduction. Similarly, performance improvements may arise due to better cache locality in accessing r -cliques that share many vertices, but this is not guaranteed, and cache misses are inevitable while traversing through different levels.

The idea of a multi-level parallel hash table is more generally applicable in scenarios where the efficient storage and access of sets with significant overlap is desired, particularly if memory usage is a bottleneck. An example use case is to efficiently store the hyperedge adjacency lists for a hypergraph. The multi-level parallel hash table can also be used for data with multiple fields or dimensions, where each level keys a different field.

10.5.2 Contiguous Space

In using the two-level and multilevel data structures as T , a natural way to implement the last level parallel hash tables in these more complicated data structures is to simply allocate separate blocks of memory as needed for each last level parallel hash table. However, this approach may be memory-inefficient and lead to poor cache locality. An optimization for the two-level and multi-level tables is to instead first compute the space needed for all last level parallel hash tables and then allocate a contiguous block of memory such that the last level tables are placed consecutively with one another, using a parallel prefix sum to determine each of their indices into the contiguous block of memory. This optimization requires little overhead, and has the additional benefit of greater cache locality. This optimization does not apply to

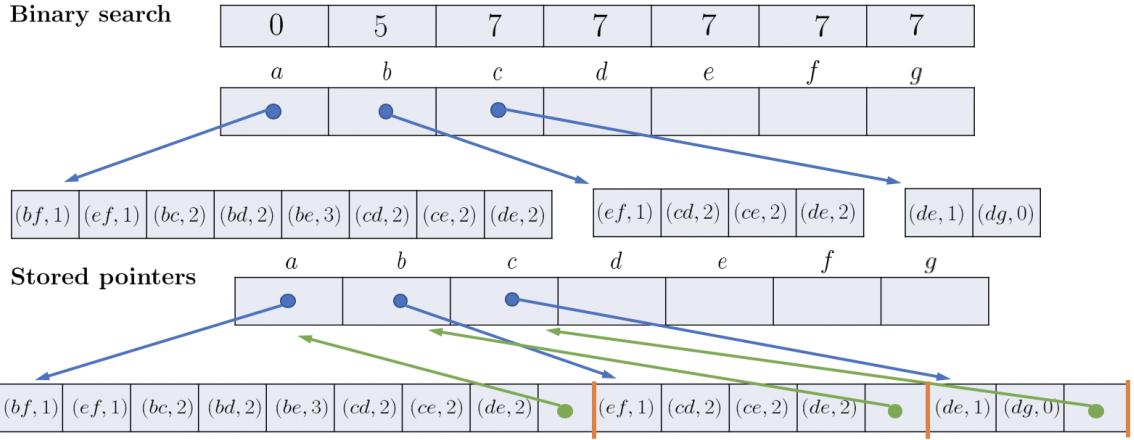


Figure 10.5: Using the same graph as shown in Figure 10.1 and the two-level T from Figure 10.3, for $(3,4)$ nucleus decomposition, an example of the binary search and stored pointers methods.

one-level parallel hash tables, since they are by nature contiguous.

10.5.3 Binary Search vs. Stored Pointers

An important implementation detail from ARB-NUCLEUS is the interfacing between the representation of r -cliques in the bucketing structure B and the representation of r -cliques in the parallel hash table T . It is impractical to use the theoretically-efficient Fibonacci heap used to obtain our theoretical bounds, due to the complexity of Fibonacci heaps in general; in practice, we use the efficient parallel bucketing implementation by Dhulipala et al. [95].

This bucketing structure requires each r -clique to be represented by an index, and to interface between T and B , we require a map translating each r -clique in T to its index in B , and vice versa. More explicitly, for a given r -clique in T , we must be able to find the number of s -cliques it participates in using B , and symmetrically, for a given r -clique that we peel from B , we must be able to find its constituent vertices using T . It would be impractical in terms of space to represent the r -clique directly using its constituent vertices in B , since its constituent vertices are already stored in T . Thus, we seek a unique index to represent each r -clique in B , that is easy to map to and from the r -clique in T . If T is represented using a one-level parallel hash table, then a natural index to use for each r -clique is its key in T , and the map is implicitly the identity map. For instance, considering the one-level T in Figure 10.2, instead of storing cde in bucket 0 before any rounds begin, we would instead store 13, which is the index of cde in T . However, if T is represented using a two-level or multi-level structure, then the index representation and map are less natural, and require additional overheads to maintain.

One important property of the mapping from T to B is space-efficiency, and so it is desirable to maintain an implicit map. Therefore, we represent each r -clique by the unique index corresponding to its position in the last level parallel hash table in

Simple array

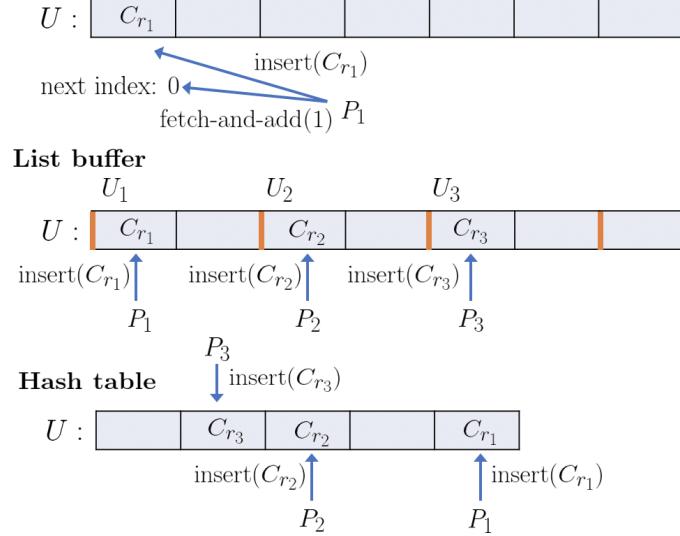


Figure 10.6: An example of the simple array, list buffer, and hash table options in aggregating the set U of r -cliques with updated s -clique counts. Processors P_1 , P_2 , and P_3 are storing r -cliques C_{r_1} , C_{r_2} , and C_{r_3} , respectively, in U .

T , which can be implicitly obtained by its location in memory if T is represented contiguously. If T is not represented contiguously, then we can store the prefix sums of the sizes of each successive level of parallel hash tables, and use these plus the r -clique's index at its last level table in T to obtain the unique index. For equivalent T , the index corresponding to each r -clique is the same regardless of whether T is represented contiguously in memory or not. For instance, considering the two-level T in Figure 10.3, instead of storing abc in bucket 2 before any rounds begin, we would instead store 2, which is the index of abc in the second level of T . Similarly, instead of storing bef in bucket 1, we would instead store 8; this is because we take the index corresponding to bef to be its index in the second level hash table corresponding to vertex b , plus the sizes of earlier second level hash tables, namely the size of the hash table corresponding to vertex a . Then, it is also necessary to implicitly maintain the inverse map, from an index in B to the constituent vertices of the corresponding r -clique, for which we provide two methods.

Binary Search Method. One solution is to store the prefix sums of the sizes of each successive level of parallel hash tables in both the contiguous and non-contiguous cases, and use a serial binary search to find the table corresponding to the given index at each level. We also store the vertex corresponding to each intermediate level alongside each table, which can be easily accessed after the binary search. The key at the last level table can then be translated to its constituent $r - \ell$ vertices. This does not require the last level parallel hash tables to be stored contiguously in memory. Figure 10.5 shows an example of this method in the non-contiguous case, where the top array is the prefix sum of the sizes of the second level hash tables. Each entry

in the prefix sum corresponds to an entry in the first level, which contains the first vertex of the triangle and points to the second level, which contains the other two vertices.

Stored Pointer Method. However, while using binary searches is a natural solution, they may be computationally inefficient, both in terms of work and cache misses, considering the number of such translations that are needed throughout ARB-NUCLEUS. Instead, we consider a more sophisticated solution which, assuming contiguous memory, is to place barriers between parallel hash tables on each level that contain up-pointers to prior levels, and fill empty cells of the parallel hash tables with the same up-pointers. Then, given an index to the last level, which translates directly into the memory location of the cell containing the last level r -clique key due to the contiguous memory, we can perform linear searches to the right until it reaches an empty cell, which directly gives an up-pointer to the prior level (and thus, also the vertex corresponding to the prior level). We differentiate between empty and non-empty cells by reserving the top bit of each key to indicate whether the cell is empty or non-empty. The benefit of this method is that with a good hashing scheme, the linear search for an empty cell will be faster and more cache-friendly compared to a full binary search. Figure 10.5 also shows an example of this method using contiguous space, in which the barriers placed to the right of each hash table in the second level point back to the parent entry in the first level, allowing an index to traverse up these pointers to obtain the corresponding first vertex of the triangle. Note that the bold orange lines on the second level mark the boundaries between parallel hash tables corresponding to different vertices from the first level.

With both the contiguous space and stored pointers optimizations applied to two-level and multi-level parallel hash tables T , we show in Section 10.6.2 that we achieve up to a 1.32x speedup of using a two-level T , and up to a 1.46x speedup of using an ℓ -multi-level T for $\ell > 2$, over one level.

10.5.4 Graph Relabeling

We sort the vertices in each r -clique prior to translating it into a unique key for use in the parallel hash table T . However, the lexicographic ordering of vertices in the input graph may not be representative of the access patterns used in finding r -cliques. In particular, because we use directed edges based on an $O(\alpha)$ -orientation to count r -cliques and s -cliques, an optimization that could save work and improve cache locality in accessing T is to relabel vertices based on the $O(\alpha)$ -orientation, so that the sorted order is representative of the order in which vertices are discovered in the REC-LIST-CLIQUE subroutine. The first benefit of this optimization is that there is no need to re-sort r -cliques based on the orientation, which is implicitly performed on Line 4 of Algorithm 10.2, as after relabeling, the vertices in a clique will be added in increasing order. A second benefit is that r -cliques discovered in the same recursive hierarchy will be located closer together in our parallel hash table, potentially leading to better cache locality when accessing their counts in T . In our evaluation in Section 10.6.2, we see up to a 1.29x speedup using graph relabeling.

10.5.5 Obtaining the Set of Updated r -cliques

A key component of the bucketing structure in ARB-NUCLEUS is the computation of the set U of r -cliques with updated s -clique counts, after peeling a set of r -cliques. Using a parallel hash table with size equal to the number of r -cliques is slow in practice due to the need to iterate through the entire hash table to retrieve the r -cliques and to clear the hash table. We present here three options for computing U . Figure 10.6 shows an example of each of these.

Simple Array. The first option is to represent U as a simple array to hold r -cliques, along with a variable that maintains the next open slot in the array. We use a fetch-and-add to update the s -clique count of a discovered r -clique in the UPDATE-FUNC subroutine, and if in the current round this is the first such modification of the r -clique's count, we use a fetch-and-add to reserve a slot in U , and store the r -clique in the reserved slot in U . This method introduces contention due to the requirement of all updated r -cliques to perform a fetch-and-add with a single variable to reserve a slot in U . As shown in Figure 10.6, processor P_1 successfully updates the index variable and inserts its r -clique C_{r_1} into the first index in U , but the other processors must wait until their fetch-and-add operations succeed. However, the r -cliques are compactly stored in U and there is no need to clear elements in U , which results in time savings.

List Buffer. The second option improves upon the contention incurred by the first option, offering better performance. We use a data structure that we call a *list buffer*, which consists of an array U that holds r -cliques, and P variables $\{i_1, \dots, i_P\}$ that maintain the next open slots in the array, where P is the number of threads. Each of the P variables is exclusively assigned to one of the P threads. The data structure also uses a constant buffer size, and each thread is initially assigned a contiguous block of U of size equal to this constant buffer size. When a thread j is the first to modify an r -clique's count in a given round, the thread updates its corresponding i_j , and uses the reserved open slot in U to store the r -clique. If a thread runs out of space in its assigned block in U , it uses a fetch-and-add operation to reserve the next available block in U of size equal to the constant buffer size. We filter U of all unused slots, prior to returning U to the bucketing data structure. The reduced contention is due to the exclusive i_j that each thread j can update without contention. Threads may still contend on reserving new blocks of space in U , but the contention is minimal with a large enough buffer size. In reusing the list buffer data structure in later rounds, there is no need to clear U , as it is sufficient to reset the i_j . In Figure 10.6, the buffer size is 2, and the slots in U assigned to each processor P_i are labeled by U_i . Each P_i can store its corresponding r -clique C_{r_i} in parallel, since there is no contention within their buffer as long as the buffer is not full.

Hash Table. The last option reduces potential contention even further by removing the necessity to reserve space, but at the cost of additional work needed to clear U between rounds. This option uses a parallel hash table as U , but dynamically determines the amount of space required on each round based on the number of r -cliques peeled, and as such, the maximum possible number of r -cliques with updated s -clique counts. Thus, in rounds with fewer r -cliques peeled, less space is reserved for

U for that round, and as a result, less work is required to clear the space to reuse in the next round. Figure 10.6 shows each processor P_i storing its corresponding r -clique C_{r_i} into the parallel hash table. However, the entirety of U must be cleared in order to reuse U between rounds.

In our evaluation in Section 10.6.2, we achieve up to a 3.98x speedup using a list buffer and up to a 4.12x speedup using a parallel hash table, both over a simple array.

10.5.6 Graph Contraction

In the special case of $(2, 3)$ nucleus (truss) decomposition, we introduce an optimization that filters out peeled edges when a significant number of edges have been peeled over successive rounds. When many edges have been peeled, reducing the overall size of the graph and contracting adjacency lists may save work in future rounds, in that work does not need to be spent iterating over previously peeled edges, allowing for performance improvements.

We perform this contraction when the number of peeled edges since the previous contraction is at least twice the number of vertices in the graph, and we only contract adjacency lists of vertices that have lost at least a quarter of their neighbors since the previous contraction. We chose these boundaries for contraction heuristically based on performance on real-world graphs; these generally allow for contraction to be performed only in instances in which it could meaningfully impact the performance of future operations. Specifically, we found that the overhead of each contraction operation is balanced by reduced work in future iterations only when vertices can significantly reduce the size of their adjacency lists, and iterating through vertices to check for this condition is only worthwhile when sufficiently many edges have been peeled. Note that the k -truss decomposition implementation by Che *et al.* [70] also periodically contracts the graph. In terms of implementation details, the contraction operations are performed in parallel for each vertex. If a vertex meets the heuristic criteria, its adjacency list is filtered into a new adjacency list sans the peeled edges, using the parallel filter primitive; this new adjacency list replaces the previous adjacency list. This optimization does not extend to higher (r, s) nucleus decomposition, because if an r -clique has been peeled for $r > 2$, its edges may still be involved in other unpeeled r -cliques. Thus, there is no natural way to contract out an r -clique from a graph for $r > 2$ to allow for work savings in future rounds. In Section 10.6.2, we see up to a 1.08x speedup using graph contraction in $(2, 3)$ nucleus decomposition.

10.6 Evaluation

10.6.1 Environment and Graph Inputs

We run experiments on a Google Cloud Platform instance of a 30-core machine with two-way hyper-threading, with 3.8 GHz Intel Xeon Scalable (Cascade Lake) processors and 240 GiB of main memory. We compile with g++ (version 7.4.0) and use the `-O3` flag. We use the work-stealing scheduler PARLAYLIB by Blelloch *et al.* [46].

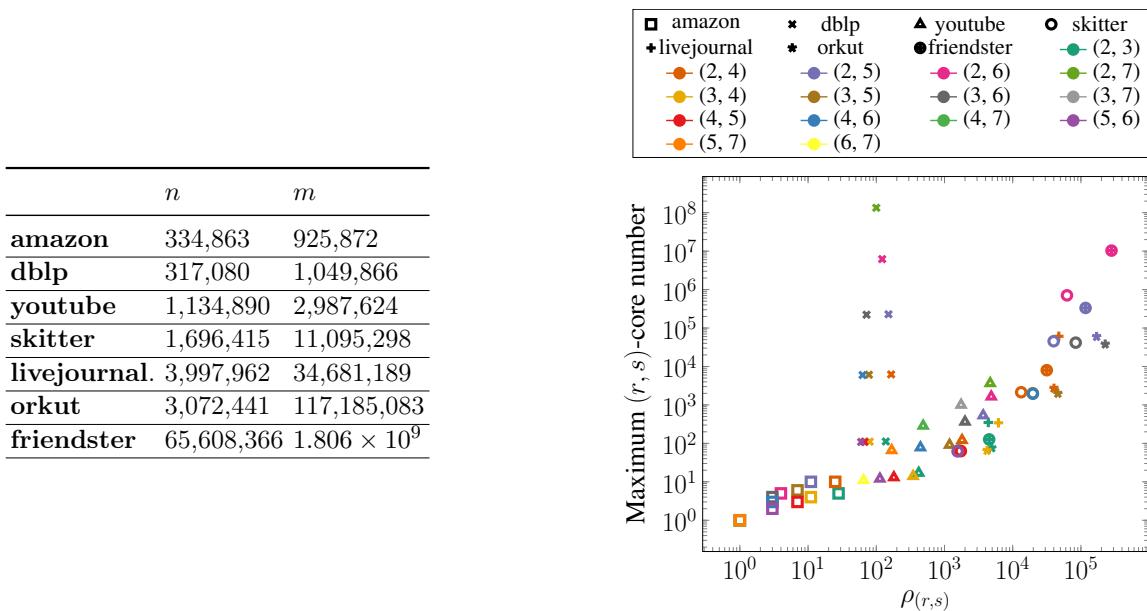


Figure 10.7: Sizes of our input graphs, all of which are from [207], on the left, and the number of rounds required $\rho_{(r,s)}$ and the maximum (r, s) -core numbers for $r < s \leq 7$ for each graph on the right.

We terminate any experiment that takes over 6 hours. We test our algorithms on real-world graphs from the Stanford Network Analysis Project (SNAP) [207], shown in Figure 10.7 with $\rho_{(r,s)}$ and the maximum (r, s) -core numbers for $r < s \leq 7$. We also use synthetic rMAT graphs [64], with $a = 0.5$, $b = c = 0.1$, and $d = 0.3$.

We compare to Sariyüce *et al.*'s state-of-the-art parallel [283] and serial [284] nucleus decomposition implementations, which address $(2, 3)$ and $(3, 4)$ nucleus decomposition. For the special case of $(2, 3)$ nucleus decomposition, we compare to Che *et al.*'s [70] highly optimized state-of-the-art parallel PKT-OPT-CPU, and implementations by Kabir and Madduri's PKT [186], Smith *et al.*'s MSP [305], and Blanco *et al.* [45], which represent the top-performing implementations from the MIT GraphChallenge [276]. We run all of these using the same environment as our experiments on ARB-NUCLEUS.

10.6.2 Tuning Optimizations

There are six total optimizations that we implement in ARB-NUCLEUS: different numbers of levels in our parallel hash table T (*numbers of levels*), the use of contiguous space (*contiguous space*), a binary search versus stored pointers to perform the mapping of indices representing r -cliques to the constituent vertices (*inverse index map*), graph relabeling (*graph relabeling*), a simple array method versus a list buffer versus a parallel hash table for maintaining the set U of r -cliques with changed s -clique counts per round (*update aggregation*), and graph contraction specifically for $(2, 3)$ nucleus decomposition (*graph contraction*).

The most unoptimized form of ARB-NUCLEUS uses a one-level parallel hash table

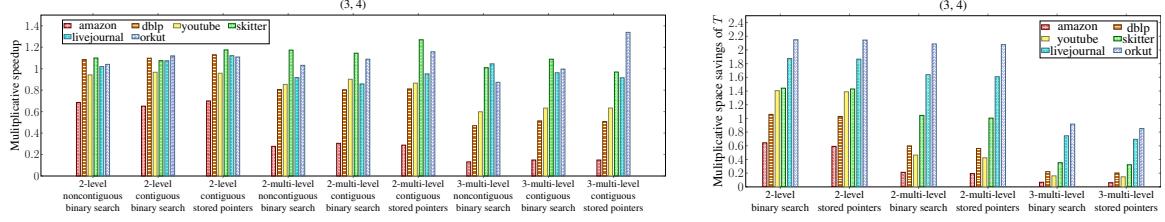


Figure 10.8: On the left, multiplicative speedups of different combinations of optimizations on T in ARB-NUCLEUS, over an unoptimized setting of ARB-NUCLEUS, for $(3, 4)$ nucleus decomposition. On the right, multiplicative space savings for T of different combinations of optimizations on T in ARB-NUCLEUS, over the unoptimized setting, for $(3, 4)$ nucleus decomposition. Note that the space usage between the non-contiguous option and the contiguous option is equal. Friendster is omitted because ARB-NUCLEUS runs out of memory for this graph.

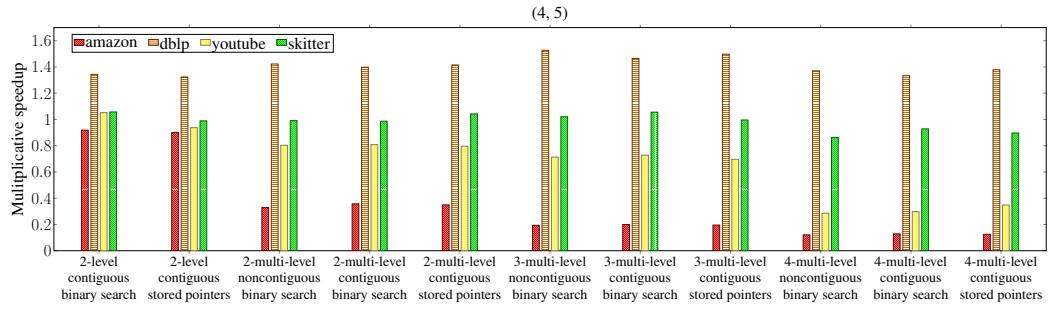


Figure 10.9: Multiplicative speedups of different combinations of optimizations on T in ARB-NUCLEUS, over an unoptimized setting of ARB-NUCLEUS, for $(4, 5)$ nucleus decomposition. Livejournal, orkut, and friendster are omitted, because ARB-NUCLEUS runs out of memory for these instances.

T , no graph relabeling, and the simple array method for the update aggregation (as this is the simplest and most intuitive method), and in the case of $(2, 3)$ nucleus decomposition, no graph contraction. Note that the contiguous space and inverse index map optimizations do not apply to the one-level T .

Because the first three optimizations, namely numbers of levels, contiguous space, and the inverse index map, apply specifically to the parallel hash table T , while the remaining optimizations apply generally to the rest of the nucleus decomposition algorithm, we first tune different combinations of options for the first three optimizations against the unoptimized ARB-NUCLEUS. We then separately tune the remaining optimizations, namely graph relabeling, update aggregation, and graph contraction optimizations, against both the unoptimized ARB-NUCLEUS and the best choice of optimizations from the first comparison.

Optimizations on T . Across $(2, 3)$, $(2, 4)$, $(3, 4)$, and $(4, 5)$ nucleus decomposition, using a two-level T with contiguous space and stored pointers for the inverse index map gives the overall best configuration across these r and s values, with up to 1.32x speedups over the unoptimized case. This configuration either outperforms or offers

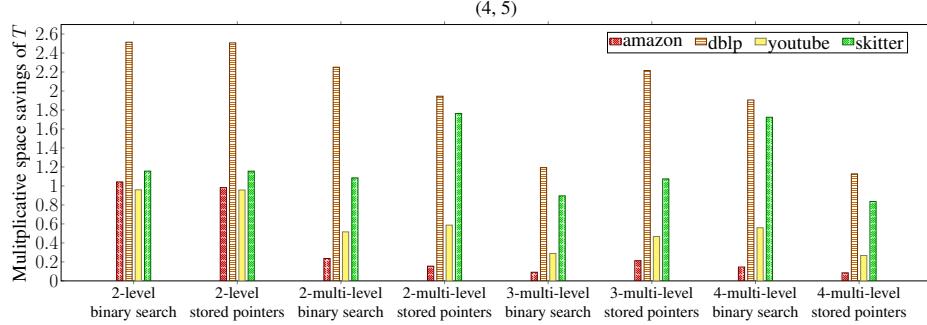


Figure 10.10: Multiplicative space savings for T of different combinations of optimizations on T in ARB-NUCLEUS, over the unoptimized setting, for $(4, 5)$ nucleus decomposition. Note that the space usage between the non-contiguous option and the contiguous option is equal. Also, livejournal, orkut, and friendster are omitted, because ARB-NUCLEUS runs out of memory for these instances.

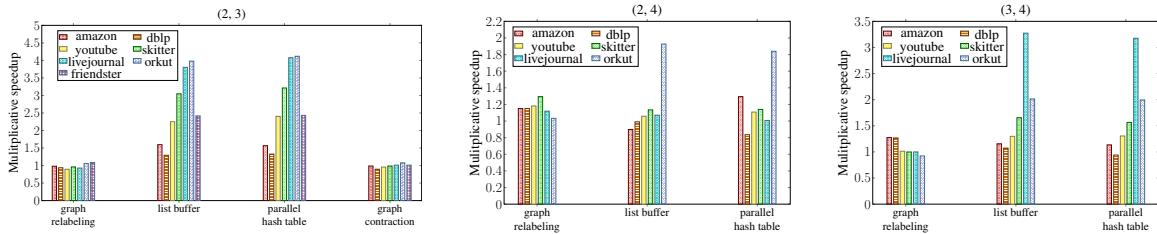


Figure 10.11: Multiplicative speedups of the graph relabeling, update aggregation, and graph contraction optimizations in ARB-NUCLEUS, over a two-level setting with contiguous space and stored pointers, and using the simple array for U . Friendster is omitted from the $(2, 4)$ and $(3, 4)$ nucleus decomposition experiments, because the unoptimized ARB-NUCLEUS times out for $(2, 4)$, and ARB-NUCLEUS runs out of memory for $(3, 4)$.

comparable performance to other configurations. Figure 10.8 and 10.9 shows the multiplicative speedup of each combination of optimizations on T over the unoptimized T for $(3, 4)$ nucleus decomposition and $(4, 5)$ nucleus decomposition, respectively; friendster is omitted from our $(3, 4)$ experiments because ARB-NUCLEUS runs out of memory on these graphs. Also, the speedups for $(2, 3)$ and $(2, 4)$ nucleus decomposition are omitted, but show similar behavior overall.

On the smallest graph, amazon, we see universally poor performance of the two-level and multi-level options, and the one-level T outperforms these options; however, we note that the one-level running times for ARB-NUCLEUS on amazon in these cases is < 0.2 seconds and the maximum core number of amazon is particularly small (≤ 10), and so the optimizations simply add too much overhead to see performance improvements. The main case where the two-level T performs noticeably worse than other options is for large graphs and for large r and s . In $(3, 4)$ nucleus decomposition, we note that for orkut, using a 3-multi-level T , also with contiguous space and stored pointers, significantly outperforms the analogous two-level option, with a

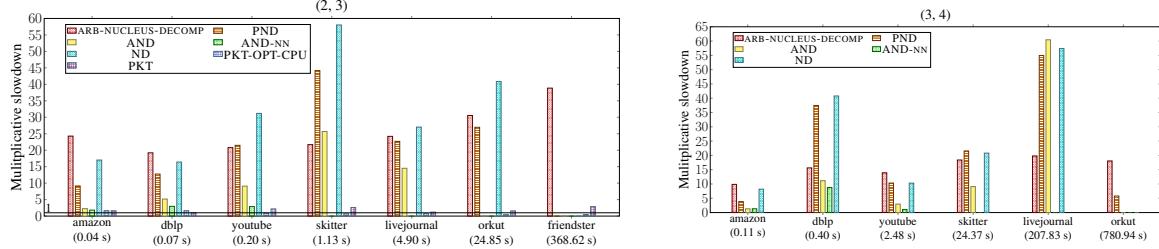


Figure 10.12: Multiplicative slowdowns over our parallel ARB-NUCLEUS of PKT-OPT-CPU and PKT for $(2, 3)$ nucleus decomposition, and of single-threaded ARB-NUCLEUS, PND, AND, AND-NN, and ND for $(2, 3)$ and $(3, 4)$ nucleus decomposition. The label ARB-NUCLEUS in the legend refers to our single-threaded running times. We have omitted bars for PND, AND, AND-NN, and ND where these implementations run out of memory or time out. We have included in parentheses the times of our parallel ARB-NUCLEUS on 30 cores with hyper-threading. We have also included a line marking a multiplicative slowdown of 1 for $r = 2$, $s = 3$, and we see that PKT-OPT-CPU outperforms ARB-NUCLEUS on skitter, livejournal, orkut, and friendster.

1.34x speedup over the unoptimized case compared to a 1.11x speedup. Similarly, in $(4, 5)$ nucleus decomposition, using a 3-multi-level T offers comparable performance to the two-level T on dblp and skitter (the 3-multi-level T gives 1.46x and 1.06x speedups over the unoptimized T , respectively, while the 2-level T gives 1.32x and 1.06x speedups, respectively), but underperforms on amazon and youtube. The benefit of using large ℓ relative to r is difficult to observe for small r , since $\ell \leq r$ and speedups only appear in graphs with sufficiently many r -cliques. We see relatively small speedups from larger ℓ for certain graphs, but in these cases, the performance is comparable, so we consider the two-level T to be the best overall option.

Additionally, the two-level and multi-level options offer significant space savings due to their compact representation of vertices shared among many r -cliques, particularly for larger graphs and larger values of r and s . Figures 10.8 and 10.10 also show the space savings in T of each optimization on $(3, 4)$ nucleus decomposition and $(4, 5)$ nucleus decomposition, respectively; we omit the figures for $(2, 3)$ and $(2, 4)$ nucleus decomposition due to space limitations, although they show similar behavior. Across $(2, 3)$ and $(2, 4)$ nucleus decomposition, the two-level options give up to a 1.79x reduction in space usage, and this increases to up to a 2.15x reduction in space usage on $(3, 4)$ nucleus decomposition and up to a 2.51x reduction in space usage on $(4, 5)$ nucleus decomposition. We similarly see greater space reductions in using ℓ -multi-level T for $\ell > 2$ on $(4, 5)$ nucleus decomposition compared to the same ℓ on $(3, 4)$ nucleus decomposition for the same graphs; however, r is not large enough and there is not enough overlap between r -cliques such that using $\ell > 2$ offers significant space savings over the two-level options.

Overall, the optimal setting for the parallel hash table T is a two-level combination of an array and a parallel hash table, with contiguous space and stored pointers for the inverse index map.

Other optimizations. We now consider the graph relabeling and update aggrega-

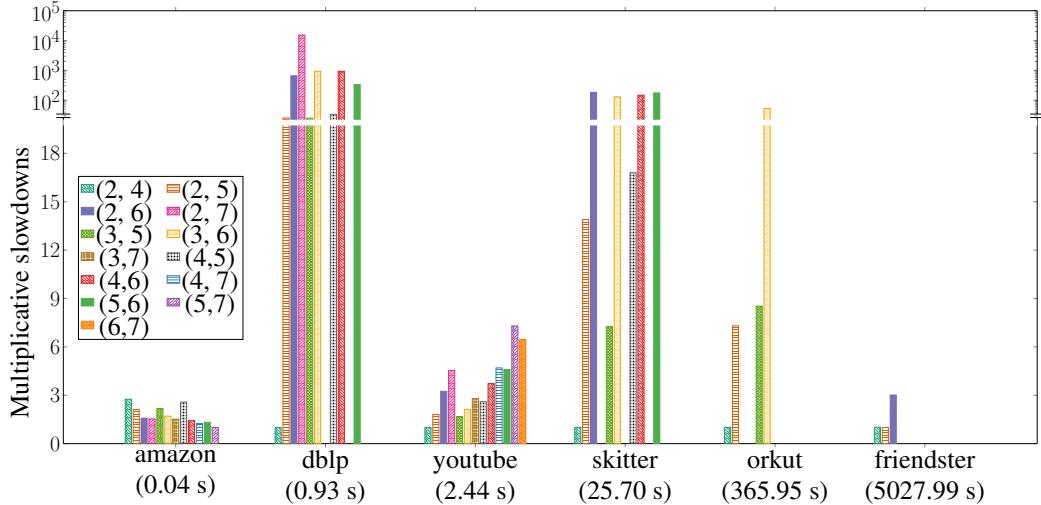


Figure 10.13: Multiplicative slowdowns of parallel ARB-NUCLEUS for each (r, s) combination over the fastest running time for parallel ARB-NUCLEUS across all $r < s \leq 7$ for each graph (excluding $(2, 3)$ and $(3, 4)$, which are shown in Figure 10.12). The fastest running time is labeled in parentheses below each graph. Also, livejournal is excluded because for these r and s , ARB-NUCLEUS is only able to complete $(2, 4)$ nucleus decomposition, in 484.74 seconds, and the rest timed out. We have omitted bars where ARB-NUCLEUS runs out of memory or times out.

tion optimizations, fixing a one-level setting and a two-level setting with contiguous space and stored pointers for the inverse index map, and using the simple array for U with a fetch-and-add to reserve every slot. Figure 10.11 shows these speedups for the two-level case; we omit the analogous figure for the one-level case, which shows similar behavior to the two-level case.

Across $(2, 3)$, $(2, 4)$, and $(3, 4)$ nucleus decomposition, the graph relabeling optimization gives up to 1.23x speedups on the one-level case and up to 1.29x speedups on the two-level case. We see greater speedups on the two-level case, because the increased locality across both levels due to the relabeling is more significant, whereas there is little improved locality in the one-level case. We note that graph relabeling provides minimal speedups in $(2, 3)$ nucleus decomposition, with slowdowns of up to 1.11x, whereas graph relabeling is almost universally optimal in $(2, 4)$ and $(3, 4)$ nucleus decomposition. This is due to fewer benefits from locality when merely computing triangle counts from edges, versus computing higher clique counts.

In terms of the options for update aggregation, using a list buffer gives up to 3.43x speedups on the one-level case and up to 3.98x speedups on the two-level case. Using a parallel hash table gives up to 3.37x speedups on the one-level case and up to 4.12x speedups on the two-level case. Notably, the parallel hash table is the fastest for $(2, 3)$ nucleus decomposition, whereas the list buffer outperforms the parallel hash table for $(2, 4)$ and $(3, 4)$ nucleus decomposition. This behavior is particularly evident on larger graphs, where the time to compute updated s -clique counts is more significant, and thus there is less contention in using a list buffer.

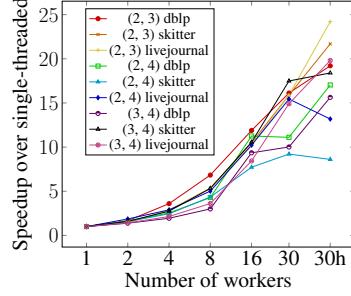


Figure 10.14: Speedup of ARB-NUCLEUS over its single-threaded running times. "30h" denotes 30-cores with two-way hyper-threading.

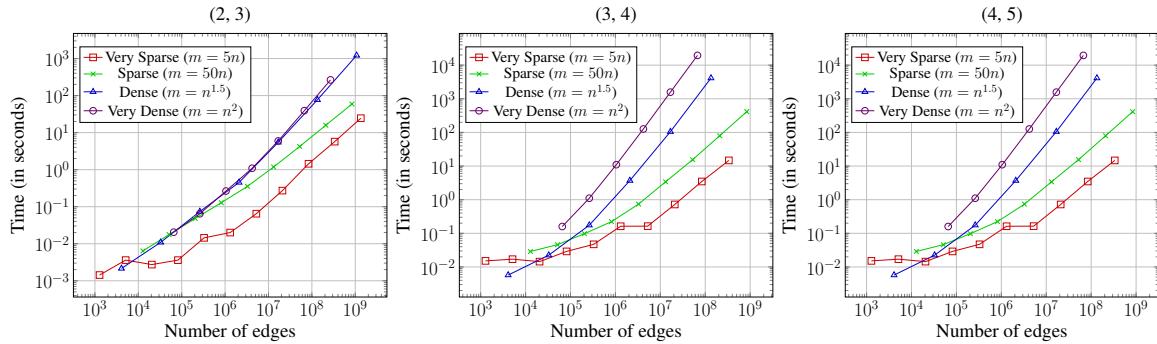


Figure 10.15: Running times of parallel ARB-NUCLEUS on rMAT graphs of varying densities for $(2, 3)$, $(3, 4)$, and $(4, 5)$ nucleus decomposition. We remove duplicate generated edges.

Finally, in the special case of $(2, 3)$ nucleus decomposition, we evaluate the performance of graph contraction over the two-level setting. We see up to 1.08x speedups using graph contraction, but up to 1.11x slowdowns when using graph contraction on small graphs, due to the increased overhead; however, the two-level running times of ARB-NUCLEUS on these graphs is < 0.2 seconds.

Overall, the optimal setting for $(2, 3)$ nucleus decomposition is to use a parallel hash table for update aggregation and graph contraction (with no graph relabeling), and the optimal setting for general (r, s) nucleus decomposition is to use a list buffer for update aggregation and graph relabeling. Combining all optimizations, over $(2, 3)$, $(2, 4)$, and $(3, 4)$ nucleus decomposition, we see up to a 5.10x speedup over the unoptimized ARB-NUCLEUS.

10.6.3 Performance

Figures 10.12 and 10.13 show the parallel runtimes for ARB-NUCLEUS using the optimal settings described in Section 10.6.2, for (r, s) where $r < s \leq 7$. We only show in these figures our self-relative speedups on $(r, s) = (2, 3)$ and $(r, s) = (3, 4)$, but we computed self-relative speedups for all $r < s \leq 7$, and overall, on 30 cores with two-way hyper-threading, ARB-NUCLEUS obtains 3.31–40.14x self-relative speedups. We

see larger speedups on larger graphs and for greater r . We also see good scalability over different numbers of threads, which we show in Figure 10.14 for (2, 3), (2, 4), and (3, 4) nucleus decomposition on dblp, skitter, and livejournal. Figure 10.15 additionally shows the scalability of ARB-NUCLEUS over rMAT graphs of varying sizes and varying edge densities. We see that our algorithms scale in accordance with the increase in the number of s -cliques, depending on the density of the graph.

Comparison to other implementations. Figure 10.12 also shows the comparison of our parallel (2, 3) and (3, 4) nucleus decomposition implementations to other implementations. We compare to Sariyüce *et al.*'s [284, 283] parallel implementations, including their global implementation PND, their asynchronous local implementation AND, and their asynchronous local implementation with notification AND-NN, where the notification mechanism offers performance improvements at the cost of space usage. We run the local implementations to convergence. We furthermore compare to their implementation ND, which is a serial version of PND. We note that Sariyüce *et al.* provide implementations only for (2, 3) and (3, 4) nucleus decomposition.

Compared to PND, ARB-NUCLEUS achieves 3.84–54.96x speedups, and compared to AND, ARB-NUCLEUS achieves 1.32–60.44x speedups. Notably, PND runs out of memory on friendster for (2, 3) nucleus decomposition, while our implementation can process friendster in 368.62 seconds. Moreover, AND runs out of memory on both orkut and friendster for both (2, 3) and (3, 4) nucleus decomposition, while ARB-NUCLEUS is able to process orkut and friendster for (2, 3) nucleus decomposition, and orkut for (3, 4) nucleus decomposition.

Compared to AND-NN, ARB-NUCLEUS achieves 1.04–8.78x speedups. AND-NN outperforms the other implementations by Sariyüce *et al.*, but due to its increased space usage, it is unable to run on the larger graphs skitter, livejournal, orkut, and friendster for both (2, 3) and (3, 4) nucleus decomposition. Considering the best of Sariyüce *et al.*'s parallel implementations for each graph and each (r, s) , ARB-NUCLEUS achieves 1.04–54.96x speedups overall. Compared to Sariyüce *et al.*'s serial implementation ND, ARB-NUCLEUS achieves 8.19–58.02x speedups. We significantly outperform Sariyüce *et al.*'s algorithms due to the work-efficiency of our algorithm. Notably, AND and AND-NN are not work-efficient because they perform a local algorithm, in which each r -clique locally updates its s -clique-core number until convergence, compared to the work-efficient peeling process where the minimum s -clique-core is extracted from the entire graph in each round. The total work of these local updates can greatly outweigh the total work of the peeling process. We measured the total number of times s -cliques were discovered in each algorithm, and found that AND computes 1.69–46.03x the number of s -cliques in ARB-NUCLEUS, with a median of 15.15x. AND-NN reduces this at the cost of space, but still computes up to 3.45x the number of s -cliques in ARB-NUCLEUS, with a median of 1.4x.

Moreover, PND does perform a global peeling-based algorithm like ARB-NUCLEUS, but does not parallelize within the peeling process; more concretely, all r -cliques with the same s -clique count can be peeled simultaneously, which ARB-NUCLEUS accomplishes by introducing optimizations, notably the update aggregation optimization, that specifically address synchronization issues when peeling multiple r -cliques si-

multaneously. PND instead peels these r -cliques sequentially in order to avoid these synchronization problems, leading to significantly more sequential peeling rounds than required in ARB-NUCLEUS; in fact, PND performs 5608–84170x the number of rounds of ARB-NUCLEUS.

We note that additionally, the speedups of ARB-NUCLEUS over PND, AND, and AND-NN are not solely due to our use of an efficient parallel k -clique counting subroutine [296]. We replaced the k -clique counting subroutine in ARB-NUCLEUS with that used by Sariyüce *et al.* [284, 283], and found that ARB-NUCLEUS using Sariyüce *et al.*'s k -clique counting subroutine achieves between 1.83–28.38x speedups over Sariyüce *et al.*'s best implementations overall for $(2, 3)$ and $(3, 4)$ nucleus decomposition. Within ARB-NUCLEUS, the efficient k -clique counting subroutine gives up to 3.04x speedups over the subroutine used by Sariyüce *et al.*, with a median speedup of 1.03x.

Comparison to k -truss implementations. In the special case of $(2, 3)$ nucleus decomposition, or k -truss, we compare our parallel implementation to Che *et al.*'s [70] highly optimized parallel CPU implementation PKT-OPT-CPU, using all of their successive optimizations and considering the best performance across different reordering options, including degree reordering, k -core reordering, and no reordering. Figure 10.12 also shows the comparison of ARB-NUCLEUS to PKT-OPT-CPU. ARB-NUCLEUS achieves up to 1.64x speedups on small graphs, but up to 2.27x slowdowns on large graphs compared to PKT-OPT-CPU. However, we note that PKT-OPT-CPU is limited in that it solely implements $(2, 3)$ nucleus decomposition, and its methods do not generalize to other values of (r, s) . ARB-NUCLEUS outperforms PKT-OPT-CPU on small graphs due to a more efficient graph reordering subroutine; ARB-NUCLEUS computes a low out-degree orientation which it then uses to reorder the graph, and ARB-NUCLEUS's reordering subroutine achieves a 3.07–5.16x speedup over PKT-OPT-CPU's reordering subroutine.⁶ PKT-OPT-CPU uses its own parallel sample sort implementation, which is slower compared to that used by ARB-NUCLEUS [98]. However, the cost of graph reordering is negligible compared to the cost of computing the $(2, 3)$ nucleus decomposition in large graphs, and PKT-OPT-CPU uses highly optimized intersection subroutines which achieve greater speedups over ARB-NUCLEUS's generalized implementation.

We also compared to additional $(2, 3)$ nucleus decomposition implementations. Specifically, we compared to Kabir and Madduri's PKT [186], which outperforms Che *et al.*'s PKT-OPT-CPU on the small graphs amazon and dblp by 1.01–1.53x, and which is also shown in Figure 10.12. However, ARB-NUCLEUS achieves 1.07–2.88x speedups over PKT on all graphs, including amazon and dblp. Moreover, we compared to Smith *et al.*'s MSP [305], but found that MSP is slower than PKT and PKT-OPT-CPU, and ARB-NUCLEUS achieves 2.35–7.65x speedups over MSP. We compared to Blanco *et al.*'s [45] implementations as well, which we found to also be slower than PKT and PKT-OPT-CPU, and ARB-NUCLEUS achieves 2.45–21.36x speedups over their best implementation.⁷ We also found that Che *et al.*'s implementations outperform the

⁶For PKT-OPT-CPU, we consider the reordering option that gives the fastest overall running time for each graph.

⁷Blanco *et al.*'s implementation requires the maximum k -truss number to be provided as an input, which we did in these experiments.

reported numbers from a recent parallel $(2, 3)$ nucleus decomposition implementation by Conte *et al.* [83].

10.7 Discussion

We have presented a novel theoretically efficient parallel algorithm for (r, s) nucleus decomposition, which improves upon the previous best theoretical bounds. We have also developed practical optimizations and showed that they significantly improve the performance of our algorithm. Finally, we have provided a comprehensive experimental evaluation demonstrating that on a 30-core machine with two-way hyper-threading our algorithm achieves up to 55x speedup over the previous state-of-the-art parallel implementation.

Chapter 11

Nucleus Decomposition Hierarchy

11.1 Introduction

The original formulation of the *nucleus decomposition problem*, by Sariyüce *et al.* [284], involves capturing higher-order structures in graphs in a hierarchy, as mentioned in Chapter 10. More precisely, a c - (r, s) nucleus is defined to be the maximal induced subgraph such that every r -clique in the subgraph is contained in at least c s -cliques. The goal of the (r, s) nucleus decomposition problem is to (1) identify for each r -clique in the graph, the largest c such that it is in a c - (r, s) nucleus (known as the coreness value) and (2) generate a *nucleus hierarchy* over the nuclei, where for $c' < c$ a c' - (r, s) nucleus A is a descendant of a c - (r, s) nucleus B if A is a subgraph of B . Figure 11.1 shows the hierarchy for $(1, 2)$ nucleus. The nucleus hierarchy is an unsupervised method for revealing dense substructures at different resolutions in the graph. Since the hierarchy is a tree, it is easy to visualize and explore as part of structural graph analysis tasks [281].

Sariyüce and Pinar present the first algorithm for solving the nucleus decomposition problem [281], and their algorithm is sequential. However, in order to scale to the large graph sizes of today, it is important to design parallel algorithms that take advantage of modern parallel hardware. The existing parallel algorithms for nucleus decomposition only computes the coreness values but do not generate the hierarchy. Sariyüce *et al.* [283] present two parallel algorithms for the nucleus decomposition problem: (1) a global peeling-based algorithm and (2) a local update model that iterates until convergence. These algorithms are either not very parallel or not work-efficient (i.e., takes significantly more work than the best sequential algorithm). In Chapter 10, we present an algorithm that is work-efficient with high parallelism. A work-efficient algorithm for nucleus decomposition runs in time proportional to enumerating all s -cliques, i.e., $O(m\alpha^{s-2})$ time where α is the arboricity of the input graph. Our algorithm leverages the work-efficient parallel clique counting algorithm from Chapter 5, along with a multi-level hash table structure to store data associated with cliques space efficiently, and techniques for traversing this structure in a cache-

friendly manner. However, the lack of a hierarchy in the output of existing algorithms limits the applicability of this work.

In this chapter, we design the first work-efficient parallel algorithm for hierarchy construction in nucleus decomposition. Our work-efficient algorithm runs in $O(m\alpha^{s-2})$ expected work and $O(k \log n + \rho_{(r,s)}(G) \log n + \log^2 n)$ span w.h.p., where $\rho_{(r,s)}(G)$ is the (r, s) peeling complexity of G and k is the maximum (r, s) -clique core number in G . The key to our theoretical efficiency is our careful construction of subgraphs representing the s -clique-connectivity of r -cliques, that allows us to exploit linear-work graph connectivity instead of more expensive union-finds as used in prior work. Our approach gives as a byproduct the most theoretically-efficient serial algorithm for computing the hierarchy, improving upon the previous best known serial bounds by Sariyüce and Pinar [281].

We also present a practical parallel algorithm that interleaves the hierarchy construction with the computation of the coreness values. Prior work by Sariyüce and Pinar [281] also includes a (serial) interleaved hierarchy algorithm. However, their algorithm requires storing all adjacent r -cliques with different coreness values, which could potentially be proportional to the number of s -cliques in G (i.e., use $O(m\alpha^{s-2})$ space), and which results in sequential dependencies in their post-processing step to construct the hierarchy. Our parallel algorithm fully interleaves the hierarchy construction with the coreness computation, and uses only two additional arrays of size proportional to the number of r -cliques in G . Our algorithm uses a concurrent union-find data structure in an innovative way. Also, our post-processing step to construct the hierarchy tree is fully parallel. Our main insight is a technique to fully extract the connectivity information from adjacent r -cliques with different core numbers while computing the coreness values.

Note that the span of the above algorithms can be large for graphs with large peeling complexity ($\rho_{(r,s)}(G)$). We introduce an approximate algorithm for nucleus decomposition and show that it can achieve work efficiency and polylogarithmic span. Our algorithm relaxes the peeling order by allowing all r -cliques within a $((\binom{s}{r}) + \epsilon)$ factor of the current value of k to be peeled in parallel. We show that our algorithm generates coreness estimates that are an $((\binom{s}{r}) + \epsilon)$ -approximation of the true coreness values.

We experimentally study our parallel algorithms on real-world graphs using different (r, s) values, for up to $r < s \leq 7$. On a 30-core machine with two-way hyper-threading, our exact algorithm which generates both the coreness numbers and the hierarchy achieves up to a 30.96x self-relative parallel speedup, and a 3.76–58.84x speedup over the state-of-the-art sequential algorithm by Sariyüce and Pinar [281]. In addition, we show that on the same machine our approximate algorithm is up to 3.3x faster than our exact algorithm for computing coreness values, while generating coreness estimates with a multiplicative error of 1–2.92x (with a median error of 1.33x). Our algorithms are able to compute the (r, s) nucleus decomposition hierarchy for $r > 3$ and $s > 4$ on graphs with over a hundred million edges for the first time.

We summarize our contributions below:

- The first exact and approximate parallel algorithms for nucleus decomposition that generates the hierarchy with strong theoretical bounds on the work, span, and approximation guarantees.
- A number of practical optimizations that lead to fast implementations of our algorithms.
- Experiments showing that our new exact algorithm achieves 3.7–58.8x speedups over state of the art on a 30-core machine with two-way hyper-threading.
- Experiments showing that our new approximate algorithm achieves up to 3.3x speedups over our exact algorithm, while generating coreness estimates with a multiplicative error of 1–2.9x (with a median error of 1.3x).

Our code is publicly available at: <https://github.com/jeshi96/arb-nucleus-hierarchy>.

11.2 Related Work

The k -core [290, 229] and k -truss problems [79, 350, 353] are classic problems that relate to dense substructures in graphs. Many algorithms have been developed for k -cores and k -trusses in the static [144, 238, 128, 189, 185, 95, 337, 103, 324, 71, 355, 186, 305, 70, 83, 218, 45] and dynamic [209, 337, 349, 279, 11, 171, 18, 214, 224, 169, 310, 184, 222, 351, 172, 216] settings. Similar ideas have been studied in bipartite graphs [298, 329, 202, 282, 215, 173].

A related problem is the k -clique densest subgraph problem [317], which defines the density of subgraphs based on the number of k -cliques in it, rather than the number of edges. Fang *et al.* [127] further generalize this notion to arbitrary fixed-sized subgraphs (although they only present experiments for cliques). Similar to k -core and k -truss, algorithms for approximating the k -clique densest subgraph are based on iteratively peeling (removing) elements from the graph. Shi *et al.* [296] present efficient parallel algorithms for solving the k -clique densest subgraph problem.

Sariyüce *et al.* define the (r, s) nucleus decomposition problem and show that it can be used to find higher quality dense substructures in graphs than previous approaches. The authors provide efficient sequential [284, 281] and parallel algorithms [283] for this problem. They introduce a sequential algorithm for constructing the nucleus decomposition hierarchy [281], but as far as we know there are no parallel algorithms for constructing this hierarchy. Their algorithm is not work-efficient as it uses union-find. We also note that their space-usage can in theory be as large as $O(n\alpha^{s-2})$, i.e., proportional to the number of s -cliques in the graph. Chu *et al.* [76] present a parallel algorithm for generating the k -core decomposition hierarchy, which is a special case of nucleus decomposition for $r = 1$ and $s = 2$, but their algorithm does not trivially generalize to higher r and s . Their serial and parallel algorithms both use union-find and run in $O(m\alpha(n))$ work (where $\alpha(n)$ is the inverse Ackermann function), which is not work-efficient, and their parallel algorithm has depth that depends on the peeling-complexity of the input. Shi *et al.* [297] present an improved parallel algorithm for nucleus decomposition, with good theoretical guarantees and practical performance, but it does not generate the hierarchy. More recently, Sariyüce generalizes the r -

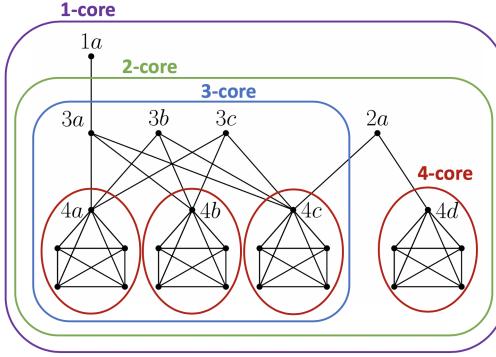


Figure 11.1: An example of the $(1, 2)$ nucleus (k -core) hierarchy.

cliques and s -cliques in nucleus decomposition to any pair of subgraphs [278]. There has also been work on nucleus decomposition in probabilistic graphs [123].

11.3 Preliminaries

A c - (r, s) **nucleus** is a maximal subgraph H of an undirected graph formed by the union of s -cliques S , such that each r -clique R in H has induced s -clique degree at least c (i.e., each r -clique is contained within at least c induced s -cliques). The goal of the (r, s) **nucleus decomposition problem** is to compute the following: (1) the (r, s) -**clique core number** of each r -clique R , or the maximum c such that R is contained within a c - (r, s) nucleus and (2) a hierarchy over the nuclei, where for $c' < c$, a c' - (r, s) nucleus A is a descendant of a c - (r, s) nucleus B if A is a subgraph of B . In this chapter, similar to all prior work on nucleus computations, we take r and s to be constants and ignore constants depending on these values in our bounds. The k -core and k -truss problems correspond to the k - $(1, 2)$ and k - $(2, 3)$ nucleus, respectively. Figure 11.1 shows the hierarchy for $(1, 2)$ nucleus (k -core). We call two r -cliques **s -clique-adjacent** if there exists an s -clique S such that both r -cliques are subgraphs of S . We also define the **s -clique-degree** of a r -clique R to be the number of s -cliques that R is contained within.

We also consider approximate nucleus decompositions in this chapter. If the true (r, s) -clique core number of an r -clique R is k_R , then a γ -**approximate** (r, s) -**clique core number** of R is a value that is at least k_R and at most γk_R , where $\gamma > 1$.

11.4 Nucleus Decomposition Hierarchy

We now describe our theoretically efficient parallel nucleus decomposition hierarchy algorithm, ARB-NUCLEUS-HIERARCHY. ARB-NUCLEUS-HIERARCHY computes the hierarchy by first running an efficient parallel nucleus decomposition algorithm, ARB-NUCLEUS [297], in order to obtain the (r, s) -clique core numbers corresponding to each r -clique. It then constructs a data structure consisting of k levels, where k is

Algorithm 11.1 – Parallel (r, s) nucleus decomposition hierarchy algorithm

```

1: Initialize  $r, s$                                  $\triangleright r$  and  $s$  for  $(r, s)$  nucleus decomposition
2: procedure ARB-NUCLEUS-HIERARCHY( $G = (V, E)$ )
3:    $ND \leftarrow$  ARB-NUCLEUS( $G$ )  $\triangleright$  Compute the nucleus core numbers, where  $ND$  maps
    $r$ -cliques to their core numbers
4:    $k \leftarrow$  maximum core number in  $ND$ 
5:   For each  $i \in [k]$ , let  $L_i$  denote a hash table, where the keys are  $r$ -cliques and the
   values are linked lists
6:   parfor all  $s$ -cliques  $S$  in  $G$  do
7:     parfor all pairs of  $r$ -cliques  $R, R'$  in  $S$  where  $ND[R'] \leq ND[R]$  do
8:       Add  $R'$  to the linked list keyed by  $R$  in  $L_{ND[R']}$ 
9:   Initialize the hierarchy tree  $T$  with leaves corresponding to each  $r$ -clique
10:  For each  $i \in [k]$ , let  $ID_i$  denote a hash table, where the keys are  $r$ -cliques and the
    values are  $r$ -cliques
11:  Initialize each  $ID_i$  to contain each key in  $L_i$ , mapped to itself
12:  for  $i \in \{k, k - 1, \dots, 1\}$  do
13:    Relabel each  $r$ -clique in each linked list in  $L_i$  with its corresponding value in  $ID_i$ 
14:    Apply parallel list ranking to transform the linked lists in  $L_i$  to arrays, which
    forms a graph  $H$  (where the edges are given by each key paired with each element in its
    corresponding linked list)
15:    Run parallel linear-work connectivity on  $H$ 
16:    parfor each connected component  $\mathcal{C} = \{R_1, \dots, R_c\}$  in  $H$  where  $|\mathcal{C}| \geq 2$  do
17:      Construct a new parent in  $T$  for all of the nodes corresponding to each  $R_\ell$ 
      (for  $\ell \in [c]$ ), and represent the parent as the  $r$ -clique  $R_1$ 
18:      parfor each  $j \leq i$  do
19:        Concatenate in parallel the linked lists corresponding to all  $R_\ell$  (for  $\ell \in [c]$ )
        in  $L_j$ , as the updated value for the key  $R_1$  in  $L_j$ 
20:        For each  $R_\ell$  (for  $\ell \in [c]$ ), update the value of  $R_\ell$  in  $ID_j$  to be  $R_1$ 
21:    return  $T$ 

```

the maximum (r, s) -clique core number. Each level is represented by a hash table mapping sets of r -cliques to a linked list of r -cliques, which is used to efficiently store s -clique-adjacent r -cliques. ARB-NUCLEUS-HIERARCHY proceeds in levels through this data structure to construct the hierarchy tree from the bottom up, by performing linear-work connectivity on each level and updating the connectivity information in prior levels as a result.

Our Algorithm. We now provide a more detailed description of our algorithm. The pseudocode is in Algorithm 11.1. We refer to the example of $(r, s) = (1, 2)$ nucleus (k -core) decomposition in Figure 11.1. For simplicity, we have omitted labeling the other vertices participating in the 5-cliques represented by $4a, 4b, 4c$, and $4d$. The number in front of each vertex label is its core number.

ARB-NUCLEUS-HIERARCHY first calls ARB-NUCLEUS on Line 3, to compute the (r, s) -clique core numbers of each r -clique. It stores these values in a hash table, ND , keyed by the r -cliques. Then, it constructs a data structure consisting of k hash

| | |
|---|--|
| | <i>i</i> = 4: Connected component = \emptyset |
| | <i>i</i> = 3: |
| $L_1 = \begin{array}{ c c } \hline \text{key} & 3a \\ \hline \text{value} & \{1a\} \\ \hline \end{array}$, $L_2 = \begin{array}{ c c c } \hline \text{key} & 4c & 4d \\ \hline \text{value} & \{2a\} & \{2a\} \\ \hline \end{array}$ | $ID_3 = \begin{array}{ c c c c c c c c c c } \hline \text{r-clique} & 1a & 2a & 3a & 3b & 3c & 4a & 4b & 4c & 4d \\ \hline \text{value} & 1a & 2a & 3a & 3b & 3c & 4a & 4b & 4c & 4d \\ \hline \end{array}$ Connected component = {3a, 3b, 3c, 4a, 4b, 4c} |
| $L_3 = \begin{array}{ c c c c } \hline \text{key} & 4a & 4b & 4c \\ \hline \text{value} & \{3a, 3b, 3c\} & \{3a, 3b, 3c\} & \{3a, 3b, 3c\} \\ \hline \end{array}$, $L_4 = \emptyset$ | <i>i</i> = 2: $ID_2 = \begin{array}{ c c c c c c c c c c } \hline \text{r-clique} & 1a & 2a & 3a & 3b & 3c & 4a & 4b & 4c & 4d \\ \hline \text{value} & 1a & 2a & 3a & 3a & 3a & 3a & 3a & 3a & 4d \\ \hline \end{array}$ Connected component = {2a, 3a, 4d} |
| | <i>i</i> = 1: $ID_1 = \begin{array}{ c c c c c c c c c c } \hline \text{r-clique} & 1a & 2a & 3a & 3b & 3c & 4a & 4b & 4c & 4d \\ \hline \text{value} & 1a & 2a & 2a & 3a & 3a & 3a & 3a & 3a & 2a \\ \hline \end{array}$ Connected component = {1a, 2a} |

Figure 11.2: An example of the L_i data structures maintained by ARB-NUCLEUS-HIERARCHY while computing the k -core hierarchy on the graph in Figure 11.1. For each round i of the hierarchy construction, the connected components of H and the ID_i table used to construct H is shown (except ID_4 , which maps every r -clique to itself).

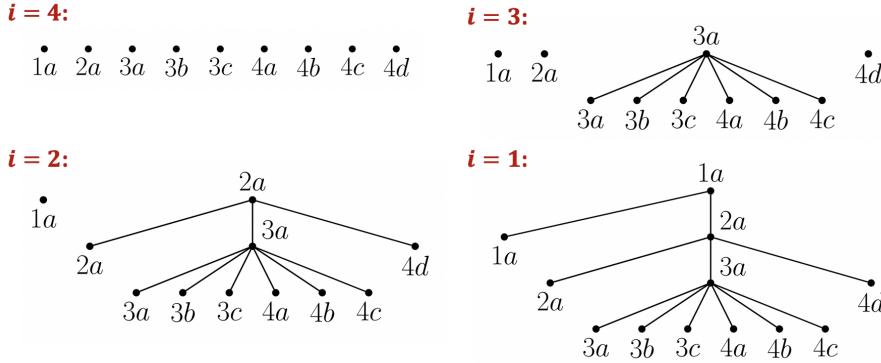


Figure 11.3: An example of the intermediate hierarchy trees T after each round i in ARB-NUCLEUS-HIERARCHY, while computing the k -core hierarchy on the graph in Figure 11.1.

tables on Line 5, where k is the maximum (r, s) -clique core number. Each hash table, L_i , maps r -cliques to a linked list of r -cliques, where i corresponds to a core number. The first stage of ARB-NUCLEUS-HIERARCHY inserts all s -clique-adjacent r -cliques into the hash tables on Lines 6–8. Specifically, for adjacent r -cliques R and R' where $ND[R'] \leq ND[R]$, we insert R' into the hash table corresponding to R' 's core number, $L_{ND[R']}$, with $\{R\}$ as the key. If an entry for $\{R\}$ already exists in $L_{ND[R']}$, we append R' to the existing linked list. The hash tables are shown in Figure 11.2. For instance, $R' = 1a$ is adjacent to $R = 3a$, so we add $1a$ to the linked list keyed by $3a$ in L_1 .

In order to iterate in parallel over all r -cliques and over all s -cliques containing each r -clique, our algorithm uses a s -clique enumeration algorithm based on previous work by Shi *et al.* [296], which recursively finds and lists c -cliques in parallel and which can efficiently extend given r -cliques to find the s -cliques they are contained within. ARB-NUCLEUS-HIERARCHY then takes all combinations of r vertices in each

discovered s -clique to find adjacent r -cliques in Lines 7–8. Note that this s -clique enumeration subroutine is already used in ARB-NUCLEUS in order to compute the (r, s) -clique core numbers of each r -clique, and in practice, we can construct the hash tables L_i while computing the core numbers in ARB-NUCLEUS (rather than running the s -clique enumeration subroutine twice).

After constructing the initial set of hash tables, ARB-NUCLEUS-HIERARCHY proceeds to construct the nucleus decomposition hierarchy on Lines 9–20. The broad idea is to construct the hierarchy starting at the leaf nodes, each of which correspond to an r -clique, and to merge tree nodes from the bottom up (i.e., in decreasing order of core number). The algorithm begins by considering only connected components formed by r -cliques with the greatest core number, k , which dictate the leaf nodes to be merged into super-nodes at the second-to-last level of the hierarchy tree based on their s -clique connectivity. In subsequent rounds, when processing core number i , the algorithm considers connected components formed by r -cliques with core numbers $\geq i$, which similarly dictate the merges to be performed in the next level of the hierarchy tree. ARB-NUCLEUS-HIERARCHY efficiently maintains connected components from higher core numbers when computing connected components at lower core numbers, by concatenating the relevant linked lists in the hash tables corresponding to the lower core numbers when processing higher core numbers. Figure 11.3 shows the construction of this hierarchy throughout the different rounds (for $i = 4, \dots, 1$), and Figure 11.2 lists the connected components computed in each round. In our example, since we omit the other vertices in each component represented by $4a, 4b, 4c$, and $4d$ (for simplicity), we begin with singleton leaf nodes after processing round $i = 4$.

In more detail, on Line 9, we begin with a hierarchy tree T consisting only of leaf nodes corresponding to each r -clique. We also initialize a data structure consisting of k hash tables on Lines 10–11, where each hash table ID_i maps r -cliques to r -cliques. The idea is to maintain a mapping of each r -clique to the component that it is contained within in L_i , for the corresponding level. Initially, each ID_i maps each key in L_i to itself.

Then, ARB-NUCLEUS-HIERARCHY considers each core number i , starting from k and going down to 1 (Line 12). For each i , we relabel each r -clique in each linked list in L_i with its corresponding value in ID_i (Line 13). Then, it uses parallel list ranking to convert all linked lists in L_i to arrays, forming a graph H where edges are given by each key paired with each element in its corresponding linked list (Line 14). Note that the edges actually denote s -clique-adjacent r -cliques (or components of r -cliques). The vertices correspond to r -cliques, which represent components of r -cliques in G . In our example, we process the hash table L_3 in round $i = 3$. ID_3 maps every vertex to itself, so we do not relabel any of the labels in L_3 . The graph H that we construct consists of the vertices $3a, 3b, 3c, 4a, 4b$, and $4c$, with edges given by $\{3a, 3b, 3c\} \times \{4a, 4b, 4c\}$.

ARB-NUCLEUS-HIERARCHY proceeds by running an efficient parallel linear-work connectivity algorithm on H (Line 15), and processes the given connected components. Note that each connected component C consists of a set of vertices in H , which represents a set of r -cliques, say $\{R_1, \dots, R_c\}$. In our example, in round $i = 3$, we have a single connected component in H consisting of all of the vertices, $3a, 3b, 3c$,

$4a$, $4b$, and $4c$. Then, for each such connected component representing more than one r -clique, in the hierarchy tree T , we construct a new parent for the nodes corresponding to each R_ℓ for $\ell \in [c]$ (Line 17). Note that each R_ℓ could correspond to a subtree containing multiple tree nodes, owing to parent nodes constructed in previous steps. We represent the new parent by the r -clique R_1 , which we designate arbitrarily as the representative for the connected component C . In Figure 11.3, under $i = 3$, we construct a new parent node labeled by $3a$, which we designate as the representative of the component $\{3a, 3b, 3c, 4a, 4b, 4c\}$.

Finally, for each core number $j < i$, we update the connectivity information on each hash table L_j , by concatenating all linked lists in L_j corresponding to the r -cliques in the component C (Line 19). More precisely, we update the value of the key R_1 to be the concatenation, or we insert the concatenation into L_j with R_1 as the key if R_1 does not already exist in the hash table (this is possible if R_1 had no neighbors with core number j). We use tombstones to delete the other keys R_ℓ ($\ell \neq 1$) in each L_j . Additionally, on Line 20, we update the component ID in ID_j of each R_ℓ to be R_1 . For $i = 3$, we see that in our example there are no lists to concatenate, but in ID_2 , we map each of $3b$, $3c$, $4a$, $4b$, and $4c$ to the representative $3a$.

We repeat this process until we have processed all k rounds. In our example, for round $i = 2$, we relabel $4c$ in L_2 with $3a$, as given by ID_2 , and we construct a subgraph H with vertices $2a$, $3a$, and $4d$, and edges $(2a, 3a)$ and $(2a, 4d)$. Again, there is only one connected component, given by $\{2a, 3a, 4d\}$. In the hierarchy tree for round $i = 2$, we construct a new parent labeled with the representative $2a$, whose children are the leaves $2a$ and $4d$, and the previously constructed parent node $3a$. Again, there are no lists to concatenate in L_1 , but in ID_1 , we map both $3a$ and $4d$ to the representative $2a$. Then, for round $i = 1$, we relabel $3a$ in L_1 with $2a$, as given by ID_1 . We construct a subgraph H with vertices $1a$ and $2a$, and a single edge between them. There is only one connected component, $\{1a, 2a\}$, and in the hierarchy tree for round $i = 1$, we construct a new parent labeled with the representative $1a$, whose children are the leaf $1a$ and the previously constructed parent node $2a$. This concludes the construction of the hierarchy T in our example.

Theoretical Efficiency. We now analyze the theoretical efficiency of our hierarchy algorithm, ARB-NUCLEUS-HIERARCHY. Note that as in Shi *et al.*'s [297] work, $\rho_{(r,s)}(G)$ is defined to be the *(r,s) peeling complexity* of G , or the number of rounds needed to peel the graph where in each round, all r -cliques with the minimum s -clique count are peeled (removed). Importantly, $k \leq \rho_{(r,s)}(G) \leq O(m\alpha^{r-2})$, since at least one r -clique is peeled in each round, and the number of rounds is at least the maximum (r,s) -clique core number in G .

Theorem 11.1. ARB-NUCLEUS-HIERARCHY computes the (r,s) nucleus decomposition hierarchy in $O(m\alpha^{s-2})$ expected work and $O(k \log n + \rho_{(r,s)}(G) \log n + \log^2 n)$ span w.h.p., where $\rho_{(r,s)}(G)$ is the (r,s) peeling complexity and k is the maximum (r,s) -clique core number.

Proof. First, the theoretical complexity of computing the (r,s) -clique core numbers of each r -clique (Line 3) is given by Shi *et al.* [297], which they show takes $O(m\alpha^{s-2})$

work and $O(\rho_{(r,s)}(G) \log n + \log^2 n)$ span w.h.p. for constant r and s . Note that the work bound is given by the version of their algorithm that takes space proportional to the number of s -cliques in G , which we incur regardless to store the hash tables L_i . Additionally, iterating through all s -cliques in G (Line 6) is superseded by the work required to compute the (r, s) -clique core numbers. For every pair of r -cliques in each s -cliques, we hash each r -clique and append to a linked list (Lines 7–8), which in total takes $O(m\alpha^{s-2})$ work and $O(\log^2 n)$ span w.h.p. for constant r and s .

We now discuss the work and span of constructing the hierarchy tree T level-by-level, in k rounds (Lines 12–20). The key idea here is that the linked lists in each hash table L_i are iterated over at most once across all rounds, and the concatenation of linked lists in intermediate rounds incurs minimal costs (since concatenating linked lists does not require iterating through the linked lists). The sum of the lengths of the linked lists in each L_i remains invariant throughout this portion of the construction, so in total, the cost of iterating over the linked lists is bounded by $O(m\alpha^{s-2})$ work, matching the work needed to construct each linked list in each L_i originally. Also, the number of keys in each L_i only monotonically decreases, so processing the connected components of the constructed subgraphs H takes at most $O(m\alpha^{s-2})$ work as well.

In more detail, for fixed i , let the sum of the lengths of the linked lists in L_i be ℓ_i , and let the number of keys in L_i be y_i . For each round $i \in [k]$, applying ID_i and parallel list ranking on the linked lists in L_i (Lines 13–14) takes work proportional to ℓ_i and $O(\log n)$ span w.h.p.. The number of edges in the subgraph H constructed from the linked lists in L_i is at most ℓ_i , so performing connectivity on H (Line 15) takes work linear in ℓ_i and $O(\log n)$ span w.h.p. [143]. Also, there are at most $O(y_i)$ vertices in H , so processing each connected component and updating the hierarchy tree T (Line 17) takes $O(y_i)$ work and $O(1)$ span. In total, for Lines 13–17, we incur $O(\sum_i (\ell_i + y_i)) = O(m\alpha^{s-2})$ expected work and $O(k \log n)$ span w.h.p.

It remains to bound the cost of the loop in Lines 18–20. First, note that across all rounds, each value corresponding to every key in each L_i is concatenated at most once (Line 19). This is because we concatenate linked lists and store the concatenation under exactly one existing key. We empty the corresponding values for the previously associated keys, which allows us to maintain the invariant that the sum of the lengths of the linked lists in each L_i remains fixed. Also, the previously associated keys are never used again, since we update the value associated with each r -clique in ID_j . Thus, the concatenations are bounded by $O(m\alpha^{s-2})$ work across all rounds, matching the work of constructing all L_i . Additionally, iterating through all levels $j \leq i$ for each component \mathcal{C} does not increase the work. This is because we only reach this loop if $|\mathcal{C}| \geq 2$, which means that we are effectively merging multiple r -cliques together in each hash table L_j for $j \leq i$, and assigning the r -clique R_1 as the new representative for the other r -cliques in the component (updating ID_j). Thus, we can assign the work of iterating through all $j \leq i$ to the r -cliques that are being merged (R_ℓ where $\ell \neq 1$). Once the r -clique R_ℓ is merged, due to the concatenation on Line 19 and the update in ID_j on Line 20, it never participates as a vertex in H again in future rounds, so is never re-processed in a future connected component of H ; this is due to the mapping on Line 13. The amount of work that we assign per merged r -clique R_ℓ is at most the core number of R_ℓ . This is because the amount

of work we assign to R_ℓ is given by the number of rounds in which we merge, or i , and R_ℓ only appears in L_i if $ND[R_\ell] \geq i$, by construction on Line 8. Thus, in total, for each r -clique, we incur work upper bounded by the core number. We have in total $O(\sum_{r\text{-clique } R \in G} ND[R]) = O(m\alpha^{s-2})$ work, since the sum of the (r, s) -clique core numbers in G is necessarily bounded by the number of s -cliques for constant r and s . This is because each s -clique contributes to at most $\binom{s}{r}$ r -clique's core numbers, so the summation across all core numbers is upper bounded by $\binom{s}{r} \cdot$ (the number of s -cliques). Thus, in total, the loop in Lines 18–20 incurs $O(m\alpha^{s-2})$ work and $O(k \log n)$ span, where the span is due to list ranking.

In total, we have $O(m\alpha^{s-2})$ expected work and $O(k \log n + \rho_{(r,s)}(G) \log n + \log^2 n)$ span w.h.p., as desired. \square

Comparison to Prior Work. The prior state-of-the-art algorithm is the sequential (r, s) nucleus decomposition hierarchy algorithm by Sariyüce and Pinar [281]. They provide an algorithm similar to ARB-NUCLEUS-HIERARCHY in that it first computes the (r, s) -clique core numbers of each r -clique, and then builds the hierarchy tree from the bottom up. They show that the time complexity is upper bounded by the time complexity of computing the (r, s) -clique core numbers. Note that they omit a factor of $O(\alpha(n_s, n_r))$ where α is the inverse Ackermann function and n_s and n_r are the number of s -cliques and r -cliques, respectively, in the graph; this factor is necessary for their algorithm, since they use union-find to construct the hierarchy tree. Thus, the time complexity of Sariyüce and Pinar's algorithm is $O(m\alpha^{s-2}\alpha(n_s, n_r))$ (this is achieved using the state-of-the-art algorithm for computing the (r, s) -clique core numbers [297]). Our ARB-NUCLEUS-HIERARCHY algorithm avoids the additional inverse Ackermann factor, by efficiently constructing graphs to represent different levels of the hierarchy tree and using linear-work graph connectivity to construct the hierarchy tree. Thus, we improve the sequential running time of constructing the (r, s) nucleus decomposition hierarchy to $O(m\alpha^{s-2})$, and ARB-NUCLEUS-HIERARCHY is work-efficient.

11.5 Approximate Nucleus Decomposition

Given the potentially large span (i.e., longest critical path) of computing the exact nucleus decomposition hierarchy, we develop a new parallel approximate nucleus decomposition hierarchy algorithm, ARB-APPROX-NUCLEUS-HIERARCHY. Instead of computing exact (r, s) -clique-core numbers of each r -clique, the main idea of the new algorithm is to compute an approximation of the (r, s) -clique core number. Specifically, we compute a $(\binom{s}{r} + \varepsilon)$ -approximate (r, s) -clique core number of each r -clique; that is to say, if we let the true (r, s) -clique core number of each r -clique be k_R , our approximation is at least k_R and at most $(\binom{s}{r} + \varepsilon) \cdot k_R$.

Our approximate computation uses the same peeling paradigm as that from Shi *et al.* [297] for exact nucleus decomposition, but with an important modification that allows it to take only $O(\log^2 n)$ rounds of peeling, thus significantly improving upon the span of the algorithm. As a result, we only have polylogarithmically many

core numbers, leading to a hierarchy tree with polylogarithmic height. Notably, our hierarchy construction for ARB-APPROX-NUCLEUS-HIERARCHY is exactly the same as that of ARB-NUCLEUS-HIERARCHY, and the only salient difference between the two is that for ARB-APPROX-NUCLEUS-HIERARCHY, we replace the ARB-NUCLEUS subroutine in Line 3 of Algorithm 11.1 with our approximate nucleus decomposition subroutine, APPROX-ARB-NUCLEUS.

Our Algorithm. We present our pseudocode for APPROX-ARB-NUCLEUS in Algorithm 11.2. Note that it takes as input a parameter $\delta > 0$, which controls the ε in the $(\binom{s}{r} + \varepsilon)$ -approximation. Specifically, APPROX-ARB-NUCLEUS gives a $(\binom{s}{r} + \delta) \cdot (1 + \delta) = (\binom{s}{r} + \varepsilon)$ -approximation, which we prove in Theorem 11.2.

First, on Line 3, APPROX-ARB-NUCLEUS computes a low out-degree orientation of the graph G , which directs the edges such that every vertex has out-degree at most $O(\alpha)$, using an efficient algorithm by Shi *et al.* [296]. Then, on Lines 4–5, it counts the number of s -cliques per r -clique in G , and stores the result in a parallel hash table U . It uses an s -clique counting subroutine, REC-LIST-CLIQUEs, from Shi *et al.*'s previous work [296]. The key difference between APPROX-ARB-NUCLEUS and ARB-NUCLEUS is on Line 6, where the buckets in the bucketing structure hold r -cliques with a range of s -clique-degrees instead of a single s -clique-degree. Specifically, for an input parameter δ , we define the range of each bucket B_i to be $[(\binom{s}{r} + \delta) \cdot (1 + \delta)^i, (\binom{s}{r} + \delta) \cdot (1 + \delta)^{i+1}]$, where $i \in [s \log_{1+\delta} n]$ since $\binom{n}{s} = O(n^s)$ is a trivial upper bound on the maximum s -clique-degree possible in any given graph.

The peeling algorithm then proceeds as it does in [297], except using our modified bucketing structure. While not all r -cliques have been peeled, APPROX-ARB-NUCLEUS processes the set of r -cliques A (that have not yet been peeled) within the lowest bucket B_i (starting with $i = 0$), and peels them from the graph (Lines 8–20). For each r -clique R in A , we iterate over all s -clique-adjacent r -cliques R' , and update the recorded s -clique-degree of R' given R 's removal (Lines 12–15). Then, we peel (remove) the r -cliques in A from the graph and update the buckets of all unpeeled r -cliques based on the updated s -clique-degrees on Line 16. Notably, if a r -clique R 's s -clique-degree falls below the range of the current bucket of r -cliques that is being peeled, we do not rebucket R into a lower bucket, and instead aggregate these r -cliques within the current bucket. As such, in any given round of peeling, we are actually peeling all r -cliques with s -clique-degree $\leq (\binom{s}{r} + \delta) \cdot (1 + \delta)^i$, which is important for our theoretical bounds. Note that we process a given bucket B_i at most $O(\log_{1+\delta}/\binom{s}{r}(n))$ times; if we have exceeded this threshold, or if B_i is empty, then we move on to processing the next bucket, B_{i+1} (Lines 17–20). If there are unpeeled r -cliques remaining in B_i once we reach this threshold, we include them in the next bucket B_{i+1} (Line 18). Note that the approximate (r, s) -clique core number that we compute for each r -clique is given by the upper bound of the bucket in which it was peeled. In practice, we can improve this by taking the minimum of the upper bound of the bucket, and the s -clique-degree of each r -clique (in the original graph).

Once we have peeled all r -cliques, this concludes our subroutine. ARB-APPROX-NUCLEUS-HIERARCHY is then given by replacing ARB-NUCLEUS in Line 3 of Algorithm 11.1 with APPROX-ARB-NUCLEUS.

Algorithm 11.2 – Approximate parallel (r, s) nucleus decomposition algorithm

```

1: Initialize  $r, s$                                  $\triangleright r$  and  $s$  for  $(r, s)$  nucleus decomposition
2: procedure APPROX-ARB-NUCLEUS( $G = (V, E), \delta$ )
3:    $DG \leftarrow \text{ARB-ORIENT}(G)$             $\triangleright$  Apply an arboricity-orientation algorithm
4:   Initialize  $U$  to be a parallel hash table with  $r$ -cliques as keys, and  $s$ -clique counts as
   values
5:   REC-LIST-CLIQUEs( $DG, s, U$ )  $\triangleright$  Count  $s$ -cliques, and store the counts per  $r$ -clique
   in  $U$ 
6:   Let  $ND$  be a bucketing structure mapping each  $r$ -clique to a bucket based on # of
    $s$ -cliques, where each bucket  $B_i$  contain all  $r$ -cliques with  $s$ -clique-degree in the range
    $[(\binom{s}{r} + \delta) \cdot (1 + \delta)^i, (\binom{s}{r} + \delta) \cdot (1 + \delta)^{i+1}]$ , for all  $i \in [s \log_{(\binom{s}{r})+\delta} n]$ 
7:   finished  $\leftarrow 0$ , num_rounds  $\leftarrow 0$ ,  $i \leftarrow 0$ 
8:   while finished  $< |U|$  do
9:      $A \leftarrow r$ -cliques in the bucket  $B_i$  in  $ND$  (to be peeled)
10:    finished  $\leftarrow$  finished +  $|A|$ 
11:    num_rounds  $\leftarrow$  num_rounds + 1
12:    parfor all  $r$ -cliques  $R$  in  $A$  do
13:      parfor all  $s$ -cliques  $S$  containing  $R$  do
14:        parfor all  $r$ -cliques  $R'$  in  $S$  where  $R' \neq R$  do
15:          Update  $s$ -clique count of  $R'$  in  $U$ 
16:    Peel  $A$  and update the buckets of  $r$ -cliques with updated  $s$ -clique counts
17:    if num_rounds  $\geq O(\log_{1+\delta/\binom{s}{r}}(n))$  or  $B_i$  is empty then
18:      Add the remaining  $r$ -cliques in  $B_i$  (if it is non-empty) to  $B_{i+1}$ 
19:       $i \leftarrow i + 1$ 
20:    num_rounds  $\leftarrow 0$ 
21:  return  $ND$ 

```

Theoretical Guarantees and Efficiency. We now discuss the theoretical guarantees and theoretical efficiency of APPROX-ARB-NUCLEUS, and by extension, ARB-APPROX-NUCLEUS-HIERARCHY.

We introduce the following lemmas to help prove that APPROX-ARB-NUCLEUS guarantees a $(\binom{s}{r} + \varepsilon)$ -approximation of the true (r, s) -clique core numbers of each r -clique.

In particular, Lemma 11.1 bounds the number of r -cliques with (r, s) -clique core numbers $\leq \ell$ for a fixed ℓ . We use this to prove Lemma 11.2, which bounds the proportion of r -cliques with core numbers $\leq \ell$, but with s -clique-degree $> \ell(\binom{s}{r} + \delta)$. We can then set ℓ such that at any given step of our peeling process, we obtain a bound on the maximum proportion of r -cliques with core number at most ℓ that is not within the current bucket to be peeled. In essence, this gives us a bound on the number of times that a bucket must be reprocessed, such that moving on to the next bucket does not degrade the approximation factor, which gives us our approximation guarantees.

Note that Lemmas 11.1 and 11.2 apply to any stage of the peeling process in APPROX-ARB-NUCLEUS; that is to say, even if we have peeled a set of r -cliques from the graph G , the lemmas hold true for the remaining unpeeled r -cliques in G , with the

updated s -clique-degrees (which are the original s -clique-degrees minus the number of incident s -cliques that have been removed from the graph due to the set of peeled r -cliques).

Lemma 11.1. *Let S_ℓ be the set of remaining, or unpeeled, r -cliques with (r, s) -clique core numbers $\leq \ell$, considering an arbitrary fixed stage in the peeling process. Then, the number of s -cliques incident to S_ℓ is $\leq \ell \cdot |S_\ell|$.*

Proof. We prove this by considering the exact (r, s) nucleus decomposition algorithm given by Sariyüce *et al.* [284]. In their algorithm, at each step, an r -clique with the minimum s -clique-degree in the graph is peeled. In peeling an r -clique R , for every s -clique S that R participated in, the algorithm decrements the s -clique-degree of the remaining r -cliques $R' \in S$ that have not yet been peeled. The (r, s) -clique core number of each r -clique R is given by the maximum k_R such that at some point before R was peeled in this algorithm, all r -cliques that were not yet peeled had s -clique-degree at least k_R . In order for a given r -clique R to have (r, s) -clique core number k_R , it must have had s -clique-degree at most k_R when it was peeled from the graph. If R has s -clique-degree greater than k_R when it was peeled, since R must have had the minimum s -clique-degree when it was peeled, this would mean that before R was peeled, every r -clique had s -clique-degree greater than k_R , so by definition, R 's core number must be greater than k_R , which is a contradiction.

Note that Sariyüce *et al.* [284] prove that considering the r -cliques in the order in which they are peeled, the (r, s) -clique core numbers of the r -cliques are monotonically increasing.

Using these observations, the set S_ℓ of r -cliques is given by a contiguous sequence of r -cliques peeled in this algorithm, where upon being peeled, each r -clique has induced s -clique-degree at most ℓ . As such, we can assign each s -clique S incident to S_ℓ to the r -clique R in which S contributed to R 's induced s -clique-degree when R was peeled. As a result, the total number of s -cliques incident to S_ℓ must be $\leq \ell \cdot |S_\ell|$, since the induced s -clique-degree of each such r -clique when it was peeled is upper bounded by ℓ . \square

Lemma 11.2. *Let F_ℓ be the set of unpeeled, or remaining, r -cliques with s -clique-degree $> \ell(\binom{s}{r} + \delta)$ and with (r, s) -clique core number $\leq \ell$, considering an arbitrary fixed stage in the peeling process. Then, $|F_\ell| \leq \binom{s}{r} \cdot |S_\ell| / (\ell(\binom{s}{r} + \delta))$.*

Proof. Let S_ℓ be the set of unpeeled r -cliques with (r, s) -clique core numbers $\leq \ell$. First, note that $F_\ell \subseteq S_\ell$ by definition. By Lemma 11.1, we know that at most $\ell \cdot |S_\ell|$ s -cliques are incident to S_ℓ . Thus, the sum of the s -clique-degrees of the r -cliques in S_ℓ is at most $\binom{s}{r} \cdot \ell \cdot |S_\ell|$, since each s -clique can contribute to the s -clique-degree of at most $\binom{s}{r}$ r -cliques. Then, since each r -clique in F_ℓ has s -clique-degree $> \ell(\binom{s}{r} + \delta)$ by definition, the maximum number of r -cliques in F_ℓ is $\binom{s}{r} \cdot \ell \cdot |S_\ell| / (\ell(\binom{s}{r} + \delta)) = \binom{s}{r} \cdot |S_\ell| / (\binom{s}{r} + \delta)$, as desired. \square

Theorem 11.2. ARB-APPROX-NUCLEUS-HIERARCHY computes a $(\binom{s}{r} + \varepsilon)$ -approximate (r, s) nucleus decomposition hierarchy in $O(m\alpha^{s-2})$ expected work and $O(\log^3 n)$ span w.h.p.

Proof. Note that the work bound for ARB-APPROX-NUCLEUS-HIERARCHY follows directly from that for ARB-NUCLEUS [297] and ARB-NUCLEUS-HIERARCHY. This is because the only difference between APPROX-ARB-NUCLEUS and ARB-NUCLEUS is the boundaries used in the bucketing structure, which affects the number of r -cliques that are peeled in any given round, but does not affect the amount of work needed to rediscover s -cliques containing peeled r -cliques. The latter dominates the work of both APPROX-ARB-NUCLEUS and ARB-NUCLEUS, so as a result, the work of APPROX-ARB-NUCLEUS is precisely the work of ARB-NUCLEUS. To be more specific, as given by [297], APPROX-ARB-NUCLEUS takes $O(m\alpha^{s-2})$ work w.h.p. assuming space proportional to the number of s -cliques.¹ Additionally, ARB-APPROX-NUCLEUS-HIERARCHY is precisely ARB-NUCLEUS-HIERARCHY, except using APPROX-ARB-NUCLEUS to compute the (r, s) -clique core numbers. Thus, following the proof of Theorem 11.1, we see that ARB-APPROX-NUCLEUS-HIERARCHY overall takes $O(m\alpha^{s-2})$ expected work w.h.p.

For our span bound, first note that there are only a logarithmic number of possible i values (since $\binom{n}{s} = O(n^s)$ upper bounds the maximum possible s -clique-degree). For each bucket B_i , we by construction process B_i at most $O(\log_{1+\delta}/\binom{s}{r}(n))$ times (Line 17). Thus, in total, we have $O(\log^2 n)$ rounds of peeling in APPROX-ARB-NUCLEUS. The span of each peeling round is $O(\log n)$ w.h.p. to retrieve the next bucket of s -cliques and to perform hash table operations to update the s -clique counts, as discussed in more detail by Shi *et al.* [297]. Therefore, the total span of peeling is $O(\log^3 n)$ span w.h.p.

The work and span of orienting the graph and counting the number of s -cliques follows from Shi *et al.*'s s -clique enumeration algorithm [296], which takes $O(m\alpha^{s-2})$ expected work and $O(\log^2 n)$ span w.h.p. Thus, in total, ARB-APPROX-NUCLEUS-HIERARCHY takes $O(m\alpha^{s-2})$ expected work and $O(\log^3 n)$ span w.h.p., as desired.

It remains to argue that APPROX-ARB-NUCLEUS computes an $(\binom{s}{r} + \varepsilon)$ -approximate (r, s) -clique core number for every r -clique. Our proof here follows arguments by Ghaffari *et al.* [144], in their approximate k -core decomposition algorithm. We generalize their arguments to apply to (r, s) nucleus decomposition.

We prove this using induction on i , considering each bucket B_i . Our inductive hypothesis is that before beginning to peel B_i , we have already peeled all r -cliques with (r, s) -clique core numbers $\leq (1 + \delta)^i$. We show here that after a logarithmic number of rounds peeling all r -cliques in B_i , we have necessarily peeled all r -cliques with (r, s) -clique core numbers $\leq (1 + \delta)^{i+1}$. The key to this argument is Lemma 11.2, where we set $\ell = (1 + \delta)^{i+1}$. Notably, all r -cliques in B_i have s -clique-degree at most $(\binom{s}{r} + \delta) \cdot (1 + \delta)^{i+1} = (\binom{s}{r} + \delta) \cdot \ell$, by construction of B_i . Thus, Lemma 11.2 bounds the number of unpeeled r -cliques outside of the bucket B_i (that is to say, in a bucket B_j where $j > i$), but with (r, s) -clique core number $\leq \ell$ (these r -cliques are precisely those in F_ℓ). Specifically, after each round of peeling B_i , at most $\binom{s}{r} \cdot |S_\ell| / (\binom{s}{r} + \delta)$

¹ As an aside, following the arguments in [297], if we instead restrict our space usage to be proportional to the number of r -cliques, we can modify the bucketing structure to use a batch-parallel Fibonacci heap [298], which would increase the work bound to $O(m\alpha^{s-2} + \log^3 n)$ amortized expected work w.h.p.

r -cliques remaining outside of B_i have core number $\leq \ell$, so it takes $O(\log_{1+\delta/\binom{s}{r}}(n))$ rounds until no r -cliques with core number $\leq \ell$ are outside of B_i (that is to say, F_ℓ is empty). This means that after a logarithmic number of rounds of peeling all r -cliques in B_i , we have necessarily peeled all r -cliques with core number $\leq \ell$.

Now, in our algorithm, we assign the approximate core number of these peeled r -cliques to be the upper boundary of B_i , or $(\binom{s}{r} + \delta) \cdot (1 + \delta)^{i+1}$. Note that the core numbers of the r -cliques that we peel while processing B_i are $> (1 + \delta)^i$ (by our inductive hypothesis) and $\leq \ell = (1 + \delta)^{i+1}$. Thus, our approximation is within a $(\binom{s}{r} + \delta) \cdot (1 + \delta) = (\binom{s}{r} + \varepsilon)$ factor of the true core number. Thus, APPROX-ARB-NUCLEUS gives a $(\binom{s}{r} + \varepsilon)$ -approximation of the true core numbers of each r -clique.

□

11.6 Practical Implementations

While the algorithm presented in Section 11.4 (Algorithm 11.1) is efficient in theory, we present a number of optimizations that improve its practical performance. Algorithm 11.1 requires two passes over the r -cliques and their s -clique-adjacent neighbors, first to compute the (r, s) -clique core numbers and then to construct the hierarchy. We present algorithms that interleave these two computations, so that only a single pass is required. Specifically, we present two algorithms that are not as theoretically efficient as Algorithm 11.1, but are faster in practice, particularly when the difference between r and s is large, as we demonstrate in Section 11.7.2.

11.6.1 Interleaved Hierarchy Framework

Our algorithms use the same framework, given in Algorithm 11.3, and the main difference between the two algorithms is the implementation of the key subroutines `LINK` and `CONSTRUCT-TREE`. The framework is based on the peeling process used to compute the (r, s) -clique core numbers in Shi *et al.*'s work [297]. The main idea is that when we peel an r -clique R , while computing the updated s -clique counts due to peeling R , we are already iterating over all s -clique-adjacent r -cliques R' . Note that additionally, the nucleus decomposition algorithm in Shi *et al.* [297] uses a bucketing structure that maintains the intermediate (r, s) -clique core numbers of each r -clique throughout the peeling process (which begin as simply the s -clique count of each r -clique, and throughout the peeling process are updated to the actual (r, s) -clique core numbers). Then, if the intermediate (r, s) -clique core number of R' is less than or equal to that of R , as maintained in the bucketing structure, the intermediate (r, s) -clique core numbers of R and R' are actually the final (r, s) -clique core numbers of R and R' respectively. Thus, each such R and R' pair are connected in the nucleus decomposition hierarchy up to the level given by $\min(ND[R], ND[R'])$, and after the peeling process completes, we will have all of the relevant connectivity information to completely compute the nucleus decomposition hierarchy. In this sense, it suffices to define a `LINK` subroutine to process the s -clique-adjacent r -cliques

Algorithm 11.3 – Parallel (r, s) nucleus decomposition hierarchy algorithm framework

```

1: Initialize  $r, s$                                  $\triangleright r$  and  $s$  for  $(r, s)$  nucleus decomposition
2: procedure ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK( $G = (V, E)$ )
3:    $DG \leftarrow$  ARB-ORIENT( $G$ )            $\triangleright$  Apply an arboricity-orientation algorithm
4:   Initialize  $U$  to be a parallel hash table with  $r$ -cliques as keys, and  $s$ -clique counts as
   values
5:   REC-LIST-CLIQUEs( $DG, s, U$ )     $\triangleright$  Count  $s$ -cliques, and store the counts per  $r$ -clique
   in  $U$ 
6:   Let  $ND$  be a bucketing structure mapping each  $r$ -clique to a bucket based on # of
    $s$ -cliques
7:   finished  $\leftarrow 0$ 
8:   while finished  $< |U|$  do
9:      $A \leftarrow r$ -cliques in the next bucket in  $ND$  (to be peeled)
10:    finished  $\leftarrow$  finished +  $|A|$ 
11:    parfor all  $r$ -cliques  $R$  in  $A$  do
12:      parfor all  $s$ -cliques  $S$  containing  $R$  do
13:        parfor all  $r$ -cliques  $R'$  in  $S$  where  $R' \neq R$  do
14:          if  $ND[R'] \leq ND[R]$  then
15:            LINK( $R', R, ND$ )
16:          else Update  $s$ -clique count of  $R'$  in  $U$ 
17:    Update the buckets of  $r$ -cliques with updated  $s$ -clique counts, peeling  $A$ 
18:    return CONSTRUCT-TREE( $ND$ )   $\triangleright$  Return the hierarchy tree  $T$ , constructed based
   on LINK

```

given the intermediate (r, s) -clique core numbers throughout the peeling algorithm. Based on LINK, CONSTRUCT-TREE constructs the final hierarchy tree.

In more detail, ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK first uses an efficient low out-degree orientation algorithm by Shi *et al.* [296] to direct the graph G such that every vertex has out-degree at most $O(\alpha)$ (Line 3). Then, it counts the number of s -cliques per r -clique in G and stores the counts in a parallel hash table U , where the keys are r -cliques and the values are the counts (Lines 4–5). Note that ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK uses a subroutine REC-LIST-CLIQUEs based on previous work by Shi *et al.* [296], to count the number of s -cliques per r -clique. Also, our algorithm initializes a parallel bucketing structure ND that maps r -cliques to buckets, initially based on their s -clique counts (Line 6). We use the bucketing structure by Dhulipala et al. [95]. This structure ND stores the aforementioned intermediate (r, s) -clique core numbers of each r -clique, and supports efficient operations to update buckets and return the lowest unpeeled bucket. Our algorithm then proceeds with a classic peeling paradigm, where while not all r -cliques have been peeled, it processes the r -cliques (that have not yet been peeled) incident to the lowest number of s -cliques and peels them from the graph (Lines 8–17). For a set A of r -cliques with the lowest number of incident s -cliques (Line 9), we iterate over all s -clique-adjacent r -cliques R' to each r -clique R in A (Lines 11–13).

Note that if $ND[R'] \leq ND[R]$, this means that R' was either previously peeled

Algorithm 11.4 – Basic link and tree construction.

```
1: Initialize  $k$  union-find data structures,  $uf_i$  for  $i \in [k]$ , where  $k$  is the maximum  $(r, s)$ -  
clique core number  
2: procedure LINK-BASIC( $R, Q, ND$ )  
3:   parfor  $i \in [\min(ND[R], ND[Q])]$  do  
4:      $uf_i.\text{unite}(R, Q)$   
5: procedure CONSTRUCT-TREE-BASIC( $ND$ )  
6:   Initialize the hierarchy tree  $T$  with leaves corresponding to each  $r$ -clique  
7:   for  $i \in \{k, k-1, \dots, 1\}$  do  
8:     parfor each connected component  $C = \{R_1, \dots, R_c\}$  in  $uf_i$  do  
9:       Construct a new parent in  $T$ , to be the parent of the roots of the leaf nodes  
       corresponding to each  $R_\ell$  (for  $\ell \in [c]$ )  
10:    return  $T$ 
```

or is currently being peeled (and is also in A); this is because at any given peeling step, we process the bucket of unpeeled r -cliques with the minimum incident s -clique count. This also means that $ND[R']$ and $ND[R]$ are the actual (r, s) -clique core numbers of R' and R respectively, which follows directly from the correctness of the peeling paradigm [284]. Thus, each such R and R' pair are connected in the nucleus decomposition hierarchy up to the level given by $\min(ND[R], ND[R'])$, which we process using the LINK subroutine (Lines 14–15). The LINK subroutine will construct the hierarchy, and we describe it in Sections 11.6.2 and 11.6.3.

On the other hand, if $ND[R] > ND[R']$, then this means that R' has not yet been peeled, and the s -clique removed by R must be properly accounted for. In this case, ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK updates the s -clique count of R' in the hash table U (Line 16). After processing all R and R' pairs, we then update the buckets of the r -cliques with updated s -clique counts in U (Line 17). We omit the details of these steps for conciseness, since they are described in Shi *et al.*'s parallel nucleus decomposition algorithm [297].

11.6.2 Basic Version of LINK

The salient detail that remains is how we perform the LINK subroutine (Line 15) and how we construct the hierarchy tree T with CONSTRUCT-TREE (Line 18). In Algorithm 11.4, we present a basic LINK subroutine, LINK-BASIC, and the corresponding CONSTRUCT-TREE subroutine, CONSTRUCT-TREE-BASIC. LINK-BASIC maintains a parallel union-find data structure uf_i per core number $i \in [k]$, which corresponds to a level of the hierarchy tree T . Each uf_i connects r -cliques that are s -clique-adjacent considering only r -cliques with core numbers $\geq i$. To construct these uf_i 's, given two r -cliques R and Q , LINK-BASIC simply unites R and Q in each uf_i where $i \leq \max(ND[R], ND[Q])$ (Lines 3–4). Then, given the uf_i for all $i \in [k]$, we construct the hierarchy tree T from the bottom-up, starting with leaf nodes corresponding to r -cliques. CONSTRUCT-TREE-BASIC begins with $i = k$, where for each connected component in uf_k (Line 8), we construct a parent in T where its children are the leaf

nodes corresponding to the r -cliques in the connected component (Line 9). Then, for $i = k - 1, \dots, 1$ (Line 7), we construct a new parent for each connected component in uf_i , where its children are the parents of the leaf nodes corresponding to the r -cliques that compose the component (Line 9). This produces the desired T .

However, LINK-BASIC is not efficient, since it requires a union-find data structure per level, and for every pair of r -cliques, we could perform up to k UNITE operations. Indeed, in Section 11.7.2, we empirically show that LINK-BASIC performs many unnecessary UNITE operations in practice. If we let n_r and n_s denote the number of r -cliques and the number of s -cliques in the graph respectively, LINK-BASIC incurs additional space proportional to $O(kn_r)$ and total work upper bounded by $O(kn_s)$ (since there are at most $O(n_s)$ pairs of s -clique-adjacent r -cliques). In the next subsection, we introduce more efficient LINK and CONSTRUCT-TREE subroutines.

11.6.3 Efficient Version of LINK

Our improved subroutines LINK-EFFICIENT and CONSTRUCT-TREE-EFFICIENT are shown in Algorithm 11.5. We refer to an example of $(r, s) = (1, 2)$ nucleus (k -core) decomposition, given by the graph in Figure 11.1. Recall that we have omitted labeling other vertices in the 5-cliques represented by $4a$, $4b$, $4c$, and $4d$.

The main idea of LINK-EFFICIENT is instead of maintaining k union-find data structures, we maintain a single parallel union-find data structure uf and an additional hash table L that maps r -cliques to r -cliques. First, uf stores connected r -cliques considering only other r -cliques with equal core numbers. For instance, in Figure 11.1, the vertices $3a$, $3b$, and $3c$ are connected and all have core number 3, so we would store these as a component in uf . Note that for all core numbers i , we can store this information using a single union-find data structure because the sets of r -cliques with distinct core numbers are disjoint. We can arbitrarily represent each connected component in uf by a single r -clique in that component.

The main idea of L is to connect the components in uf to the “nearest” core with a different core number that it is contained within (if it exists). For instance, in Figure 11.1, we note that the component in uf corresponding to $4a$ (consisting of vertices with core number 4) is contained within the 3-core consisting of the component $\{3a, 3b, 3c\}$. In L , we would store one of $3a$, $3b$, or $3c$ in an entry corresponding to key $4a$, indicating that this is the “nearest” core that $4a$ must join in the hierarchy. We note that $4a$ is also contained within a larger 2-core and a larger 1-core, but its “nearest” core, or the smallest core such that the component $4a$ is a proper subset of that core, is given by the 3-core. It is sufficient to store only the “nearest” core to $4a$ in L , because the component in uf corresponding to $\{3a, 3b, 3c\}$ is responsible for storing its “nearest” core in L as well, to the 2-core it is contained within. On the other hand, the component corresponding to $4d$ is not contained within the 3-core, and its “nearest” core would be the 2-core containing the component $2a$, so $4d$ would store in L the value $2a$.

More formally, for each r -clique R representing a connected component in uf , let R' be a r -clique with the maximum $ND[R']$ such that $ND[R'] < ND[R]$, and R' is connected to R through s -cliques considering only r -cliques with core number

Algorithm 11.5 – Efficient link and tree construction.

```

1: Initialize a union-find data structure,  $uf$ , of length equal to the number of  $r$ -cliques
2: Initialize a hash table,  $L$ , where the keys and values are  $r$ -cliques
3: procedure LINK-EFFICIENT( $R, Q, ND$ )
4:   if  $R$  or  $Q$  is empty then return
5:   if  $ND[Q] < ND[R]$  then Swap  $R$  and  $Q$ 
6:    $R \leftarrow uf.parent(R)$ ,  $Q \leftarrow uf.parent(Q)$ 
7:   if  $ND[R] = ND[Q]$  then
8:      $uf.unite(R, Q)$ 
9:     if  $uf.parent(R) \neq R$  then LINK-EFFICIENT( $L[R], uf.parent(R), ND$ )
10:    if  $uf.parent(Q) \neq Q$  then LINK-EFFICIENT( $L[Q], uf.parent(Q), ND$ )
11:   else ▷  $ND[R] < ND[Q]$ 
12:     while true do
13:        $LQ \leftarrow L[Q]$ 
14:        $Q \leftarrow uf.parent(Q)$ 
15:       if COMPARE-AND-SWAP( $L[Q]$ , empty,  $R$ ) then
16:         if  $uf.parent(Q) \neq Q$  then
17:           LINK-EFFICIENT( $R, uf.parent(Q), ND$ )
18:         break
19:       else if  $ND[LQ] < ND[R]$  then
20:         if COMPARE-AND-SWAP( $L[Q]$ ,  $LQ$ ,  $R$ ) then
21:           if  $uf.parent(Q) \neq Q$  then
22:             LINK-EFFICIENT( $R, uf.parent(Q), ND$ )
23:             LINK-EFFICIENT( $R, LQ, ND$ )
24:             break
25:       else
26:         LINK-EFFICIENT( $R, L[Q], ND$ )
27:         break
28: procedure CONSTRUCT-TREE-EFFICIENT( $ND$ )
29:   Initialize the hierarchy tree  $T$  with leaves corresponding to each  $r$ -clique
30:   parfor each connected component  $\mathcal{C} = \{R_1, \dots, R_c\}$  in  $uf$  do
31:     Construct a parent node  $uf_{\mathcal{C}}$  in  $T$ , where its children are the leaves corresponding
      to the  $r$ -cliques in  $\mathcal{C}$ 
32:     parfor each parent node  $uf_{\mathcal{C}}$  in  $T$  do
33:       if  $L[\mathcal{C}]$  is non-empty then
34:          $R \leftarrow uf.parent(L[\mathcal{C}])$  ▷ Note that  $\mathcal{C}$  is the  $r$ -clique representing the
          component in  $uf$ 
35:         Make  $uf_{\mathcal{C}}$  a child of  $uf_R$  in  $T$  ▷ Note that  $uf_R$  necessarily exists since  $R$ 
          represents a component in  $uf$ 
36:   return  $T$ 

```

$\geq ND[R']$. Then, each such r -clique R is a key in L , and L stores the corresponding R' that satisfies these conditions as the value. Note that if there are multiple such R' where $ND[R']$ is maximized under these conditions, it is irrelevant which R' is stored in L , because it is simple to look up the component that R' corresponds to using uf .

After Round 3:
 $A = \{3a, 3b, 3c\}$

| | | | | | | | | | | | |
|--------------------------|-----------------|----------------------|----|----|----|----|----|----|----|----|--|
| $uf =$ | r-clique | 1a | 2a | 3a | 3b | 3c | 4a | 4b | 4c | 4d | |
| | parent | 1a | 2a | 3a | 3b | 3c | 4a | 4b | 4c | 4d | |
| $L =$ | key | 3a | | | | | | | | | |
| | value | 1a | | | | | | | | | |
| After (3a, 4c): | | | | | | | | | | | |
| $uf =$ | r-clique | 1a | 2a | 3a | 3b | 3c | 4a | 4b | 4c | 4d | |
| | parent | 1a | 2a | 3a | 3b | 3c | 4a | 4b | 4c | 4d | |
| $L =$ | key | 3a 4c | | | | | | | | | |
| | value | 1a 3a | | | | | | | | | |
| After (3b, 4c): | | | | | | | | | | | |
| $uf =$ | r-clique | 1a | 2a | 3a | 3b | 3c | 4a | 4b | 4c | 4d | |
| | parent | 1a | 2a | 3b | 3b | 3c | 4a | 4b | 4c | 4d | |
| $L =$ | key | 3a 3b 4c | | | | | | | | | |
| | value | 1a 1a 3a | | | | | | | | | |
| After (2a, 4c): | | | | | | | | | | | |
| $uf =$ | r-clique | 1a | 2a | 3a | 3b | 3c | 4a | 4b | 4c | 4d | |
| | parent | 1a | 2a | 3b | 3b | 3c | 4a | 4b | 4c | 4d | |
| $L =$ | key | 2a 3a 3b 4c | | | | | | | | | |
| | value | 1a 1a 2a 3a | | | | | | | | | |
| After Round 4: | | | | | | | | | | | |
| $A = \{4a, 4b, 4c, 4d\}$ | | | | | | | | | | | |
| $uf =$ | r-clique | 1a | 2a | 3a | 3b | 3c | 4a | 4b | 4c | 4d | |
| | parent | 1a | 2a | 3b | 3b | 3b | 4a | 4b | 4c | 4d | |
| $L =$ | key | 2a 3a 3b 4a 4b 4c 4d | | | | | | | | | |
| | value | 1a 1a 2a 3a 3b 3b 2a | | | | | | | | | |

Figure 11.4: An example of the uf and L data structures maintained by LINK-EFFICIENT when computing the k -core hierarchy on the graph in Figure 11.1. The data structures are shown after the third and fourth rounds of peeling in ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK, and after intermediate calls to LINK-EFFICIENT within the fourth round.

That is to say, it is irrelevant which of $3a$, $3b$, and $3c$ we store for $4a$ in L , because we can look up the parent of $3a$, $3b$, and $3c$ in uf , which would resolve to the same parent. LINK-EFFICIENT updates uf and L , depending on the core numbers of the given r -cliques R and Q .

Tree Construction. We describe first how CONSTRUCT-TREE-EFFICIENT constructs the hierarchy tree T given the specifications for uf and L . We refer to the construction shown in Figure 11.5, where uf and L are given under “After Round 4” in Figure 11.4.

The main idea for the construction is that if we begin with a hierarchy tree T consisting of only leaf nodes corresponding to each r -clique, the highest (bottom-most) level in which that leaf node may join a non-trivial connected component is the level corresponding to the leaf node’s r -clique’s (r, s) -clique core number. That is to say, starting from $i = k$ and iterating to $i = 1$, an r -clique R is necessarily a singleton component from $i = k$ to $i = ND[R] + 1$. The union-find structure uf denotes the

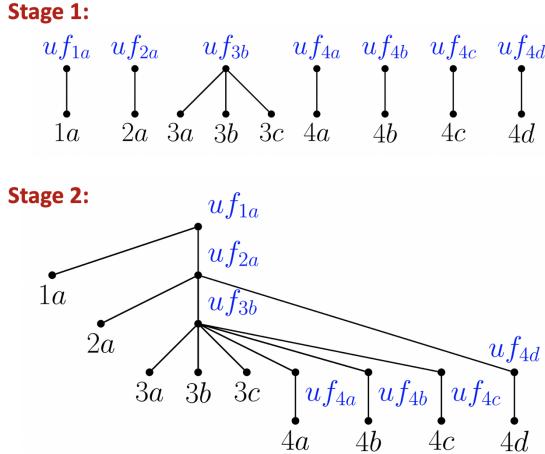


Figure 11.5: An example of the k -core hierarchy tree on the graph in Figure 11.1, constructed by CONSTRUCT-TREE-EFFICIENT. Stage 1 shows an intermediate tree constructed in the CONSTRUCT-TREE-EFFICIENT subroutine, and Stage 2 depicts the final hierarchy tree.

parent of each leaf node on the level corresponding to its core number.

Then, once we have the connected components per level corresponding to each core number, it remains to describe how components with different core numbers are contained within each other. This containment is necessarily hierarchical, where components corresponding to larger core numbers are contained within successive components corresponding to smaller core numbers, as can be seen in Figure 11.1. This containment is precisely what L describes; it points each component within a given core i to a component on the greatest core number j such that $j < i$, where the two components are connected on level j (that is to say, connected through r -cliques with core numbers $\geq j$). Thus, using L , we can point each parent constructed using our uf to another parent, which represents the first instance (from the bottom-up in T) in which the original parent is merged into another component.

In more detail, in CONSTRUCT-TREE-EFFICIENT, we begin with a hierarchy tree T consisting of isolated r -cliques (Line 29). For each component $C = \{R_1, \dots, R_c\}$ in uf , representing a connected component of r -cliques with the same core number, we construct a parent in T , where its children are leaves corresponding to the r -cliques in C (Lines 30–31). C is one of these cliques, and is chosen arbitrarily. The new parent is uf_C . This process is shown in Stage 1 in Figure 11.5; each vertex has itself as its parent in uf , except the vertices $3a, 3b$, and $3c$, which have $3b$ as their parent. Thus, we create a new parent node uf_{3b} for the leaves $3a, 3b$, and $3c$.

Then, for each parent node uf_C , we look up the nearest connected component that it should hierarchically connect to using L . What this means is that the component uf_C should join the connected component of $L[C]$ (if it exists). If $R = uf.parent(L[C])$, then the connected component of $L[C]$ is represented by uf_R . Thus, we make uf_C a child of uf_R in T (Lines 32–35). This construction is shown in Stage 2 in our example in Figure 11.5. We note that $L[2a] = 1a$ and $uf[1a] = 1a$, so we make uf_{2a} the child

of uf_{1a} . Similarly, $L[3b] = 2a$ and $uf[2a] = 2a$, so we make uf_{3b} the child of uf_{2a} , and $L[4d] = 2a$ and $uf[2a] = 2a$, so we also make uf_{4d} the child of uf_{2a} . Finally, we have $L[4a] = 3a$, $L[4b] = 3b$, and $L[4c] = 3b$, where $uf[3a] = uf[3b] = 3b$, so we make uf_{4a} , uf_{4b} , and uf_{4c} the children of uf_{3b} . In this manner, we construct T .

We make a subtle note that it is not strictly necessary to maintain parent nodes in the hierarchy tree with exactly one child; we can remove all such parents P and make the child C a direct child of its grandparent G . This is because it is inherently implied that the component that the child C represents in unchanged until it joins the component represented by G . For instance, uf_{4d} 's sole child is $4d$. If we set $4d$ to be a direct child of its grandparent, uf_{2a} , and remove uf_{4d} , then it is implied that the vertex $4d$ remains its own component until it joins the 2-core containing $2a$ (and notably, $4d$ represents an unchanged component in the 3-core and in the 4-core of the graph). In this sense, the final hierarchy tree produced in Figure 11.5 is actually equivalent to that in Figure 11.3.

LINK Subroutine. We now describe LINK-EFFICIENT. The key to LINK-EFFICIENT is to properly maintain uf and L according to their definitions given new information obtained by connected r -cliques. The main subtlety is that after updating the connectivity information of R or Q , in uf or in L , there may be cascading effects resulting in new calls to LINK-EFFICIENT. In more detail, LINK-EFFICIENT first ensures that $ND[R] \leq ND[Q]$ (Line 5). We need only perform operations on the parents of the components of the r -cliques in uf , so we set R and Q to their respective parents in uf (Line 6).

The first case is if $ND[R] = ND[Q]$ (Line 7). Then, we need only unite R and Q in uf , to maintain that uf tracks the connectivity between r -cliques with the same core number (Line 8). However, the new parent P may now need to update its value in L based on $L[R]$ and $L[Q]$. This is because it is possible that $L[P]$ must be updated; for instance, if $L[R]$ has a strictly greater core number than the current $L[P]$, $L[P]$ must be updated to be $L[R]$. LINK-EFFICIENT performs these updates by calling itself on $L[R]$ and P , and on $L[Q]$ and P , if $P \neq R$ and $P \neq Q$, respectively (Lines 9–10).

The second case is if $ND[R] < ND[Q]$ (Line 11). There are two main considerations. First, R may replace the current value of $L[Q]$, which we call LQ , if $ND[R] > ND[LQ]$. Second, R and LQ must be linked, since they are connected through Q and could affect each other's parent or value in uf or L respectively. We handle these cases with a series of if statements and COMPARE-AND-SWAPS.

We first perform a COMPARE-AND-SWAP, checking if $L[Q]$ is empty and replacing it with R if so (Line 15); if this COMPARE-AND-SWAP succeeds, then there is still the possibility that Q 's parent in uf changed before the COMPARE-AND-SWAP completed, in which case Q 's new parent is unaware of its connection to R . In this scenario, we must call LINK-EFFICIENT again on R and the new parent of Q , since R could potentially modify the r -clique stored in L corresponding to Q 's new parent (Lines 16–17).

If the previous COMPARE-AND-SWAP failed, then we check if $ND[LQ] < ND[R]$ (Line 19), which if true, means that R is a candidate to replace LQ in L . We perform another COMPARE-AND-SWAP to replace LQ with R (Line 20), and if it succeeds, we

must again check if Q 's parent in uf has potentially changed before the COMPARE-AND-SWAP completed. Again, if this occurs, we must call LINK-EFFICIENT on R and the new parent Q (Lines 21–22). We must also call LINK-EFFICIENT on R and LQ (Line 23), to store R 's connectivity to LQ . This is because R 's “nearest” core as stored in L may be superseded by LQ . If the COMPARE-AND-SWAP fails, we simply try again, hence the while loop (Line 12).

The last case is if $ND[LQ] \geq ND[R]$ (Line 25), in which case R is not a candidate to replace LQ in L , and we store R 's connectivity to Q by calling LINK-EFFICIENT on R and $L[Q]$. Note that this is necessary because R and $L[Q]$ could be united in uf if they have the same core number, or R could be a “nearest” core to $L[Q]$.

This concludes our efficient LINK subroutine, LINK-EFFICIENT. We show in Section 11.7.2 that LINK-EFFICIENT performs many fewer UNITE and LINK operations, and achieves significant speedups, over LINK-BASIC. We provide an example of running LINK-EFFICIENT.

Example of LINK-EFFICIENT. As an example of these cascading effects, we refer to an intermediate state of uf and L on the example graph in Figure 11.1, given after the third round of peeling in ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK (immediately before peeling the final set of vertices in the graph, given by the components 4a, 4b, 4c, and 4d). This intermediate state is shown in the data structures under “After Round 3” in Figure 11.4. Importantly, in uf , every vertex is its own parent, and in L , we have identified that the component corresponding to 3a is connected to 1a. Note that many link operations occur from peeling 4a, 4b, 4c, and 4d, including for the (R, Q) pairs $(3a, 4c)$, $(3b, 4c)$, and $(2a, 4c)$, which we consider in this example. We show in Figure 11.4 the state of uf and L after each of these three calls to LINK-EFFICIENT (including the cascading calls that each of these calls generate). We list the R and Q for each LINK-EFFICIENT operation, as well as the cascading calls that they invoke. We assume the operations happen sequentially for clarity, although in practice they can happen in parallel.

- $R = 3a, Q = 4c$: We now know that 4c is connected to 3a's component, and 4c does not have a previously set “nearest” core, so we set $L[4c] = 3a$ (Line 15).
- $R = 3b, Q = 4c$: We note that $L[4c]$ is now already set to a “nearest” core with core number 3, so there is nothing new to update for $L[Q]$. However, we gain from this new link the knowledge that 3a and 3b are connected (since $L[4c] = 3a$), so we must cascade a new LINK-EFFICIENT call to $(R, L[Q]) = (3b, 3a)$ (Line 26), so that 3a and 3b can be set to the same component in uf .
 - $R = 3a, Q = 3b$: We now call UNITE on 3a and 3b in uf (Line 8). Say that arbitrarily, 3b is set as the new parent of 3a and 3b in uf . Since the parents in uf are responsible for maintaining the connection to the “nearest” core, we must transfer $L[R] = L[3a]$ to $L[Q] = L[3b]$. We do so by calling LINK-EFFICIENT on $(L[R], uf.parent(R)) = (1a, 3b)$ (Line 9).
 - * $R = 1a, Q = 3b$: Now, we can set $L[Q] = L[3b]$ to $R = 1a$, since 3b's “nearest” core is now 1a (Line 15).

- $R = 2a, Q = 4c$: Since $4c$ already has an entry in L that's "nearer" to it than $2a$, there is nothing new to update for $L[Q]$. However, we now know that $3a$ and $2a$ are connected, so we must cascade a new LINK-EFFICIENT call to $(R, L[Q]) = (2a, 3a)$ (Line 26).
 - $R = 2a, Q = 3a$: Since the parent of Q in uf is $3b$, we can treat this as $R = 2a$ and $Q = 3b$ (since we only need to maintain connections in L for the parents in uf). Now, we find that $3b$ has recorded its "nearest" core as $L[3b] = 1a$, but $2a$ is "nearer". Thus, we update $L[3b]$ to be $2a$ (Line 20), but now we know that $2a$ is connected to $1a$. So, we call LINK-EFFICIENT on $(R, L[Q]) = (2a, 1a)$ (Line 23).
 - * $R = 1a, Q = 2a$: We discover that $2a$'s "nearest" core is given by $1a$. We set $L[Q] = L[2a]$ to $R = 1a$ (Line 15).

Note that in many ways, it is necessary for us to perform these cascading calls to LINK-EFFICIENT, because the only way for $3a$ and $3b$ to discover that they should be connected is through one of the 4-core components, and the only way for $3b$ to realize that the component with $2a$ is its "nearest" core is also through one of the 4-core components. Similarly, the only way for $2a$ to realize that the component with $1a$ is its "nearest" core is through first one of the 4-core components, then through the 3-core component, in which $3a$, which we now know is connected to $3b$, had the original adjacency to $1a$. In this sense, information must be constantly propagated through uf and L , even considering only one round of peeling.

Comparison to Prior Work. Sariyüce and Pinar [281] also provide a hierarchy construction algorithm, NH, that is performed interleaved with the peeling process. They maintain a union-find data structure that stores the connectivity of all r -cliques considering only r -cliques with the same core number. However, for adjacent r -cliques with different core numbers, they simply store all pairs of such r -cliques in a list. They process this list after the peeling process to construct the hierarchy tree, and their method for processing this list requires a global view, since NH first sorts the pairs of r -cliques in the list based on their core numbers. Storing this list incurs additional space potentially proportional to the number of s -cliques in the graph, which represents a significant overhead.

Our main innovation in LINK-EFFICIENT is that we need only incur additional space overhead proportional to the number of r -cliques in the graph, because we process adjacent r -cliques with different core numbers while performing the peeling process. We are able to process this information into a hash table L proportional to the number of r -cliques, so our memory overhead overall is $2n_r$, where n_r is the number of r -cliques. In contrast, NH uses $\binom{s}{r} \cdot n_s + n_r$ additional space, where n_s is the number of s -cliques.

In addition, NH is sequential, whereas LINK-EFFICIENT is thread-safe and carefully resolves conflicts in updating uf and L . Also, the post-processing step to construct the hierarchy tree in NH involves many sequential dependencies, where even merges on the same level of the tree may conflict with each other, whereas our post-processing step, CONSTRUCT-TREE-EFFICIENT, is fully parallel.

| | n | m |
|--------------------|------------|---------------------|
| amazon | 334,863 | 925,872 |
| dblp | 317,080 | 1,049,866 |
| youtube | 1,134,890 | 2,987,624 |
| skitter | 1,696,415 | 11,095,298 |
| livejournal | 3,997,962 | 34,681,189 |
| orkut | 3,072,441 | 117,185,083 |
| friendster | 65,608,366 | 1.806×10^9 |

Table 11.1: Sizes of our input graphs, which are from SNAP [207].

11.6.4 Practical Version of ARB-NUCLEUS-HIERARCHY

Finally, we make certain modifications to our theoretically efficient (r, s) nucleus decomposition hierarchy algorithm, ARB-NUCLEUS-HIERARCHY (Algorithm 11.1), to improve its performance in practice. We maintain the two-pass paradigm of first computing the (r, s) -clique core numbers of each r -clique and then constructing the hierarchy tree T . However, we do not explicitly store linked lists containing all pairs of s -clique-adjacent r -cliques, since this represents too much of a memory overhead to be practical, particularly for larger r and s . We also do not explicitly generate the graph H given by these linked lists (Line 14).

Instead, we use a single union-find data structure to maintain the connected components (throughout the loop on Lines 12–20), and for each $i \in \{k, k - 1, \dots, 1\}$ (Line 12), we iterate through all r -cliques R with core number i and their s -clique-adjacent r -cliques R' . We perform a parallel sort on the r -cliques based on their core numbers, which allows us to efficiently extract r -cliques with the same core numbers; this adds a small additional memory overhead, which we observe in Section 11.7.2. For each such pair of r -cliques R and R' where $ND[R'] \geq ND[R]$, we directly unite them in our union-find data structure to obtain the desired connected components; this replicates the same information stored in the linked lists (on Lines 6–8). We construct the requisite new parents in the hierarchy tree T given the computed connected components, and we reuse the same union-find data structure for subsequent i .

11.7 Evaluation

11.7.1 Environment and Graph Inputs

We run our experiments on a Google Cloud Platform instance of a 30-core machine with two-way hyper-threading, with 3.9 GHz Intel Cascade Lake processors and 240 GB of main memory. We use all cores when testing parallel implementations, unless specified otherwise. Our implementations are written in C++ and we compile our code using g++ (version 7.4.0) with the `-O3` flag. We use parallel primitives and the work-stealing scheduler from PARLAYLIB by Blelloch *et al.* [46]. We terminate any

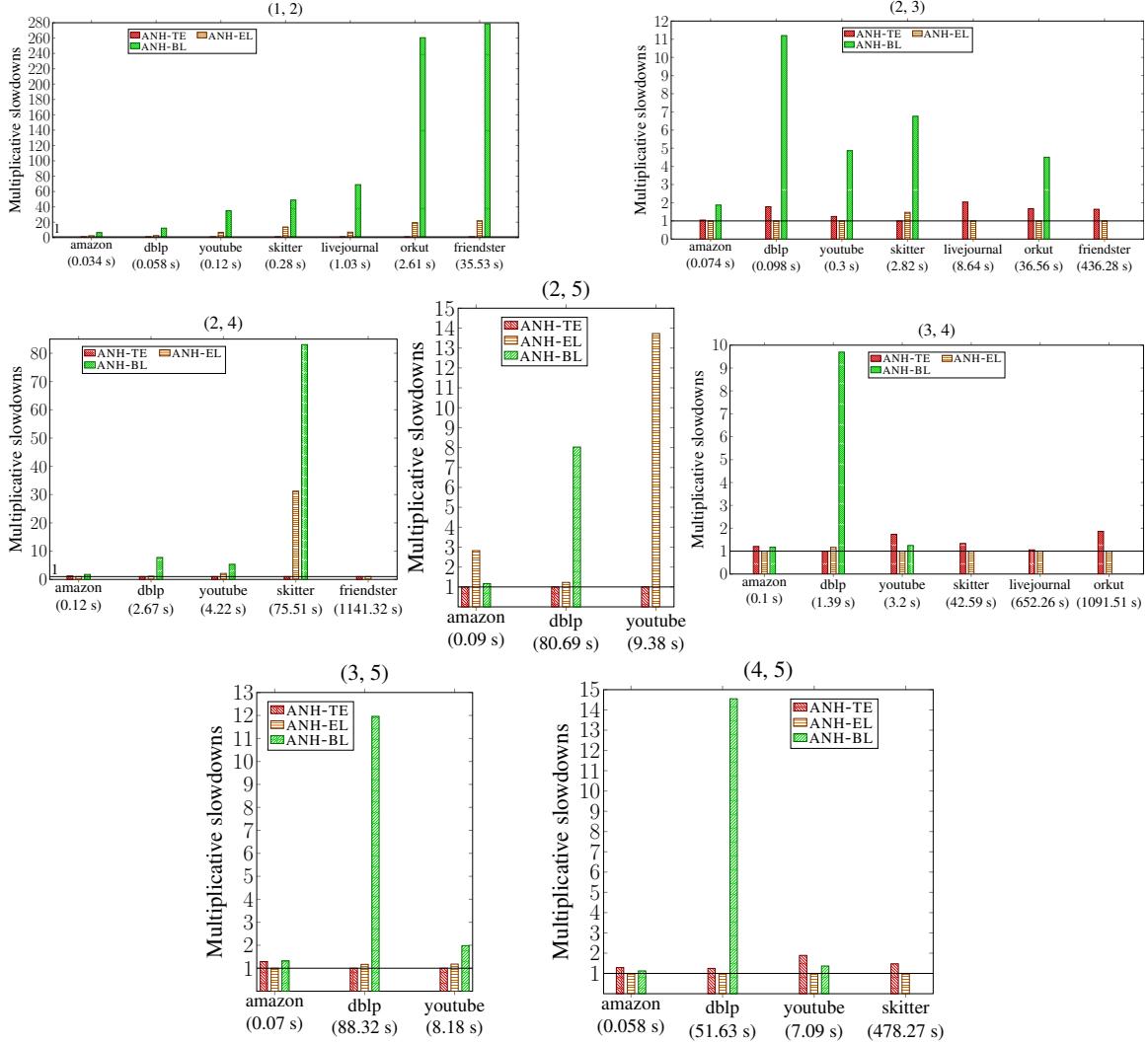


Figure 11.6: Multiplicative slowdowns of our parallel nucleus decomposition hierarchy implementations ANH-TE, ANH-EL, and ANH-BL, over the fastest of the three for each graph, for $r < s \leq 5$. We have omitted bars where our implementations run out of memory or time out after 4 hours, and we have omitted graphs where only one of ANH-TE, ANH-EL, and ANH-BL completes. Below each graph in parentheses is the fastest running time among the three implementations. We have also included a line marking a multiplicative slowdown of 1.

experiment that takes over 4 hours. We test our algorithms on real-world graphs from the Stanford Network Analysis Project (SNAP) [207], shown in Table 11.1.

We implement all three versions of our exact (r, s) nucleus decomposition hierarchy algorithms, including our theoretically-efficient ARB-NUCLEUS-HIERARCHY (Algorithm 11.1, with the practical modifications described in Section 11.6.4), which we call ANH-TE for succinctness, and our nucleus decomposition hierarchy framework ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK (Algorithm 11.3) using both LINK-BASIC (Algorithm 11.4) and LINK-EFFICIENT (Algorithm 11.5), which we call ANH-BL and ANH-EL, respectively.

We also implement our approximate (r, s) nucleus decomposition hierarchy algorithm, ARB-APPROX-NUCLEUS-HIERARCHY (using Algorithm 11.2). We integrate APPROX-ARB-NUCLEUS with each of ANH-TE, ANH-EL, and ANH-BL for the hierarchy construction, giving us three implementations, APPROX-ANH-TE, APPROX-ANH-EL, and APPROX-ANH-BL, respectively.

We compare our hierarchy algorithms to NH, the state-of-the-art sequential (r, s) nucleus decomposition hierarchy implementation by Sariyüce and Pinar [281] for $(1, 2)$, $(2, 3)$, and $(3, 4)$ nucleus decomposition. Note that NH does not generalize to other r and s values. NH, like ANH-BL and ANH-EL, constructs the hierarchy while computing the (r, s) -clique-core numbers of the graph. For the special case of k -core, we also compare to PHCD, the state-of-the-art parallel k -core hierarchy implementation by Chu *et al.* [76].

11.7.2 Comparison of ANH-TE, ANH-EL, and ANH-BL

Figure 11.6 compares our exact (r, s) nucleus decomposition hierarchy algorithms, ANH-TE, ANH-EL, and ANH-BL, against each other, for various r and s . We show only $r < s \leq 5$ here, but we ran all three of our algorithms for $r < s \leq 7$. Also, the running times listed in these figures do not include the time needed to compute the low out-degree orientation or to compute the initial s -clique-degrees of each r -clique, which are the same across all three of our algorithms since we use the prior s -clique counting subroutine from [296]. However, our running times do include the time required to compute the (r, s) -clique-core numbers of each r -clique, which notably for ANH-TE is given by ARB-NUCLEUS from [297].

Overall, we find that ANH-EL is faster if the difference between s and r is small (generally, if $s - r \leq 2$), and ANH-TE is faster in all other cases. The exception is for k -core (or $(1, 2)$ -nucleus decomposition), where ANH-TE is 2.38–21.95x faster than ANH-EL. This is because the k -core decomposition requires far lower overhead to compute the core numbers per vertex compared to higher r and s , since we need only maintain the degree of each vertex. The benefit of ANH-EL is due to the improved locality in iterating over and processing s -cliques once, rather than recomputing the s -cliques twice. This is not a benefit for k -core, because iterating over edges is a much simpler and cache-friendly pattern. Also, ANH-BL is significantly slower than both ANH-TE and ANH-EL, and runs out of memory for many values of r and s , since it has a much larger memory footprint from storing a union-find structure per core number. Overall, ANH-EL is up to 2.37x faster than ANH-TE, and ANH-TE is up to 41.55x faster

than ANH-EL, where we see the largest speedups in ANH-TE over ANH-EL when s is much larger than r . ANH-BL is up to 14.55x slower than ANH-EL, and up to 11.96x slower than ANH-TE.

The cases in which ANH-EL outperforms ANH-TE and vice versa, and the reason for the slowness of ANH-BL, is due to the number of LINK and UNITE operations. Indeed, for the dblp and youtube graphs, particularly for larger r and when the difference between r and s is small, the number of times in which ANH-TE calls LINK and UNITE is 1.08–13.67x the number of times in which ANH-EL calls LINK and UNITE. For smaller r and when the difference between r and s is large, ANH-EL calls LINK and UNITE between 1.02–18.94x more than ANH-TE. Looking at fixed r and increasing s , we observe that ANH-EL performs many more cascading calls to itself as s increases, since $\binom{s}{r}$ increases and ANH-EL more likely needs to connect two r -cliques across multiple levels of the hierarchy tree. ANH-BL is much slower than both ANH-TE and ANH-EL, performing up to 39.75x the number of LINK and UNITE calls. ANH-BL repeatedly performs UNITE operations equal to the core number of each r -clique, which can be redundant, since if two r -cliques are connected in a higher core, they are necessarily connected in the lower core.

In terms of memory usage, considering the memory overhead of building and constructing the hierarchy (not including the space required to store the graph or the (r, s) -clique-core numbers of each r -clique), on dblp and youtube, ANH-BL uses 1.53–10.03x the amount of overhead that ANH-EL uses, and ANH-TE uses 1.08–1.11x the amount of overhead that ANH-EL uses. We observe that ANH-EL is the most memory efficient overall, since it only maintains two arrays proportional to the number of r -cliques (uf and L). ANH-TE incurs almost the same memory overhead, with a minor additional cost attributed to maintaining r -cliques sorted by their core numbers (which we discuss in Section 11.6.4), and ANH-BL incurs much more overhead to maintain union-find data structures proportional to the number of r -cliques per core number.

11.7.3 Performance of Exact Hierarchy

Figure 11.7 shows the best running times for all graphs, over $r < s \leq 7$, considering all of our exact (r, s) nucleus decomposition hierarchy algorithms, ANH-TE, ANH-EL, and ANH-BL, excluding the time needed to compute our low out-degree orientation and the initial s -clique-degrees of each r -clique. In general, larger (r, s) values correspond to longer running times. However, some of the times for larger values of (r, s) are faster than for smaller values of (r, s) (especially on amazon) because the maximum coreness values for the larger values of (r, s) are small and the algorithms finish quickly.

Figure 11.8 shows the scalability of ANH-TE and ANH-EL over different numbers of threads on dblp and skitter, and we see good scalability overall. Across all of our graphs and for $r < s \leq 7$, we observe up to 24.75x self-relative speedups (and a median of 15.57x) for ANH-TE and up to 30.96x self-relative speedups (and a median of 14.13x) for ANH-EL. Generally, we observe greater self-relative speedups for larger r and s and for larger graphs.

Comparison to other implementations. Figure 11.9 shows the comparison of

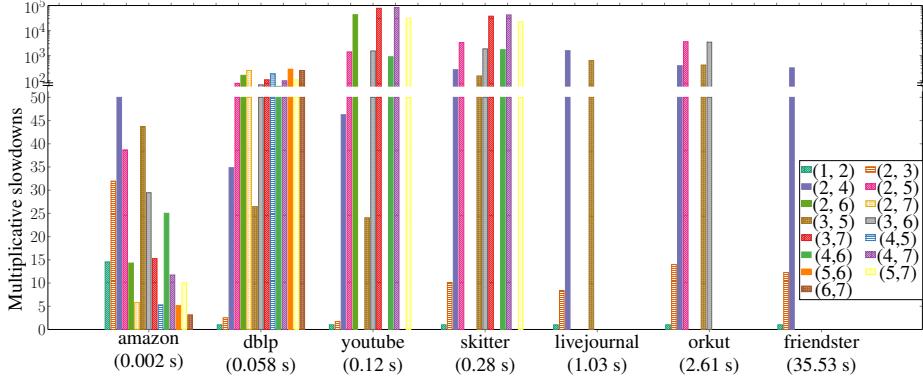


Figure 11.7: Multiplicative slowdowns of parallel ARB-NUCLEUS-HIERARCHY for each (r, s) combination over the fastest running time for parallel ARB-NUCLEUS-HIERARCHY across all $r < s \leq 7$ for each graph, considering the fastest of ANH-TE, ANH-EL, and ANH-BL. The fastest running time is labeled in parentheses below each graph. We have omitted bars where ARB-NUCLEUS-HIERARCHY runs out of memory or times out after 4 hours.

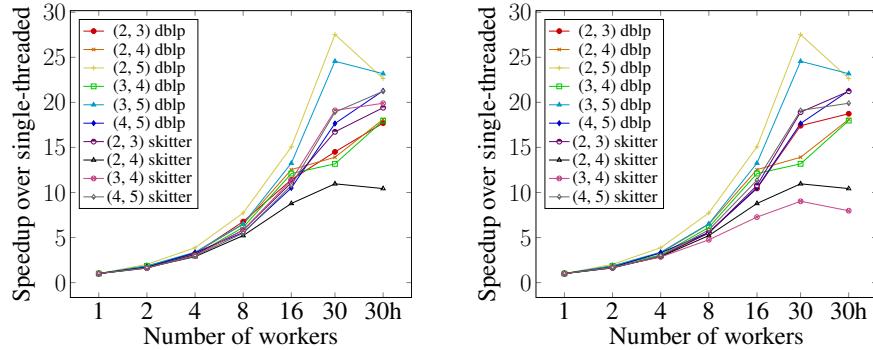


Figure 11.8: Speedup of ANH-TE on the left and ANH-EL on the right over their respective single-threaded running times, on dblp and skitter for various r and s . “30h” denotes 30-cores with two-way hyper-threading.

our parallel $(1, 2)$, $(2, 3)$ and $(3, 4)$ nucleus decomposition hierarchy implementations to other implementations. Note that here, we include in our implementations the time needed to compute the low out-degree orientation and to compute the initial s -clique-degrees of each r -clique. We do not include the time required to load the graph in both our and other implementations.

For $(1, 2)$ nucleus (k -core) decomposition, we compare to the parallel PHCD [76] and the sequential NH [281]. Our fastest implementation for k -core is ANH-TE, and we see that ANH-TE is up to 2.57x slower than PHCD overall, but 1.87x faster than PHCD on dblp. We note that PHCD is optimized for the k -core decomposition specifically, rather than general (r, s) nucleus decomposition, whereas ANH-TE generalizes for larger r and s . In particular, like ANH-TE, PHCD constructs the hierarchy tree from the bottom-up after computing the core numbers of each vertex, but unlike ANH-TE, PHCD leverages the information from computing the core numbers to optimize for the k -

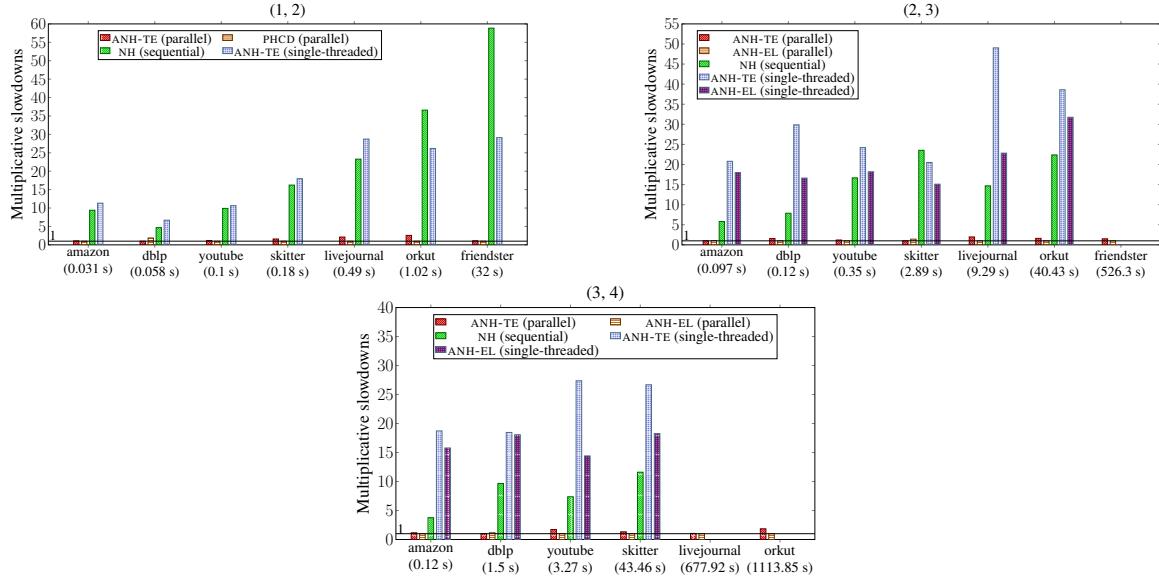


Figure 11.9: Multiplicative slowdowns, comparing ANH-TE, ANH-EL, the parallel PHCD [76], and the sequential ND [281], for $(1, 2)$, $(2, 3)$, and $(3, 4)$ nucleus decomposition. We give the multiplicative slowdown over the fastest implementation for each graph and each (r, s) , where the fastest running time is labeled in parentheses below each graph. We include end-to-end running times in this comparison, excluding only the time required to load the graph. We have omitted bars where the implementation runs out of memory or times out after 4 hours. We have also included a line marking a multiplicative slowdown of 1.

core hierarchy, by reordering vertices based on their core numbers. This optimization allows them to more efficiently divide the work of constructing the hierarchy across different threads, and allows them to reduce the work in practice when iterating over the neighbors of a vertex v with larger core numbers than that of v . Compared to the sequential NH, ANH-TE is 4.67–58.84x faster, particularly on larger graphs.

For $(2, 3)$ and $(3, 4)$ nucleus decomposition, we compare our parallel ANH-TE and ANH-EL to the sequential NH [281]. Considering the fastest of ANH-TE and ANH-EL for each graph, our implementations are 3.76–23.54x faster, demonstrating that we achieve good speedups from our use of parallelization. Sequentially, our fastest algorithm is between 2.02x faster and 4.2x slower than NH.

11.7.4 Performance of Approximate Hierarchy

We considered $\delta = 0.1, 0.5$, and 1 for our experiments for approximate (r, s) nucleus decomposition (where δ is the approximation parameter in Algorithm 11.2). We first compare APPROX-ARB-NUCLEUS to ARB-NUCLEUS [297] and see a speedup of up to 16.16x for $\delta = 0.1$, up to 8.35x for $\delta = 0.5$, and up to 10.88x for $\delta = 1$.

Besides the computation of the (r, s) -clique-core numbers, APPROX-ANH-TE, APPROX-ANH-EL, and APPROX-ANH-BL are identical to ANH-TE, ANH-EL, and ANH-BL, respectively. In other words, their hierarchy construction procedure is the same. We

observe up to a 3.3x speedup considering the fastest of our approximate algorithms for each graph and $r < s \leq 7$, over the fastest of our exact algorithms. Notably, for $\delta = 0.1$, we are able to compute the $(2, 5)$ nucleus decomposition hierarchy on friendster in 8783.2 seconds, where our exact implementations reach our timeout of 4 hours. The improvements in running time using our approximate algorithms are lower than when comparing APPROX-ARB-NUCLEUS to ARB-NUCLEUS [297] because even in our approximate algorithms, s -cliques must be exactly counted per r -clique, and much of the computation time is spent doing this.

In terms of accuracy, the average error in the (r, s) -clique-core number per r -clique is relatively low for all of our δ values. For $\delta = 0.1$, across all of our graphs and for $r < s \leq 7$, our coreness estimates per r -clique are have a multiplicative error of 1–2.92x on average (with a median of 1.33x) compared to the exact coreness numbers. For $\delta = 0.5$, the coreness estimates range from having a multiplicative error of 1–2.92x on average as well, with a median of 1.34x, and for $\delta = 1$, the coreness estimates range from having a multiplicative error of 1–3.05x on average, with a median of 1.35x. The multiplicative errors of the maximum (r, s) -clique-core number, across all graphs and for $r < s \leq 7$, are also reasonably low, with a median of 1.6x for $\delta = 0.1$, a median of 2x for $\delta = 0.5$, and a median of 2x for $\delta = 1$. The maximum multiplicative error for a given r -clique is 6.73x for $\delta = 0.1$, 6.98x for $\delta = 0.5$, and 7.32x for $\delta = 1$, but these arise for large s , notably when $r = 5$ and $s = 7$ for all δ , and these errors are still much lower than the theoretical guarantee of $((\binom{s}{r}) + \delta) \cdot (1 + \delta)$ given by Theorem 11.2.

11.8 Discussion

We have presented new parallel exact and approximate algorithms for nucleus hierarchy construction with strong theoretical guarantees. We have developed optimized implementations of our algorithms, which interleave the coreness number computation with the hierarchy construction. Our experiments showed that our implementations outperform state-of-the-art implementations while achieving good parallel scalability.

Part III

Graph Clustering

Introduction

The main focus of this part of the thesis is shared-memory parallel graph clustering algorithms that are scalable and fast for graphs with up to tens to hundreds of billions of edges, while maintaining high quality compared to ground-truth data. Graph clustering, or community detection, has a wide range of applications spanning social network analysis, recommendation and search systems, metabolic pathway analysis, and machine learning pipelines, and has been well-studied under many frameworks. While there exists a wealth of graph and metric clustering algorithms with different theoretical guarantees and use-cases [288, 227, 5], one fundamental question is how performant and how effective these algorithms are on different types of real-world datasets, especially with ground-truth clusters. We introduce in this part new highly efficient clustering algorithms optimizing for different clustering objectives, and demonstrate their effectiveness in terms of speed and quality on real-world data. Different clustering algorithms offer different challenges in terms of effective parallelization and scalability, such as unavoidable sequential dependencies and objective functions that are NP-hard to optimize [108, 295]. In this part, we explore approximation guarantees and practical heuristics to approach these difficulties.

In Chapter 12, we present a highly scalable parallel correlation clustering implementation that can cluster graphs with hundreds of millions of edges in under 10 minutes on a 30-core machine with 240 GiB of main memory. Optimizing for the correlation clustering objective is of particular importance, since it yields higher quality clusters on ground truth data than other prominent scalable algorithms for community detection, which we demonstrate in this chapter. Our implementation heuristically relaxes convergence and consistency guarantees to achieve these speeds, but matches the precision and recall of its sequential counterpart on ground truth data; we additionally show P-completeness results for the Louvain method for correlation clustering, suggesting that heuristic approaches are necessary to achieve high scalability. Moreover, our implementation is tunable with generalized parameters that allow for widespread tradeoff curves between precision and recall quality on ground truth data. To the best of our knowledge, we are the first to scale to even million-edge graphs with comparable quality to sequential algorithms.

In Chapter 13, we study hierarchical agglomerative clustering (HAC), a bottom-up approach to hierarchical clustering, which has higher quality compared to other hierarchical clustering methods. We present a parallel algorithm for approximate HAC that scales to graphs with hundreds of billions of edges on commodity multicore machines, with better objective value on average than the sequential baseline

and high quality on ground truth data. Additionally, we show P-completeness results for HAC, suggesting that parallelizing these algorithms efficiently is inherently difficult. Our algorithm is the first to parallelize approximate average-linkage HAC with polylogarithmic span, and prior to our work, no sublinear span algorithm was known.

The results in this part of the thesis have appeared in the following publications.

- Jessica Shi, Laxman Dhulipala, David Eisenstat, Jakub Łącki, and Vahab Mirrokni. “Scalable Community Detection via Parallel Correlation Clustering”. In: *Proceedings of the VLDB Endowment*, 14(11), pp. 2305–2313, 2021. (Chapter 12)
- Laxman Dhulipala, David Eisenstat, Jakub Łącki, Vahab Mirrokni, and Jessica Shi. “Hierarchical Agglomerative Graph Clustering in Poly-Logarithmic Depth”. In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 22925–22940, 2022. (Chapter 13)

Chapter 12

Correlation Clustering

12.1 Introduction

A major challenge in designing scalable graph clustering algorithms that can handle graphs with billions of edges is to design algorithms that can achieve fast speed at high scale while retaining high quality as evaluated on data sets with ground truth. Many graph clustering algorithms have been proposed to address this challenge, and our goal is to develop a state-of-the-art algorithm from both speed and quality perspectives. In particular, we adopt a new LAMBDAACC framework, introduced by Veldt *et al.* [323], which provides a general objective encompassing modularity [149] and correlation clustering [27]. Veldt *et al.* show that LAMBDAACC framework unifies several quality measures, including modularity, sparsest cut, cluster deletion, and a general version of correlation clustering. Modularity is a widely-used objective that is formally defined as the fraction of edges within clusters minus the expected fraction of edges within clusters, assuming random distribution of edges. The goal of correlation clustering is to maximize agreements or minimize disagreements, where agreements and disagreements are defined based on edge weights indicating similarity and dissimilarity.

It is NP-hard to approximate modularity within a constant factor [108], so optimizing for modularity, and by extension optimizing for the LAMBDAACC objective, is inherently difficult. The most successful and widely-used modularity clustering implementations focus on heuristic algorithms, notably the popular Louvain method [51]. The Louvain method has been well-studied for use in modularity clustering, with highly optimized heuristics and parallelizations that allow them to scale to large real-world networks [219, 314, 308, 315].

In this chapter, we design, implement, and evaluate a generalized sequential and shared-memory parallel framework for Louvain-based algorithms including modularity and correlation clustering. We optimize the LAMBDAACC objective with state-of-the-art empirical performance, scaling to graphs with billions of edges. We also show that there is an inherent bottleneck to efficiently parallelizing the Louvain method, in that the problem of obtaining a clustering matching that given by the Louvain method on the LAMBDAACC objective, is P-complete. As such, we explore heuristic

optimizations and relaxations of the Louvain method, and demonstrate their quality and performance trade-offs for the LAMBDAACC objective.

As part of our comprehensive empirical study, we show that our sequential implementation is orders of magnitude faster than the proof-of-concept implementation of Veldt *et al.* [323]. We note that for both LABMDACC and correlation clustering objective, we are unaware of any existing implementation that would scale to even million-edge graphs and achieve comparable quality. We further show that our parallel implementations obtain up to 28.44x speedups over our sequential baselines on a 30-core machine.

Moreover, we show that optimizing for the correlation clustering objective is of particular importance, by studying cluster quality with respect to ground truth data. We observe that optimizing for correlation clustering yields higher quality clusters than the ones obtained by optimizing for the celebrated modularity objective. In addition, we compare our implementation to two other prominent scalable algorithms for community detection: Tectonic [321] and SCD [259] and in both cases obtain favorable results, improving both the performance and quality. Finally, even in the highly competitive and extensively studied area of optimizing for modularity, we obtain an up to 3.5x speedup over a highly optimized parallel shared-memory modularity clustering implementation in NetworKit [308].

Our code is publicly available at: <https://github.com/jeshi96/parallel-correlation-clustering>.

Further related work. Optimization for correlation clustering has been studied empirically in the case of complete graphs, which is equivalent to LAMBDAACC objective with resolution $\gamma = 0.5$ [120, 252]. In this restricted setting, several scalable parallel implementations have been obtained based on the KWIKCLUSTER algorithm [73, 252, 141]. We observe that KWIKCLUSTER typically obtains a negative LAMBDAACC objective, which significantly limits its practical applicability.

Scalable modularity clustering has been extensively studied both in the shared-memory [308, 130, 158, 219, 132, 344] and distributed memory [285, 261, 268, 145] settings. The two fastest implementations that we identify are NetworKit [308] and Grappolo [158, 145]. Both of them offer comparable performance, but we observed the NetworKit typically computes solutions with slightly larger objective, and thus we compare to NetworKit in our empirical evaluation. We also note that compared to these papers, our algorithm optimizes for a more general LAMBDAACC objective.

12.2 Preliminaries

We consider undirected weighted graphs $G = (V, E, w)$ in this chapter, where $w : E \rightarrow \mathbb{R}$ denotes the weight of each edge, and undirected unweighted graphs $G = (V, E)$, where $w_{uv} = 1$ for all $(u, v) \in E(G)$.

We use a generalized correlation clustering objective that is equivalent to the LAMBDAACC objective given by Veldt *et al.* [323]. Note that under a specific set of parameters, our objective similarly reduces to the classic modularity objective. Moreover, our definition can be more generally applied to weighted graph inputs.

Algorithm 12.1 – Sequential Louvain method for correlation clustering

```

1: procedure SEQUENTIAL-CC( $G, k$ )
2:    $C \leftarrow$  singleton clusters for each  $v \in V$ 
3:   do
4:      $\sigma \leftarrow$  random permutation of  $V$ 
5:     for each  $v = \sigma(i)$  do
6:       Move  $v$  to the cluster in  $C$  that maximizes the CC objective
7:     while the objective  $CC(C)$  has increased
8:     if no moves were made then
9:       return  $C$ 
10:     $G', k' \leftarrow$  SEQUENTIAL-COMPRESS( $G, C$ )
11:     $C' \leftarrow$  SEQUENTIAL-CC( $G', k'$ )
12:    return SEQUENTIAL-FLATTEN( $C, C'$ )

```

We fix a clustering resolution parameter $\lambda \in (0, 1)$. We define non-negative *vertex weights* $k : V \rightarrow \mathbb{R}_0^+$, where unless otherwise specified, we take $k_v = 1$ for all $v \in V$ (a redefinition of k is required for the modularity objective). We also define the *rescaled weight* w' of each pair of vertices $(u, v) \in V \times V$ to be $w'_{uv} = 0$ if $u = v$, $w'_{uv} = w_{uv} - \lambda k_u k_v$ if $(u, v) \in E$, and $w'_{uv} = -\lambda k_u k_v$ otherwise.

The goal is to maximize the CC objective, $CC(x) = \sum_{(i,j) \in V \times V} w'_{ij} \cdot (1 - x_{ij})$, where $x = \{x_{ij}\}$ represents the distance between vertices i and j in a given clustering. Specifically, $x_{ij} = 0$ if i and j are in the same cluster, and $x_{ij} = 1$ if i and j are in different clusters.

The modularity objective can be obtained from the CC objective by defining vertex weights k and setting λ appropriately. Note that Reichardt and Bornholdt [266] defined a modularity objective with a fixed scaling parameter $\gamma \in (0, 1)$ to be $Q(x) = \frac{1}{2m} \sum_{i \neq j} (A_{ij} - \gamma^{\frac{\deg(i)\deg(j)}{2m}})(1 - x_{ij})$, where $A_{ij} = 1$ if i and j are adjacent, and $A_{ij} = 0$ otherwise. Setting $\gamma = 1$, this objective is equivalent to the simpler modularity objective given by Girvan and Newman [149]. To modify CC to match the modularity objective, we set the node weights $k(v) = \deg(v)$ for each $v \in V$, and we set the resolution $\lambda = \gamma/(2m)$. Maximizing the two objective functions is then equivalent.

12.3 Algorithm and Optimizations

12.3.1 Sequential Louvain Method

We begin by describing the classic sequential Louvain method from Blondel *et al.* [51], SEQUENTIAL-CC, adapted for the correlation clustering objective. Algorithm 12.1 shows the pseudocode for SEQUENTIAL-CC.

The main idea is to repeatedly move vertices to clusters that would maximize the objective, and once no vertices can be moved, compress clusters into vertices and repeat this process on the compressed graph. In more detail, the algorithm takes as input a graph G and node weights k , and begins with singleton clusters. Then, it

iterates over each vertex in a random order, and locally moves vertices to clusters that maximize the CC objective. The computation to determine the cluster that vertex v should move to can be performed by maintaining the total vertex weight of each cluster. Note that the computation necessary to determine the cluster that a vertex v should move to given the objective function is omitted from the pseudocode for simplicity, but it can be efficiently performed by maintaining in every iteration the total vertex weight of each cluster in C . More precisely, if we denote the total vertex weight of a cluster c by K_c , the change in objective of a vertex v moving from its current cluster c to a new cluster c' (where $c \neq c'$) is given by

$$\left(\sum_{u \in c', (u,v) \in E} w_{uv} - \lambda k_v K_{c'} \right) - \left(\sum_{u \in c, (u,v) \in E} w_{uv} - \lambda k_v K_c + \lambda k_v^2 \right).$$

In other words, the change in objective depends solely on K_c , $K_{c'}$, and the weights of the edges from v to its neighbors in c and c' .

After all vertices have been moved, the algorithm repeats this step of locally moving vertices until no vertices have performed non-trivial moves. If no vertices changed clusters during this phase, then SEQUENTIAL-CC terminates. Otherwise, once a stable state has been achieved, the algorithm compresses the graph G (using a subroutine SEQUENTIAL-COMPRESS) by creating a new graph G' with vertex weights k' . Each cluster c in G corresponds to a vertex in G' with vertex weight $k'(c) = K_c$. Edges (u, v) in G are maintained as edges between the vertices corresponding to their clusters in G' , where multiple edges incident on the same vertices are combined into a single edge with weight equal to the sum of their weights.

Finally, the algorithm recurses on G' and k' . It takes the returned clustering C' on the compressed graph G' , and composes it with the original clustering C (using a subroutine SEQUENTIAL-FLATTEN). It assigns the cluster of a vertex v in G to be the cluster of its corresponding vertex in the compressed graph G' , composing the clustering obtained in the recursion onto the original graph.

The main bottleneck in parallelizing SEQUENTIAL-CC is the sequential dependencies in moving each vertex to the cluster that maximizes the objective, and we prove a related P-completeness result in Section 12.3.3, showing that the problem of obtaining a clustering equivalent to any clustering C given by moving each vertex to its best cluster, is inherently sequential to solve under standard complexity-theory assumptions.

As such, to obtain an empirically efficient implementation that achieves good parallelism, we heuristically relax the sequential dependency and allow vertices to move to clusters *concurrently*. While vertices move to clusters that would individually maximize the objective, these moves in tandem may lower the total objective; there is no guarantee of convergence. We show an example in Figure 12.1. Note that this is a common parallelization technique in Louvain methods for modularity clustering, and it has been observed that in practice this technique converges for the modularity objective [308]. We now discuss optimizations that can be used with this relaxation.

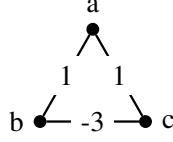


Figure 12.1: An example where parallel local vertex moves lowers the total objective. Assume $\lambda = 0$ and initial clusters are singletons with objective 0. If b and c are both scheduled to move at the same time, they each choose cluster $\{a\}$, leading to a single cluster $\{a, b, c\}$ with objective -1.

Algorithm 12.2 – Parallel Louvain method for correlation clustering

```

1: procedure BEST-MOVES( $G, k, num\_iter, C$ )
2:   Define  $id(c)$  to be the index of cluster  $c \in C$ 
3:    $D \leftarrow$  array of size  $n$ 
4:    $V' \leftarrow V$ 
5:   for  $i$  in range( $num\_iter$ ) do
6:     parfor  $v \in V'$  do
7:        $D[v] \leftarrow id(c)$  such that moving  $v$  to  $c \in C$  would maximize  $CC(C)$ 
8:       Move  $v$  to  $D[v]$ 
9:     if no moves were made in iteration  $i$  then break
10:     $V' \leftarrow \{ \text{neighbors of } v \mid v \text{ moved} \}$ 
11:   return  $C$ 

1: procedure PARALLEL-CC( $G, k, num\_iter$ )
2:    $C \leftarrow$  singleton clusters for each  $v \in V$ 
3:    $C \leftarrow$  BEST-MOVES( $G, k, num\_iter, C$ )
4:   if no moves were made in BEST-MOVES then
5:     return  $C$ 
6:    $G', k' \leftarrow$  PARALLEL-COMPRESS( $G, C$ )
7:    $C' \leftarrow$  PARALLEL-CC( $G', k', num\_iter$ )
8:    $C \leftarrow$  PARALLEL-FLATTEN( $C, C'$ )
9:    $C \leftarrow$  BEST-MOVES( $G, k, num\_iter, C$ )
10:  return  $C$ 

```

12.3.2 Parallel Louvain Method and Optimizations

Algorithm 12.2 contains the pseudocode for each of the main optimizations that we consider in our parallelization of the Louvain method for the CC objective. Each optimization is highlighted in blue, and we display in the pseudocode only the best settings that offer a reasonable trade-off between quality and performance, as we show in Section 12.4.1. We discuss in the following sections the other modular options in our framework for each of these optimizations.

Our parallelization uses a natural heuristic relaxation of the sequential dependency in moving vertices to their desired clusters. A more faithful parallelization would fix a random permutation of V , and move in parallel the first ℓ vertices in order

for the largest ℓ such that moving these ℓ vertices would not affect each other's objectives. However, compared to a heuristic relaxation, not only does this involve greater overhead due to the prefix computation of vertices that do not conflict, but it also respects sequential dependencies that may not affect later vertex moves. Thus, in the interest of performance, we consider the parallelization given in Algorithm 12.2.

Note that the heuristic relaxation to allow vertices to move concurrently to their desired clusters is encapsulated in the subroutine BEST-MOVES. We include an additional parameter, num_iter , which bounds the number of iterations in which we move each vertex to its desired cluster; this is necessary due to the lack of guarantee of convergence. Moreover, in order to allow each vertex to efficiently compute its best move, we maintain the total vertex weights K_c of each cluster c , which is not shown in the pseudocode for simplicity.

Our main optimizations introduce symmetry breaking and work reduction techniques to improve performance while maintaining the objective. We also discuss a refinement step that improves the objective at the cost of running time and a higher memory overhead.

12.3.2.1 Optimization: Synchronous vs Asynchronous

The first optimization involves scheduling individual vertex moves in BEST-MOVES, on Line 8. We explore two options: *synchronous* and *asynchronous*. These options have been previously studied in parallelizing the Louvain method for the modularity objective [308, 285].

In the synchronous setting, instead of moving vertices on Line 8 immediately after the computation of their desired cluster, we move all vertices in parallel to their desired cluster $D[v]$ after the parallel for loop on Line 6. This can be efficiently performed in parallel by aggregating vertices that move from the same clusters and vertices that move to the same clusters. In the asynchronous setting, we perform vertex moves on Line 8 as highlighted in blue. Note that moving a vertex v in this manner potentially interferes with the computation that each vertex performs on Line 7, where other vertices' computations of their desired cluster may depend on v 's current cluster, and the total vertex weight of v 's prior and v 's new cluster. Instead of using locks or other synchronization methods, we relax consistency guarantees, performing separate operations to update the cluster and the total vertex weight of the cluster that v moves to. Thus, there is no guarantee that the stored total vertex weights of clusters represent the actual totals.

We show in Section 12.4.1 that perhaps surprisingly, the asynchronous setting outperforms the synchronous setting, particularly in terms of objective. Of the optimizations, the asynchronous optimization contributes most significantly towards an improvement in objective. This is because the asynchronous setting allows for symmetry breaking, while in the synchronous setting, vertices that are attempting to move away from each other may inadvertently move to the same cluster, since they must move in lockstep. For certain graphs, this symmetry breaking also allows the asynchronous setting to outperform the synchronous setting in running time, due to fewer vertex moves required to obtain the maximal objective.

Previous uses of the Louvain method for other objectives explored different schedules for vertex moves with more granular trade-offs [219, 25]. We found that our asynchronous setting outperforms methods that maintain consistency, in quality and speed.

12.3.2.2 Optimization: All Vertices vs Neighbors of Clusters vs Neighbors of Vertices

We now consider optimizations that reduce the set of vertices to consider moving in every iteration of BEST-MOVES. When considering vertices on Lines 6 – 7, we note that following a set of vertex moves in the previous iteration, we can reduce the number of vertices that would be likely to be induced to change clusters by the vertex moves in the previous iteration. This idea has been previously used in work on the Louvain method for the modularity objective [249, 25]. Notably, the BEST-MOVES subroutine takes a significant portion of total clustering time, and reducing the subset of vertices to consider offers performance improvements.

In more detail, isolating a vertex v which has in the previous iteration moved from cluster c to cluster c' , the vertices that would be affected by this move in the next iteration belong to three categories: (a) neighbors of v , (b) neighbors of any vertex in c , and (c) vertices in c' . Any vertex not in one of these categories is not induced to move clusters due to v 's move. This is due to the change in objective formula. For conciseness in describing category (b), we formally define *neighbors of clusters* C to be the union of the neighbors of each vertex in each cluster in C .

In our algorithm, we consider three options for this optimization: restricting considered vertices to *neighbors of vertices* moved in the previous iteration, restricting considered vertices to *neighbors of clusters* that vertices have moved to in the previous iteration, and considering *all vertices* in each iteration. The first option corresponds to the update on V' in Line 10, highlighted in blue. The second option would instead replace this line with setting $V' \leftarrow \{ \text{neighbors of the current cluster } C[i] \mid v \text{ moved from cluster } C[i] \text{ to cluster } c \}$, and the final option would set $V' \leftarrow V$.

We show in Section 12.4.1 that restricting V' to the set of neighbors of vertices that have moved in the previous iteration outperforms both other options while maintaining comparable objective. This is because the vertices that are most affected by moving vertices in terms of objective are neighbors of the moving vertices, and thus most of the objective obtained is from considering these neighbors. While there may be contrived scenarios in which more objective is affected due to non-neighbors of moving vertices with sufficiently large edge weights¹, we do not see these scenarios in practice, and considering a smaller subset of vertices in each iteration allows for less total work to be performed, since we save on the cost of computing best moves for other vertices. The performance improvements outweigh the marginal loss in objective from these cases.

¹For instance, given a large enough star graph where each leaf has a small enough positive edge weight to the center, following the first set of moves in which every leaf clusters with the center.

12.3.2.3 Optimization: Multi-level Refinement

Finally, we consider a popular multi-level refinement optimization [272, 308]. Note that the first phase of our parallel algorithm and the classic Louvain method involves what can be viewed as successive coarsening steps, in which we perform best vertex moves and compress the resulting clustering into a coarsened graph; vertices in the coarsened graph correspond to clusters, or sets of vertices, in the original graph. For instance, each vertex v in the original graph is clustered into a cluster C , which corresponds to the vertex v' in the coarsened graph. We then recurse on the coarsened graph. Following the recursion, we receive a clustering on the coarsened graph, which we must translate to a clustering on the original graph. Each vertex v' in the coarsened graph now belongs to a cluster C' in the coarsened graph, and we must now assign the cluster of the original vertex v . We use a flattening procedure for this, where we simply assign each vertex v in the original graph to the corresponding cluster C' .

However, note that v did not have an opportunity to move clusters individually in successive recursive steps, and because there is no guarantee of convergence, clusters may not reach a steady state before compressing the graph. v may have ended up in a sub-optimal cluster C when the coarsening was performed and would have been unable to move after the coarsening. Now, given its new cluster C' , v may desire to change clusters. The *multi-level refinement* optimization allows for v to move by performing a refinement step after each flattening step, as we traverse back up the recursive hierarchy. We simply perform a further iteration of BEST-MOVES on each individual vertex v , before returning the clustering.

This refinement optimization is shown on Line 9, highlighted in blue. Omitting this line removes the optimization. The optimization increases the space usage and the amount of time required for our implementation, since it requires each compressed graph to be maintained throughout and since it adds an additional subroutine, but it non-trivially improves quality, as we show in Section 12.4.1. This is due to the lack of guarantee of convergence in Algorithm 12.2, where vertices may be coarsened non-optimally. The refinement step allows for these vertices to move to better clusters as we traverse back up the recursive hierarchy, resulting in better quality.

12.3.2.4 Other Optimizations

We make use of other practical optimizations in our parallel implementation of PARALLEL-CC. First, we use the theoretically efficient parallel primitives available in the Graph Based Benchmark Suite (GBBS) [106]. Overall, these primitives and the work-efficient scheduler provided in GBBS offer on average a 1.43x speedup over Intel’s Parallel STL library [46]. Importantly, we use the EDGEMAP primitive from GBBS to maintain the frontier of neighbors of moved vertices or of modified clusters in each step of BEST-MOVES. EDGEMAP takes a vertex subset and applies a user-defined function to generate a new vertex subset – in our case, generated from specified neighbors. The primitive switches between a sparse and a dense representation of the subset depending on size, and the implementation of EDGEMAP similarly changes depending on the size of the input subset and the number of outgoing edges.

We also efficiently parallelize the sequential graph compression and cluster flattening subroutines, SEQUENTIAL-COMPRESS and SEQUENTIAL-FLATTEN respectively. Flattening a given clustering C to the clustering C' from the coarsened graph can be parallelized by maintaining a set of cluster IDs for each vertex (given by the index in $[0, n]$ of the cluster in C), and assigning for each vertex, the cluster ID in C' corresponding to the cluster containing its cluster ID in C . Moreover, we parallelize graph compression by aggregating in parallel the edges in the original graph by the cluster IDs of their endpoints, and using parallel reduces to combine edges whose endpoints correspond to the same cluster ID.

Furthermore, in computing each vertex's desired cluster on Line 7, we make use of a parallel and a sequential subroutine, which we choose heuristically depending on the degree of the vertex. The change in objective for moving a vertex v to other clusters can be efficiently parallelized by iterating through v 's neighbors in parallel and using a parallel hash table [148], from the GBBS implementation, to maintain the sum of edge weights to neighbors in the same cluster. However, for vertices of small degree and with large V' , the parallel overhead of maintaining such a hash table for each vertex is too costly. On the other hand, for vertices of large degree and with small V' , the parallel overhead is negligible compared to the improved depth in utilizing a parallel hash table. We use a fixed threshold to choose between using the sequential subroutine versus using the parallel subroutine.

12.3.3 P-completeness of Louvain

We prove here that the problem of obtaining the clustering given by the Louvain method maximizing for the CC objective is P-complete.

Theorem 12.1. *The problem of obtaining a clustering equivalent to that given by the Louvain method maximizing for the CC objective is P-complete.*

Proof. We set $\lambda = 0$ for the purposes of this proof. We will show that there is an NC reduction from the monotone circuit-value problem (CVP), where given an input circuit C of \wedge and \vee gates on n variables (x_1, \dots, x_n) and their negations, and an assignment of truth to these variables, the problem is to compute the output value of C .

The reduction works as follows. We use a fixed small $\varepsilon > 0$, and we construct a graph G . Initially, the vertices of G are the literals (x_1, \dots, x_n) and their negations, and two additional vertices t and f (representing true and false respectively). We assign an edge with a large enough constant negative weight between t and f . We also assign an edge with a large enough constant positive weight between each literal and its corresponding t or f , depending on if the literal is true or false respectively.

Then, for every gate, say $g_i \vee g_j$ which outputs to g_k , we add a corresponding vertex g_k and g'_k in G . We use a weight w_{ijk} , which we define later. We also add edges of weight w_{ijk} between (g_i, g_k) and (g_j, g_k) , and an edge of weight $(2 + \frac{2}{3} \cdot \varepsilon)w_{ijk}$ between (g_k, g'_k) . Finally, we add an edge of weight $(1 + \varepsilon)w_{ijk}$ between (g_k, t) , and

an edge of weight $(1 + \frac{1}{2} \cdot \varepsilon)w_{ijk}$ between (g_k, f) . Figure 12.2 shows these vertices and edges.

We then define w_{ijk} as follows, for each gate $g_i \vee g_j$. We take the topological sort of the directed acyclic graph given by the circuit C , where the vertices correspond to the gates and the literals. We call the ordered vertices of this DAG given by the topological sort (c_1, \dots, c_n) , in order, and we let $c(g)$ denote the c_i that corresponds to the gate or literal g . Then, we define a function $f(c_i) = 1 / \prod_{1 \leq j < i} \deg(c_j)$, where $\deg(c_j)$ denotes the degree of c_j . We note that $f(\cdot)$ can be efficiently computed for each c_i by taking a prefix product of the degrees of c_i , in order. Finally, given a gate $g_i \vee g_j$ which outputs to g_k , we set $w_{ijk} = \min(f(c(g_i)), f(c(g_j)))$.

Note that the construction for every \wedge gate is performed similarly, except we swap the edge weights between (g_k, t) and (g_k, f) . More explicitly, we instead add an edge of weight $(1 + \frac{1}{2} \cdot \varepsilon)w_{ijk}$ between (g_k, t) , and an edge of weight $(1 + \varepsilon)w_{ijk}$ between (g_k, f) .

We now claim that applying the Louvain method optimizing for the CC objective on this graph G solves the circuit C . In other words, we claim that each gate g_i will ultimately cluster with either t or f , depending on if its value in the circuit is true or false respectively. Recall that the Louvain method involves two main subroutines that iterate until convergence – a subroutine in which vertices move to their best local clusters, and a subroutine that compresses the graph. We actually prove that the clustering given by performing best local vertex moves until convergence on G produces two final clusters of this nature, one containing t and one containing f . Then, in the compression subroutine, we obtain two vertices in the compressed graph corresponding to t and f , and because the edge weight between t and f is a large enough negative constant, the two vertices in the compressed graph will not cluster with each other, terminating the algorithm.

We begin by proving a weaker statement, namely that at any given point in time throughout the best local vertex moves process, each gate g_i is clustered into either a) a singleton cluster containing only g_i , b) a two-vertex cluster containing g_i and g'_i , or c) a cluster containing g_i and either t or f (but not both), depending on g_i 's corresponding truth value.

We prove this statement using induction. The base case follows because we begin with singleton clusters. We also note that the literals x_i and their negations will always choose to cluster with their corresponding t or f , because of the large enough positive constant weight between the edge from x_i to its corresponding truth value. Similarly, the vertices t and f will always choose to cluster with corresponding x_i , due to the large enough positive constant weight. As such, we disregard literals and the vertices t and f in our inductive step. For our inductive step, we assume the inductive hypothesis, and show that the statement holds when we consider the local best move of a vertex g_k , originating from a gate $g_i \vee g_j$ (the argument for a gate $g_i \wedge g_j$ follows symmetrically).

Note that g'_k only has a single positive weight edge to g_k , so its local best move is always to move to the cluster that g_k is in. If neither g_i nor g_j are clustered with t or f when g_k decides its best move, then g_k will always choose to move to the

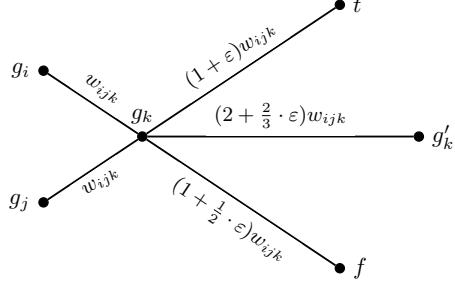


Figure 12.2: The vertices and edges added to the graph G , given a gate $g_i \vee g_j$ which outputs to g_k .

cluster that g'_k is in. This follows by construction because the edge weight (g_k, g'_k) of $(2 + \frac{2}{3} \cdot \varepsilon)w_{ijk}$ exceeds the edge weights from g_k to g_i , g_j , t , and f , and because the sum of the weights of the edges from g_k to its out-neighbors must be less than w_{ijk} . Moreover, since g'_k only ever decides to cluster with g_k , either g_k will remain in its current cluster, which satisfies the inductive step, or g_k will move to cluster with g'_k , which must be a singleton cluster. In this case, g_k forms a two-vertex cluster with g'_k .

If at least one of g_i and g_j is clustered with t when g_k decides its best move, WLOG g_i , then g_k will always choose to move to cluster with t , which by the inductive hypothesis must correspond with g_k 's truth value. This is because the sum of the weight of the edges (g_k, t) and (g_k, g_i) is given by $(2 + \varepsilon)w_{ijk}$, which exceeds the weight of the edge (g_k, g'_k) , or $(2 + \frac{2}{3} \cdot \varepsilon)w_{ijk}$, thus inducing g_k to move if it was originally in either a singleton cluster or a two-vertex cluster with g'_k . As before, the sum of the weights of the edges from g_k to its out-neighbors must be less than w_{ijk} , so g_k will not consider any other cluster.

Similarly, if both of g_i and g_j are clustered with f when g_k decides its best move, then g_k will always choose to move to cluster with f , which by the inductive hypothesis must correspond with g_k 's truth value. This again follows directly from the weights of the edges from g_k to g_i , g_j , t , f , and g'_k . Moreover, if exactly one of g_i and g_j is clustered with f , and neither are clustered with t , then g_k will always choose to move to the cluster that g'_k is in, by virtue of the constructed edge weights. In both of these cases, the inductive step is ultimately satisfied.

Thus, we have shown that at any given point in time throughout the best local vertex moves process, each gate g_i is clustered into either a) a singleton cluster containing only g_i , b) a two-vertex cluster containing g_i and g'_i , or c) a cluster containing g_i and either t or f (but not both), depending on g_i 's corresponding truth value.

To complete the proof, we must now show that the best moves process converges with each vertex clustered with either t or f , depending on its corresponding truth value. This follows from a similar argument to our inductive argument above, where for a gate $g_i \vee g_j$ that outputs to g_k , if g_i and g_k are both clustered with their corresponding truth value, then g_k will always choose to cluster with its corresponding truth value during its best move operation. Furthermore, the literals x_i and their negations necessarily choose to cluster with their corresponding truth values whenever

| Graphs | Num. Vertices | Num. Edges |
|-------------|---------------|---------------|
| amazon | 334,863 | 925,872 |
| dblp | 317,080 | 1,049,866 |
| livejournal | 3,997,962 | 34,681,189 |
| orkut | 3,072,441 | 117,185,083 |
| twitter | 41,652,231 | 1,202,513,046 |
| friendster | 65,608,366 | 1,806,067,135 |

Table 12.1: Sizes of graph inputs.

prompted, by construction of the edge weights between the literals and t and f . Thus, it follows that the best moves process converges with each gate g_i clustered with either t or f .

This completes the reduction, since we can obtain from the final clusters the solution to the circuit C . □

12.4 Experiments

In this section, we present a comprehensive evaluation of our algorithms, showing significant speedups over state-of-the-art implementations and high-quality clusters compared to ground truth.

We show that optimizations that address symmetry breaking and work reduction result in an overall faster implementation while maintaining objective, and additional refinement steps improve the objective at the cost of performance and memory usage. We also demonstrate significant speedups over state-of-the-art implementations due to our theoretically efficient parallelization of key subroutines, while obtaining high quality compared to ground truth.

Environment. We run most experiments on a c2-standard-60 Google Cloud instance, with 30 cores (with two-way hyper-threading), 3.8GHz Intel Xeon Scalable processors, and 240 GiB main memory. For experiments on large graphs we use a m1-megamem-96 Google Cloud instance, with 48 cores (with two-way hyper-threading), 2.7GHz Intel Xeon Scalable processors, and 1434 GiB main memory. We compile our programs with g++ (version 7.3.1) and the -O3 flag, and we use an efficient work-stealing scheduler, which, as shown in [46], provides on average a 1.43x speedup over Intel’s Parallel STL library. We also terminate any experiment that takes over 7 hours.

Graph Inputs. We test our implementations on real-world undirected graphs from the Stanford Network Analysis Project (SNAP) [207], namely com-dblp, com-amazon, com-livejournal, com-orkut, and com-friendster. We also use twitter, a symmetrized version of the Twitter graph representing follower-following relationships [200]. Table 12.1 shows the details of the SNAP graphs that we perform experiments upon. To show the quality of our implementations, we compare with the top 5000 ground-truth

communities given by SNAP. These communities may overlap, so to compute average precision and recall, for each ground-truth community c , we match c to the cluster c' with the largest intersection to c .² This matches the methodology used by Tsourakakis *et al.* in evaluating TECTONIC [321].

We also use an approximate k -NN algorithm [157] to construct weighted graphs from pointset data, from the UCI Machine Learning repository [114]. Specifically, we use the Optical Recognition of Handwritten Digits (digits) dataset (1,797 instances) and the Letter Recognition (letter) dataset (20,000 instances), both of which also have ground truth clusters which we compare to. We use the state-of-the-art ScaNN k -NN library [157] to perform k -NN, with $k = 50$ and using cosine similarity. We symmetrize the resulting k -NN graph. Additionally, we demonstrate scalability using synthetic graphs generated by the standard rMAT graph generator [64], with $a = 0.5$, $b = c = 0.1$, and $d = 0.3$.

All experiments are run on c2-standard-60 instances, except for experiments on the twitter and friendster graphs, which are run on m1-megamem-96 instances due to the higher memory requirement.

Implementations. We test the Louvain-based implementations of our sequential and parallel correlation clustering algorithms (SEQ-CC and PAR-CC respectively). We also redefine vertex weights and λ as discussed in Section 12.2 to obtain modularity clustering implementations (SEQ-MOD and PAR-MOD). For our parallel implementations, we use $num_iter = 10$ unless otherwise specified. For our sequential implementations, we use the superscript ^{CON} if we run to convergence (without restricting the number of iterations), and we use no superscript if we use $num_iter = 10$. We run each experiment 10 times and report the average time and objective.³

We compare to two correlation clustering implementations, namely the parallel C4 and CLUSTERWILD! by Pan *et al.* [252] (based on the sequential correlation clustering algorithm KWIKCLUSTER [10]), and the sequential Louvain-based implementation in the correlation clustering framework LAMBDAACC, by Veldt *et al.* [323].

We note that there is a rich body of work on pivot-based correlation clustering algorithms, both parallel and sequential, including prior work by Chierichetti *et al.* [73] and by García-Soriano *et al.* [141]. These works are based on KWIKCLUSTER (also known as PIVOT), and offer faster performance with matching or worse approximation guarantees. However, in our comparison to C4, which parallelizes KWIKCLUSTER and which matches KWIKCLUSTER’s approximation guarantee, we note that while C4 is much faster than PAR-CC, the quality is poor compared to PAR-CC in terms of both the CC objective and comparison to ground-truth communities, resulting in clusters of vertices with proportionally lower similarities to each other. Thus, we omit pivot-based work that offer the same or worse quality guarantees compared to C4.

We also compare to two state-of-the-art community detection algorithms which are based on triangle counts: the sequential TECTONIC by Tsourakakis *et al.* [321], and the shared-memory parallel SCD, by Prat-Pérez *et al.* [259]. Both algorithms were shown to deliver superior quality to multiple similarly scalable baseline methods. In

²Any given cluster c' may be matched to multiple or no ground-truth communities c .

³The average objective is non-deterministic when using the asynchronous setting from Section 12.3.2.1.

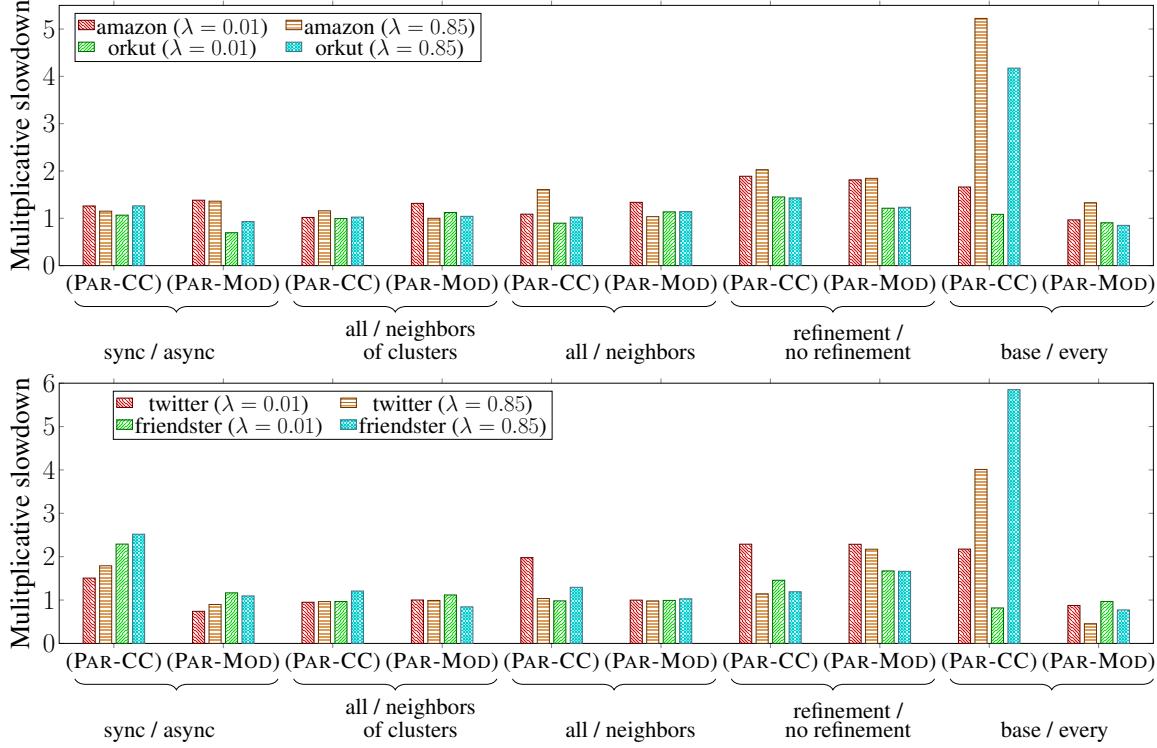


Figure 12.3: Multiplicative slowdown in average time of each optimization. Also shown is the slowdown for no optimizations (base) over every optimization. Running times were obtained for PAR-CC and PAR-MOD on the graphs amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$.

the special case of modularity, we compare to the parallel Louvian-based modularity clustering implementation in the NetworkKit toolkit (NETWORKKIT), by Staudt and Meyerhenke [308].

12.4.1 Tuning Optimizations

We evaluated the effectiveness of the different optimizations discussed in Section 12.3.2, namely, considering *synchronous* versus *asynchronous* vertex moves, considering *all vertices* versus *neighbors of clusters* that vertices have moved to versus *neighbors of vertices* that have moved as the vertex subset V' to iterate over, and considering *multi-level refinement* versus *no refinement*. We establish here that the optimizations that offer reasonable trade-offs between speed and quality are asynchronous vertex moves, considering neighbors of vertices that have moved as V' , and using multi-level refinement.

⁴We take the symmetric log of x to be $\text{sign}(x) \cdot \log |x|$. We use a symmetric log scale to more accurately depict the CC objectives, because the objectives are very large positive and negative numbers.

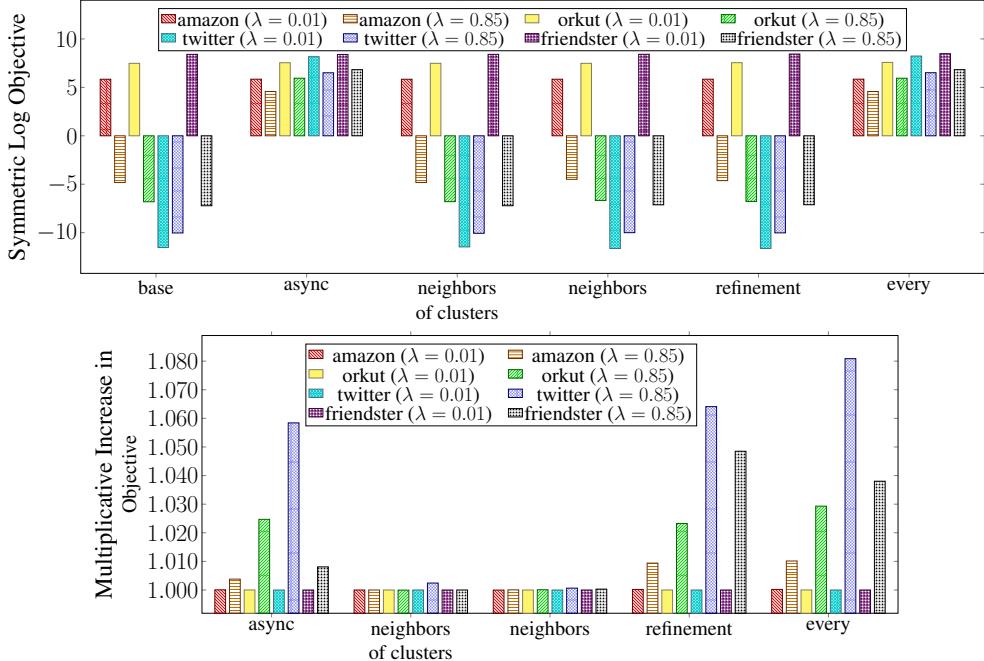


Figure 12.4: CC objective for PAR-CC on a symmetric log scale⁴(top) and multiplicative increase in the modularity for PAR-MOD over no optimizations (bottom), of each optimization and every optimization. Objectives were obtained on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$.

We tuned these optimizations on the graphs amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. Note that lower resolutions produce a clustering with fewer clusters, and higher resolutions produce a clustering with more clusters; these resolutions effectively model both scenarios, where differences in the number of clusters produced may affect performance. We fix the synchronous, all vertex moves, and no refinement options, and give running times and objectives considering turning on a single optimization at a time; these are the natural settings that do not optimize the basic sequential Louvain method. Considering both PAR-CC and PAR-MOD, Figure 12.3 shows the multiplicative slowdowns of synchronous over asynchronous, all vertices over neighbors of clusters, all vertices over neighbors of vertices, multi-level refinement over no refinement (note that multi-level refinement causes slowdowns, but improves quality over the basic no refinement option), and no optimizations over every optimization. Figure 12.4 shows the objectives for these optimizations.

We see speedups of up to 2.50x, with a median of 1.21x, using the asynchronous over the synchronous setting across PAR-CC and PAR-MOD. For PAR-CC, the synchronous setting often produces negative objective, whereas in the asynchronous setting, the objective is always positive, and we see a 1.29–156.01% increase in objective. The objective in the synchronous setting is negative likely due to the phenomenon shown in Figure 12.1, which is more likely to appear for large resolutions due to the objective computation. This phenomenon has additionally been discussed in prior work in relation to the modularity objective [262, 308]. For PAR-MOD, we see a

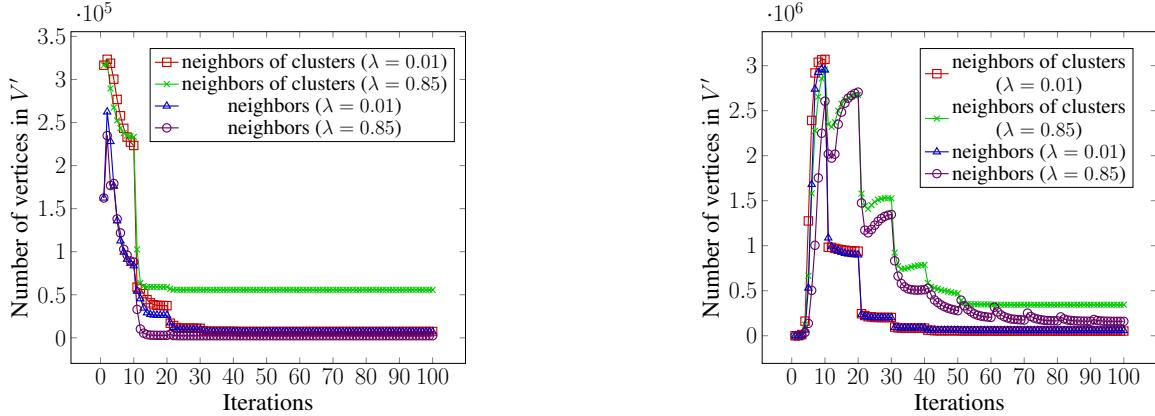


Figure 12.5: Number of vertices in V' per best move iteration of PAR-CC, for amazon (left) and orkut (right), considering neighbors of clusters and neighbors of vertices as the subset V' .

0.0015 – 5.84% increase in modularity using the asynchronous setting over the synchronous setting. The synchronous setting often leads to very poor objectives due to a lack of symmetry breaking, compared to the asynchronous setting where there is inherent randomness.

The asynchronous setting is also often faster than the synchronous setting, because due to the symmetry breaking, fewer vertices end up making moves that decrease the objective. For PAR-MOD on orkut and twitter, the asynchronous setting is not faster than the synchronous setting, but the increase in modularity using the asynchronous setting is more significant, compared to that obtained on other graphs. Up to 1.43x more time is spent computing best moves in the asynchronous setting compared to the synchronous setting, due to an increased number of vertex moves required to obtain the higher objective. Overall, considering tradeoffs between objective and speed, the best setting is the asynchronous setting.

We see up to a 1.98x speedup, with a median of 1.03x, considering neighbors of vertices compared to all vertices as the subset V' , and we see up to a 1.32x speedup, with a median of 1.01x, considering neighbors of clusters compared to all vertices. Moreover, the objectives obtained in all settings are comparable. This is because neighbors of previously moved vertices, and by extension neighbors of clusters of previously moved vertices, are most significantly affected in terms of objective by previously moved vertices, based on the change in objective formula. In the cases where the speedup is minimal, vertices in these classes represent a larger proportion of all vertices. Figure 12.5 shows the size of V' comparing neighbors of clusters and neighbors of vertices as the subset V' , for PAR-CC on amazon and orkut, using the synchronous setting and no refinement. In particular, for $\lambda = 0.85$ on amazon, the neighbors of vertices optimization obtains 1.61x speedup over all vertices, whereas the neighbors of clusters optimization obtains 1.16x speedup over all vertices. This is reflected in Figure 12.5, where there is a significant difference in the size of V' in this case. However, due to the cases with more significant speedups, the best setting in general is considering neighbors of vertices as the subset V' .

Finally, we see slowdowns of up to 2.29x, with a median of 1.67x, using multi-level refinement, compared to using no refinement. However, using refinement, we see 1.12 – 36.92% increase in the CC objective, and up to a 6.41% increase in modularity. Refinement improves objective because it allows vertices to move to better clusters following the compression steps, increasing the objective in situations where compression was not optimal. The increase in time is due to the added work in refinement, and in general, the best setting is to use refinement. Note that these results mirror prior work applying multi-level refinement for the modularity objective [272, 308]. For the modularity objective using small resolutions, the increase in objective is minimal; this is because in these cases, the objective obtained without using refinement is already very high (on average 0.99, where the maximum is 1.00).

In the remaining experiments, we fix the asynchronous setting, using neighbors of vertices, and using multi-level refinement, as the overall optimal settings, although we note that for small resolutions, multi-level refinement often offers little increase in objective considering the increase in running time. Overall, using all optimizations, we see up to a 5.85x speedup and up to a 156.01% increase in objective. For PAR-MOD, there are scenarios with up to a 2.20x slowdown in running time due to contention in the asynchronous setting compared to the synchronous setting, but in these cases, we see significant increases in modularity, of 2.93% – 8.09%.

12.4.2 Speedups and Scalability

Comparisons to Other Correlation Clustering Implementations. We first note that there exist no prior scalable correlation clustering baselines that offer high quality in terms of objective. The existing implementations are the parallel C4 and CLUSTERWILD! [252], which are based on a maximal independent set algorithm, and the sequential Louvain-based method in LAMBDACC [323]. Our implementations significantly outperform these baselines. Notably, C4 and CLUSTERWILD! offer significant speedups of up to 428.64x over PAR-CC, but achieve poor and often negative objective, with a decrease in the objective of 273.35 – 433.31% over PAR-CC. C4 and CLUSTERWILD! also achieve poor precision and recall compared to ground truth communities, with precision between 0.44 – 0.65 and recall between 0.10 – 0.15. In comparison, on the same graphs, PAR-CC achieves recall between 0.61 – 0.98 for precision greater than 0.50. Furthermore, LAMBDACC is a MATLAB implementation that uses an adjacency matrix to represent the graph, and cannot scale to graphs of more than hundreds of vertices.

In more detail, the parallel correlation clustering implementations C4 and CLUSTERWILD! [252] optimize for the same objective as our CC objective, if we set $\lambda = 0.5$; importantly, they do not generalize to other resolution parameters, and they do not take into account weighted graphs. We test the asynchronous versions of C4 and CLUSTERWILD!, which also outperform the synchronous versions while maintaining the objective, on the graphs amazon, dblp, livejournal, and orkut. Both implementations offer significant speedups over PAR-CC, of up to 139.43x and 428.64x respectively. However, rescaling the objectives given by C4 and CLUSTERWILD! to match the CC objective, we see that C4 and CLUSTERWILD! decrease the objective

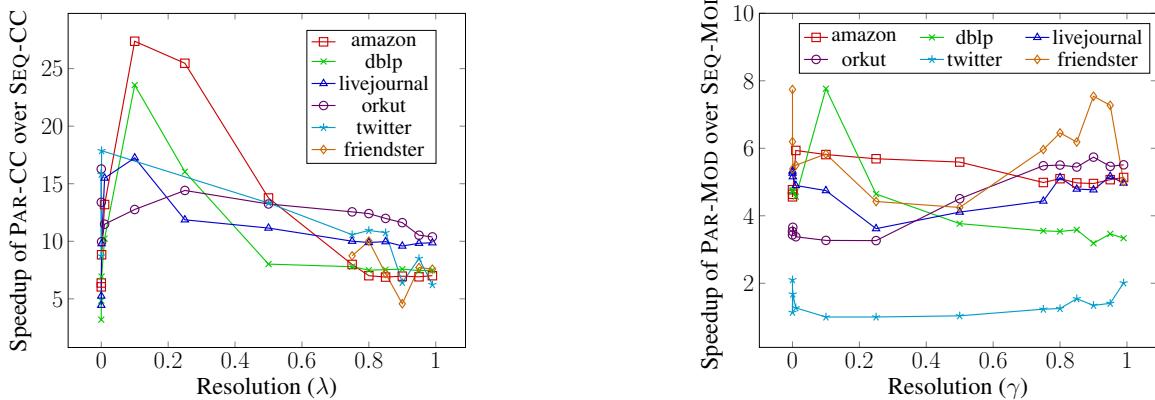


Figure 12.6: Speedup of PAR-CC over SEQ-CC (left) and of PAR-MOD over SEQ-MOD (right), on amazon, dblp, livejournal, orkut, twitter, and friendster, for varying resolutions. SEQ-CC timed out on twitter for $\lambda = 0.01, 0.1$, and 0.25 , and on friendster for $\lambda < 0.75$.

by 273.35 – 433.31% over PAR-CC. Notably, the objectives given by C4 and CLUSTERWILD! are often negative, meaning that they are unsuitable for optimizing for the CC objective. Moreover, compared to the top 5000 ground truth communities on these graphs, C4 and CLUSTERWILD! achieve poor precision and recall, with precision between 0.44–0.65 and corresponding recall between 0.10–0.15. In comparison, PAR-CC achieves recall between 0.61–0.98 for precision greater than 0.50.

Additionally, we compare against the Louvain-based sequential correlation clustering implementation in LAMBDAACC given by Veldt *et al.* [323]. Unfortunately, this implementation does not scale to large graphs of more than hundreds of vertices. We were able to test LAMBDAACC on the karate graph [343], which consists of 34 vertices and 78 edges. For $\lambda = 0.01$, LAMBDAACC takes 0.057 seconds to cluster the karate graph, whereas our PAR-CC takes 0.0002 seconds. The slowness of LAMBDAACC is because the code is in MATLAB, and it uses an adjacency matrix to represent the input graph; as such, it is unable to efficiently perform sparse graph operations.

Speedups. Given that there exist no prior scalable correlation clustering baselines that offer high quality in terms of objective, we demonstrate the speedups of our parallel implementations primarily against our own serial implementations, which include the applicable optimizations discussed in Section 12.4.1, namely the neighbors of vertices and multi-level refinement optimizations.

Figure 12.6 shows the speedup of PAR-CC and PAR-MOD over SEQ-CC and SEQ-MOD respectively. We also compared to SEQ-CC^{CON} and SEQ-MOD^{CON}; we note that running to convergence generally increases the running time while improving the objective, although the improvements are not always significant. However, as we show later in Section 12.4.3, the average precision-recall of SEQ-CC is significantly worse than that of SEQ-CC^{CON}, while our PAR-CC matches the average precision-recall of SEQ-CC^{CON}.

On the graphs amazon, dblp, livejournal, and orkut, and over varying resolutions, we see 3.19–27.38x speedups of PAR-CC over SEQ-CC, and 12.55–110.25x speedups

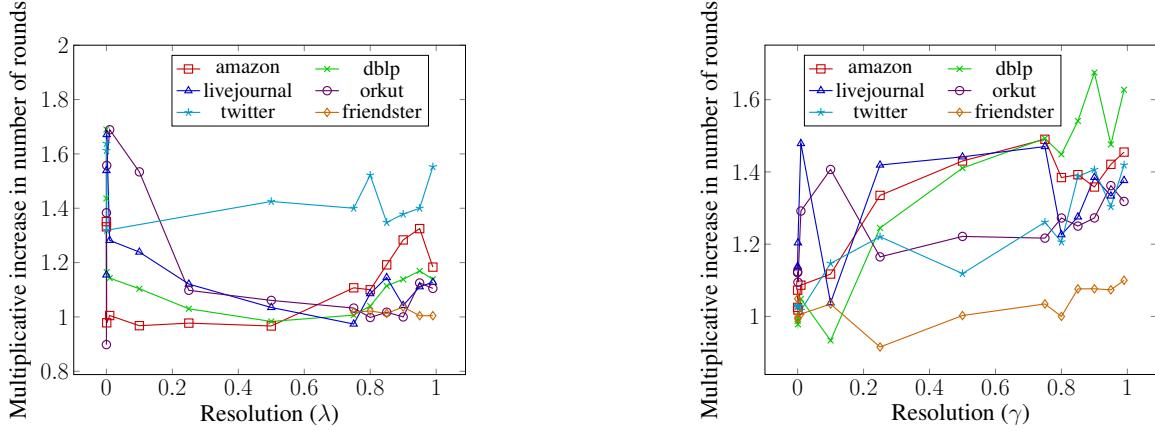


Figure 12.7: Multiplicative increase in the number of rounds until convergence or until the default number of iterations is reached, of PAR-CC over SEQ-CC (left) and of PAR-MOD over SEQ-MOD (right), on amazon, dblp, livejournal, orkut, twitter, and friendster, for varying resolutions. SEQ-CC timed out on twitter for $\lambda = 0.01$, 0.1 , and 0.25 , and on friendster for $\lambda < 0.75$.

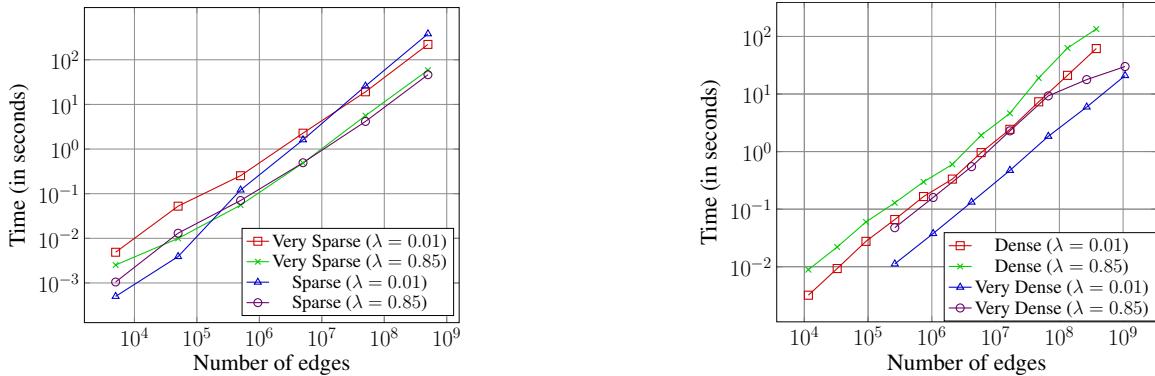


Figure 12.8: Scalability of PAR-CC over rMAT graphs of varying sizes.

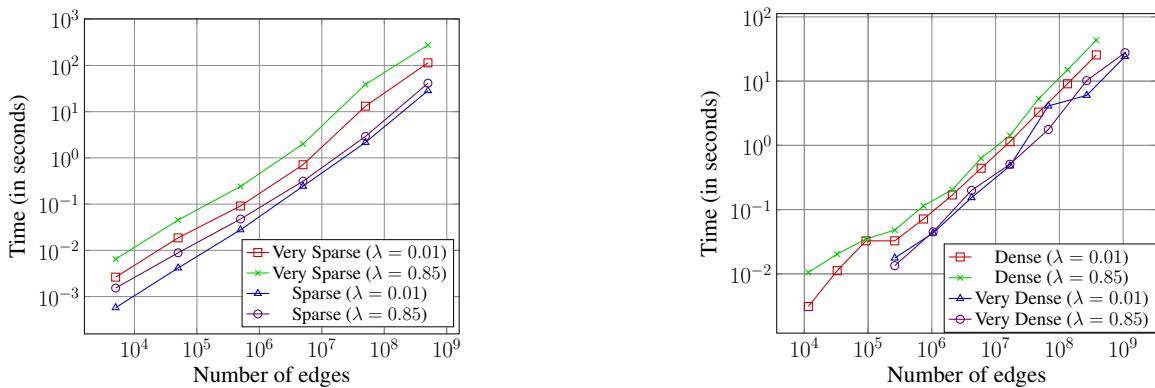


Figure 12.9: Scalability of PAR-MOD over rMAT graphs with varying numbers of vertices.

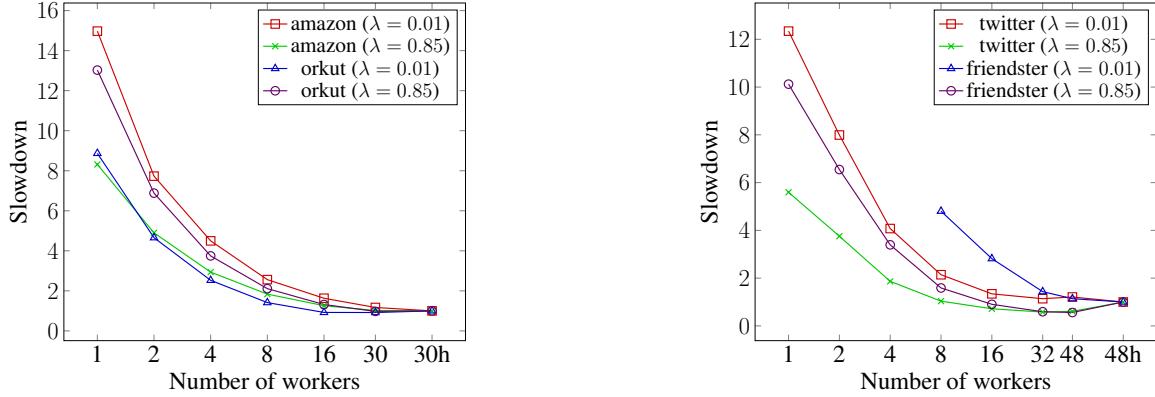


Figure 12.10: Scalability of PAR-CC over different numbers of threads, on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. 30h and 48h indicate 30 and 48 cores respectively, with two-way hyper-threading.

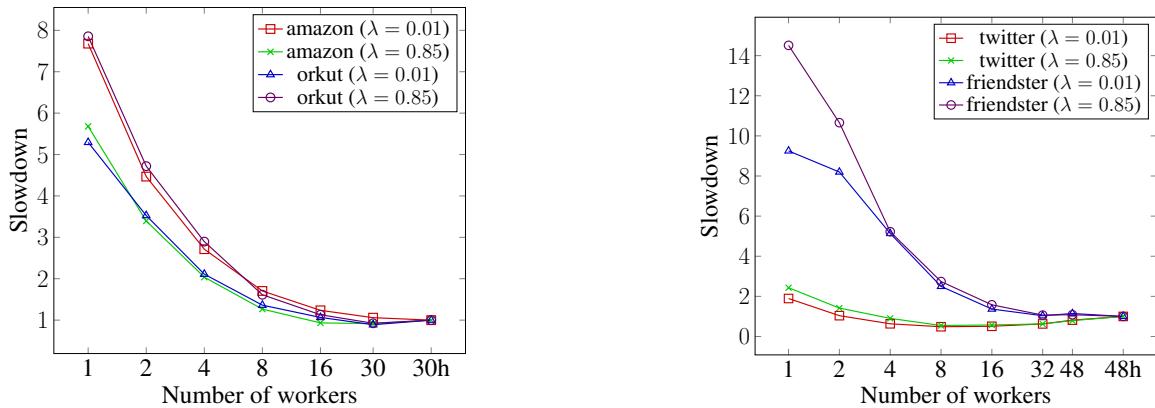


Figure 12.11: Scalability of PAR-MOD over different numbers of threads, on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. Note that 30h and 48h indicate 30 and 48 cores respectively, with two-way hyper-threading.

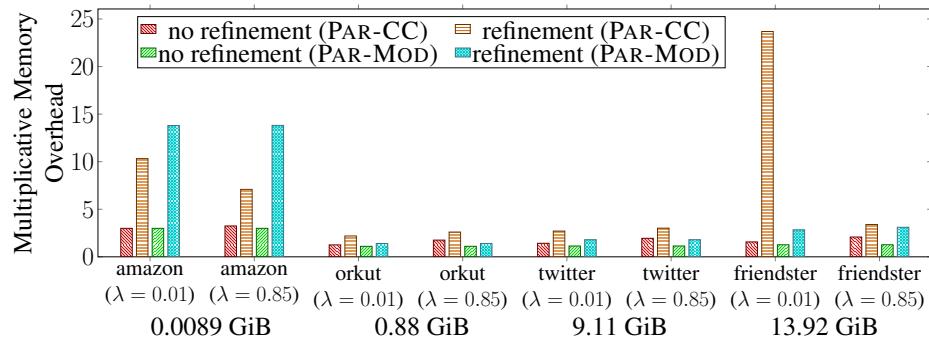


Figure 12.12: Multiplicative memory overhead, over the size of the input graph, of PAR-CC and PAR-MOD, on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. The labels below denote the size of the input graph.

of PAR-CC over SEQ-CC^{CON}. Our parallel implementations also achieve between 0.98–1.08x the CC objective of our serial implementations, demonstrating high performance while maintaining the CC objective. For PAR-MOD, we see between 3.18–7.76x speedups over SEQ-MOD, and 2.64–7.89x speedups over SEQ-MOD^{CON}, while achieving 1.00–1.06x the modularity of the serial implementations.

On the large graphs twitter and friendster, and over varying resolutions, we see 4.57–17.87x speedups of our PAR-CC over SEQ-CC, and up to 7.74x speedups of PAR-MOD over SEQ-MOD. We achieve between 0.95–1.00x the CC objective of SEQ-CC, and between 0.96–1.02x the modularity of SEQ-MOD. SEQ-CC timed out on twitter for $\lambda = 0.01, 0.1,$ and 0.25 , and on friendster for $\lambda < 0.75$. We note that we see lower speedups using PAR-MOD on twitter. We suspect that this is because twitter has a few vertices of particularly high degree (the maximum degree is 2,997,487, compared to the maximum degree in friendster of 5,214), and, across all resolutions, PAR-MOD and SEQ-MOD produce significantly fewer clusters on twitter relative to the size of the graph, compared to other graphs. In particular, for modularity clustering, the average cluster size on twitter is $2100–2.08 \times 10^7$ across all resolutions, whereas the average cluster size on friendster, a graph of similar size, is 1.11. As such, we see significantly increased contention on twitter.

Figure 12.7 shows the multiplicative increase in the total number of iterations required for PAR-CC and PAR-MOD over SEQ-CC and SEQ-MOD, which approximately displays the inverse of the behavior seen in the speedups shown in Figure 12.6 across different resolutions. We see greater speedups for resolutions where the number of iterations required in our parallel implementations match or are lower than the number of iterations required in our serial implementations. Whenever a greater number of iterations is required in parallel compared to serial, we observed lower speedups, simply due to the increased amount of work carried out by the parallel version.

Scalability. Figure 12.8 demonstrates the scalability of PAR-CC over rMAT graphs of varying sizes, with very sparse graphs ($m = 5n$), sparse graphs ($m = 50n$), dense graphs ($m = n^{1.5}$), and very dense graphs ($m = n^2$). Figure 12.9 shows the scalability of PAR-MOD over rMAT graphs of varying sizes, with very sparse graphs where $m = 5n$, sparse graphs where $m = 50n$, dense graphs where $m = n^{1.5}$, and very dense graphs where $m = n^2$. We observe that for both of our algorithms and across different resolutions ($\lambda = 0.01, 0.85$), the running times of our algorithms scale nearly linearly with the number of edges. Figure 12.10 shows the speedups of PAR-CC on amazon, orkut, twitter, and friendster, over different numbers of threads. Figure 12.11 shows the speedups of PAR-MOD on the same graphs, over different numbers of threads. Overall, we see good parallel scalability, with 5.59–14.97x self-relative speedups on PAR-CC, and 1.89–14.51x self-relative speedups on PAR-MOD. Note that using fewer threads for PAR-CC times out on friendster for $\lambda = 0.01$, and we again see lower speedups for PAR-MOD on twitter, where there is increased contention in using atomic compare-and-swaps due to the very few clusters produced relative to the size of the graph. Excluding twitter, we see 5.29–14.51x self-relative speedups on PAR-MOD.

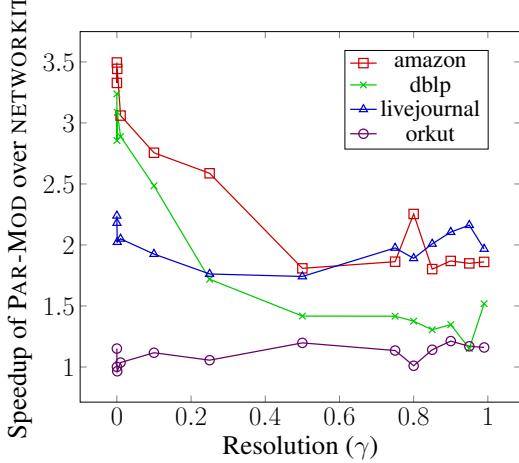


Figure 12.13: Speedup of PAR-MOD over NETWORKKIT on amazon, dblp, livejournal, and orkut, for varying resolutions.

Memory Usage. Figure 12.12 shows the memory usage of PAR-CC and PAR-MOD on amazon, orkut, twitter, and friendster.⁵ Theoretically, our memory usage is linear in the size of the input graph. We incur more memory when using multi-level refinement, particularly if more coarsening rounds are required, since we must store the coarsened graph from each recursive step. For instance, for PAR-CC on friendster with $\lambda = 0.01$, four coarsening rounds are used, compared to $\lambda = 0.85$, where only one coarsening round is used, hence the difference in memory overhead. Overall, using refinement, we incur a 1.40–23.68x memory overhead over the size of the input graph, whereas without refinement, we incur a 1.25–3.24x overhead.

Comparisons to Other Implementations. In the special case of modularity, we compare against the highly optimized parallel modularity clustering implementation NETWORKKIT [308]. Note that NETWORKKIT, like PAR-MOD, implements an asynchronous version of Louvain-based modularity clustering, and requires a parameter *num_iter* to guarantee completion. By default NETWORKKIT sets *num_iter* = 32, so to compare with PAR-MOD, we similarly set *num_iter* = 32. We show in Figure 12.13 the speedups of PAR-MOD over NETWORKKIT on amazon, dblp, livejournal, and orkut, for varying resolutions. We observe up to 3.50x speedups, primarily due to our optimization of the graph compression step (which we discuss below), and on average 1.89x speedups. For the twitter graph, setting $\gamma = 0.01, 0.85$, PAR-MOD gives between 1.08 – 3.03x speedups over NETWORKKIT while maintaining comparable modularity. For the friendster graph, we turn off NETWORKKIT’s turbo parameter (which offers a trade-off between memory usage and performance) due to space constraints, and setting $\gamma = 0.01, 0.85$, PAR-MOD gives between 1.23–1.26x speedups over NETWORKKIT while maintaining comparable modularity. Overall, we obtain between 0.99 – 1.00x the modularity given by NETWORKKIT’s implementation, where some variance

⁵The size of the input graph provided in Figure 12.12 is the total size in CSR format [274], which uses approximately 8 bytes per undirected edge.

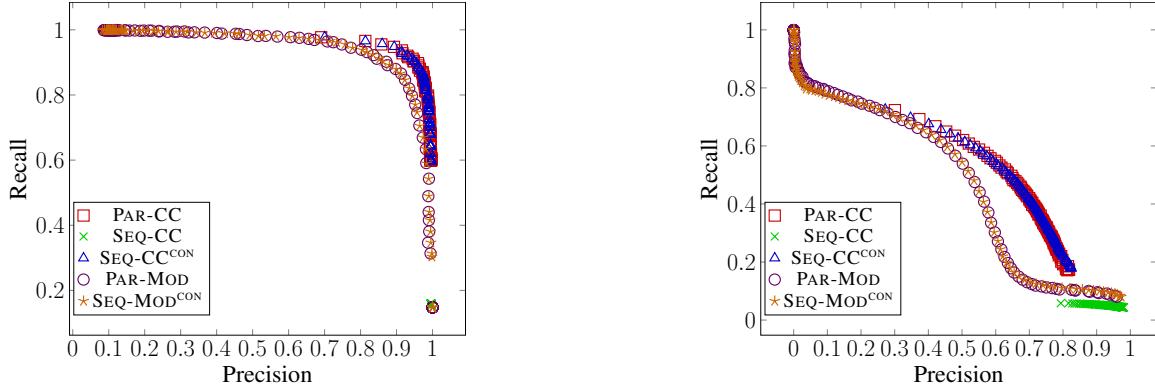


Figure 12.14: Average precision and recall compared to ground truth communities on amazon (left) and orkut (right) of the clusters obtained from PAR-CC and PAR-MOD, compared to SEQ-CC and SEQ-MOD, using varying resolutions.

appears due to the asynchronous nature of both implementations.

Our speedups over NETWORKKIT are primarily because NETWORKKIT does not efficiently parallelize the graph compression step between rounds of best vertex moves. Our implementations use a work-efficient algorithm to parallelize this step, where intra-cluster edges are aggregated in polylogarithmic depth with an efficient parallel sort, whereas no such guarantee is made in NETWORKKIT.

Additionally, we compare to the sequential TECTONIC implementation [321], which clusters based on the idea of triangle conductance and provides good average precision-recall compared to ground truth communities on SNAP graphs. Like PAR-CC, TECTONIC uses a parameter θ that can be set to achieve a range of clusters with varying average precision and recall. We discuss in more detail in Section 12.4.3 the comparison between the quality of PAR-CC’s and TECTONIC’s clusters, but for outputs where PAR-CC either outperforms or matches TECTONIC in terms of average precision and recall considering $\lambda \in \{0.01x \mid x \in [1, 99]\}$ and $\theta \in \{0.01x \mid x \in [1, 299]\}$ respectively, we see between 2.48–67.62x speedup of PAR-CC over TECTONIC on the graphs amazon, dblp, livejournal, and orkut. Notably, PAR-CC significantly outperforms TECTONIC on large graphs, with between 34.22–67.62x speedups on orkut.

Finally, we compare to SCD [259], a parallel triangle-based community detection implementation. Note that SCD is not able to vary parameters to obtain clusters with significantly different average precision and recall. For amazon, dblp, and livejournal, our PAR-CC implementation achieves 2.00–2.89x speedups over SCD while maintaining the same average precision and recall. For orkut, SCD obtains an average precision of 0.15 and an average recall of 0.05, while PAR-CC can obtain an average precision of 0.61 and an average recall of 0.53 with 1.31x speedup over SCD.

12.4.3 Quality Compared to Ground Truth

Figure 12.14 shows the average precision-recall curves obtained by PAR-CC and PAR-MOD, varying resolutions, compared to the top 5000 ground truth communities on amazon and orkut. For PAR-CC, we set $\lambda \in \{0.01x \mid x \in [1, 99]\}$, and for PAR-

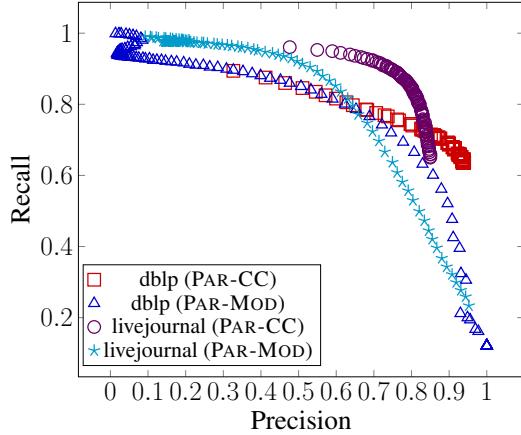


Figure 12.15: Average precision and recall compared to ground truth communities on dblp and livejournal of the clusters obtained from our parallel implementations PAR-CC and PAR-MOD, using varying resolutions.

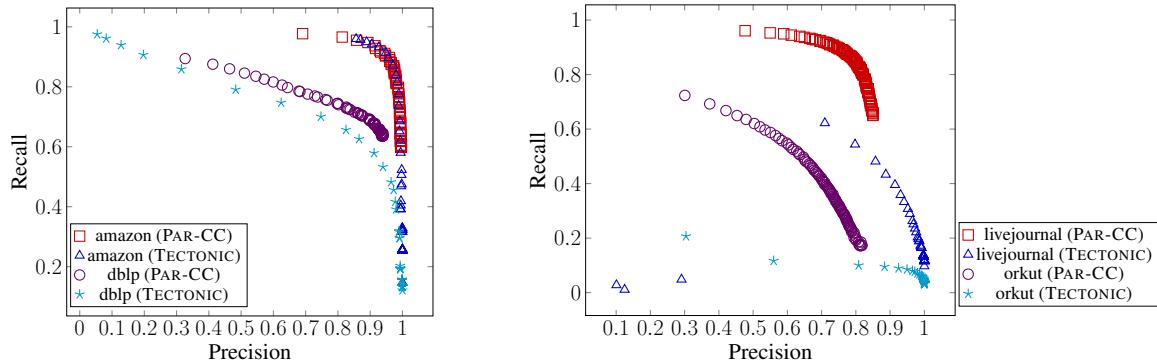


Figure 12.16: Average precision and recall compared to ground truth communities on amazon and dblp (left), and on livejournal and orkut (right), of the clusters obtained from PAR-CC using varying resolutions, and from TECTONIC using varying θ .

MOD, we set $\gamma \in \{0.02 \cdot (1.2)^x \mid x \in [1, 99]\}$. We compare these curves to those obtained by SEQ-CC and SEQ-MOD, running with both the same number of iterations ($num_iter = 10$) as the parallel implementations, and to convergence. We only show SEQ-MOD^{CON}, because the version limiting the number of iterations displays the same average precision-recall curve as the version running to convergence.

Overall, the average precision-recall obtained by PAR-CC and PAR-MOD match those obtained by their sequential counterparts for both amazon and orkut. Note that if we do not run SEQ-CC to convergence, we obtain relatively poor precision-recall compared to PAR-CC, suggesting that more progress is made in fewer iterations in our parallel implementation. This is likely inherent in the behavior in the asynchronous setting of our implementations, where the consistency guarantees are relaxed in a way such that vertices can more easily move better clusters. Overall, PAR-CC offers a better precision-recall trade-off compared to PAR-MOD, which shows the benefits of

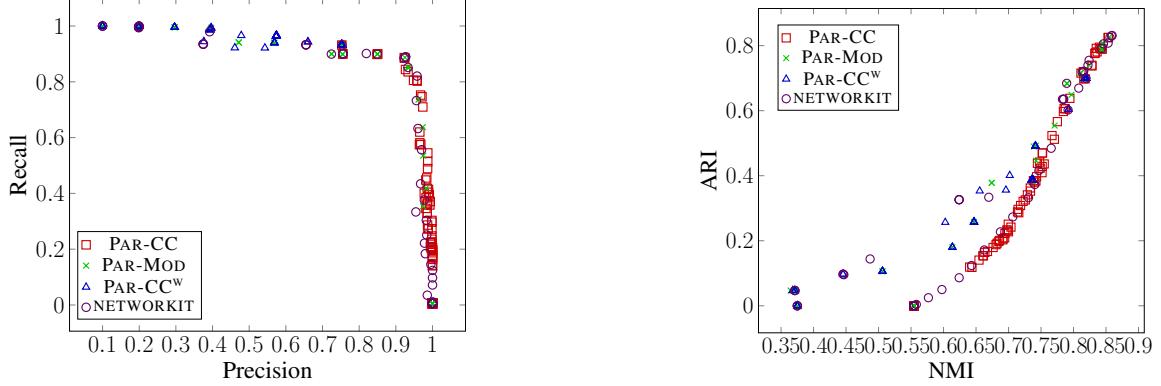


Figure 12.17: Average precision-recall and ARI-NMI scores for the digits graph, from PAR-CC, PAR-MOD, PAR-CC^W, and NETWORKKIT.

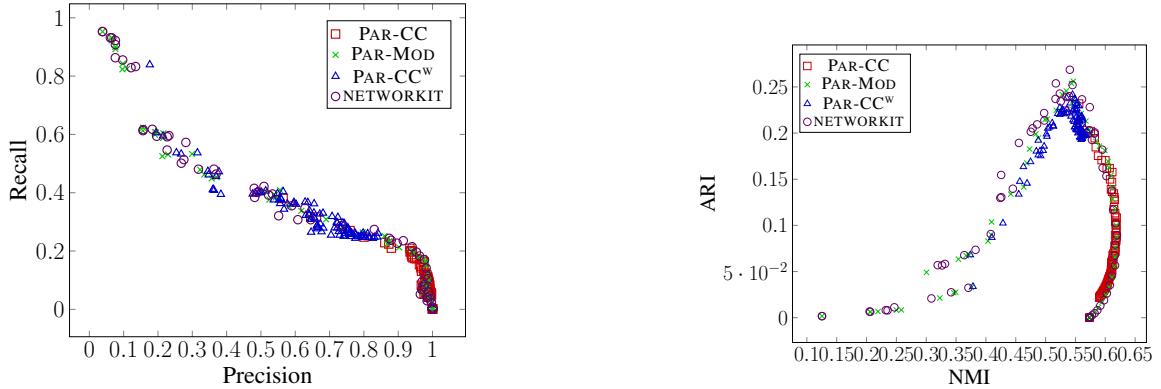


Figure 12.18: Average precision-recall and ARI-NMI scores for the letter graph, from PAR-CC, PAR-MOD, PAR-CC^W, and NETWORKKIT.

using the CC objective. We also show in Figure 12.15 similar behavior in the average precision-recall curves on dblp and livejournal.

Figure 12.16 shows the average precision-recall curves for TECTONIC [321], considering $\theta \in \{0.01x \mid x \in [1, 299]\}$, which we compare to PAR-CC on amazon, dblp, livejournal, and orkut. We see that TECTONIC achieves similar precision-recall trade-offs on amazon, but PAR-CC obtains much better precision-recall on dblp, livejournal, and orkut. Notably, TECTONIC degrades significantly on the larger graphs livejournal and orkut compared to PAR-CC.

Weighted graphs. For weighted graphs G , we test both our implementations treating G as an unweighted graph (with unit weight edges), and our implementations treating G as a weighted graph. We denote the former with no superscript, and the latter with the superscript ^w. The sequential implementations SEQ-CC and SEQ-MOD give similar results to the corresponding parallel implementations, so we discuss only the parallel implementations.

Figures 12.17 and 12.18 show the average precision-recall and ARI-NMI scores for the digits and letter graph respectively. We compare our parallel implementations to NETWORKKIT’s modularity clustering implementation, which can also take as input

weighted graphs [308]. NETWORKKIT matches our PAR-MOD^w implementation, so we omit the latter from our figures. We consider our implementations both treating the graphs as unweighted, and taking into account the edge weights. Moreover, we consider a range of resolutions, with $\lambda \in \{0.01x \mid x \in [1, 99]\}$ for the CC objective, and $\gamma \in \{0.02 \cdot (1.2)^x \mid x \in [1, 99]\}$ for the modularity objective. Overall, compared to NETWORKKIT, PAR-CC^w is more robust across different resolution parameters compared to other implementations.

12.5 Discussion

We have designed and evaluated a comprehensive and scalable parallel clustering framework, which captures both correlation and modularity clustering. Our framework offers settings with trade-offs between performance and quality. We obtained significant speedups over existing state-of-the-art implementations that scale to large datasets of up to billions of edges. Moreover, we showed that optimizing for the correlation clustering objective gives higher-quality clusters with respect to ground truth, compared to other methods in highly-scalable clustering implementations. This shows the significance of the correlation clustering objective for community detection. Finally, we proved the P-completeness of Louvain-like algorithms for parallel correlation and modularity clustering.

Chapter 13

Hierarchical Agglomerative Clustering

13.1 Introduction

Hierarchical agglomerative clustering (HAC) [190, 203, 306] is one of the most commonly used clustering methods. It has gained significant attention among practitioners, as it is a very simple algorithm that delivers high-quality results. The HAC algorithm, given n input points, proceeds in $n - 1$ steps. In each step, it merges together the two most similar points, and thus after $n - 1$ steps, all input points are merged into one.

Equivalently, this process can be described on an immutable set of n points, and a mutable clustering, which initially contains a separate cluster for each input point. Each step then replaces the two most similar clusters by its union. The exact notion of similarity between two clusters is specified by a configurable *linkage function*. This function is typically given all pairwise similarities between points from the two clusters. The most popular choices are *average-linkage* (the arithmetic mean of all similarities), *single-linkage* (the maximum similarity), and *complete-linkage* (the minimum similarity). Among these, average-linkage is of particular importance, as it is known to find high-quality clusters in real-world applications [293, 240, 82, 237].

A major shortcoming of the HAC algorithm is that it is computationally expensive. In particular, a major obstacle to obtaining a highly-efficient HAC implementation is the sequential nature of the algorithm [30, 237], which makes it challenging to parallelize. In fact, all existing attempts at parallelizing HAC either rely on optimistic assumptions on the input [309, 342], or significantly diverge from the original algorithm, potentially impacting quality [78, 30, 237].

Our Contributions. In this chapter we introduce ParHAC – an efficient parallel algorithm for $(1 + \epsilon)$ -approximate average-linkage HAC with poly-logarithmic span and near-linear total work. The algorithm takes as input a similarity graph, which contains n vertices (representing input points) and m weighted edges (representing nonzero similarities between points). We note that the same setting, i.e., parametrizing the input size by the number of *nonzero* similarities, has been recently studied in

the sequential case [100], where an $\tilde{O}(m)$ -work sequential $(1 + \epsilon)$ -approximate algorithm is known.¹

Prior to our result, no average-linkage HAC algorithm with sublinear span was known (even allowing approximation). In addition, we show that allowing approximation is necessary, as obtaining an average-linkage *exact* HAC algorithm with polylogarithmic span is not possible under standard complexity-theory assumptions. We implement and evaluate our algorithm on publicly available real-world datasets and present a comprehensive comparison to other parallel and sequential implementations. In particular, our parallel implementation obtains 50.1x speedup on average compared to the best sequential baseline, while maintaining roughly equal quality.

ParHAC is $(1 + \epsilon)$ -approximate according to the approximation notion proposed by [239]. Namely, an $(1 + \epsilon)$ -*approximate HAC algorithm* is an algorithm which at each step only merges edges that have similarity at least $\mathcal{W}_{\max}/(1 + \epsilon)$ where \mathcal{W}_{\max} is the largest weight currently in the graph.² Hence, an $(1 + \epsilon)$ -approximate algorithm is constrained to merge an edge that is “close” in similarity to the merge the exact algorithm would perform. At the same time, the definition gives the algorithm flexibility in which edge it chooses to merge, which it can exploit to save work or obtain parallelism.

For a fixed ϵ , ParHAC runs in poly-logarithmic span and $\tilde{O}(m)$ -work. Thus, up to logarithmic factors, it is work-efficient, and achieves very high parallelism (ratio of work to span).

Technical Overview. Let us now describe the main ideas behind our algorithm. As observed in [309, 342], a simple exact parallel HAC algorithm can be obtained almost directly from the 40-year old nearest-neighbor chain algorithm [33]. The parallel algorithm finds all edges xy such that xy is the highest-weight incident edge to each endpoint, x and y . For simplicity, we assume here that all edge weights are distinct. One can see that these edges form a matching in the graph (which does not necessarily match all vertices), and the correctness of the nearest-neighbor chain algorithm implies that the endpoints of these edges can be merged in parallel. Following [309], we call this algorithm reciprocal agglomerative clustering (**RAC**).

Clearly, the amount of parallelism in RAC is data-dependent. In particular, it can take a linear number of steps in the worst case [309], and, as we show in Figure 13.1, up to 21,081 steps on the YouTube (YT) real-world graph with just 1.1M vertices and 5.9M edges.

Once we consider $(1 + \epsilon)$ -approximate HAC, the set of edges that we can choose to merge in the first step *grows* to include all edges whose weight is within $(1 + \epsilon)$ factor of \mathcal{W}_{\max} , the largest edge weight in the graph. Let us call these edges $(1 + \epsilon)$ -*heavy*. A major challenge is that the $(1 + \epsilon)$ -heavy edges no longer form a matching, and thus cannot be all merged in parallel. To see this, consider an example where all $(1 + \epsilon)$ -heavy edges have a common endpoint x . Once one vertex merges with x , the size of the cluster represented by x increases, which decreases the weights of all edges

¹We define $\tilde{O}(f(x)) := O(f(x)\text{polylog}f(x))$.

²We note that [239] dealt with dissimilarities, but we adapt the definition to a setting with similarities in the natural way.

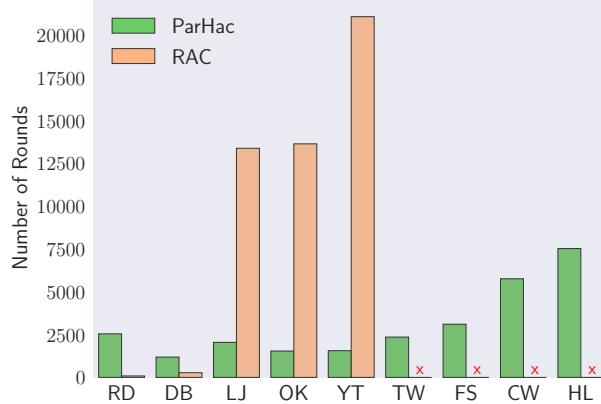


Figure 13.1: The number of rounds required by the **ParHAC** (using $\epsilon = 0.1$) and reciprocal agglomerative clustering (RAC) algorithms for six large real-world graphs. The ClueWeb (CW) and Hyperlink (HL) graphs are two of the largest publicly available graphs, with 978M and 1.72B vertices, and 74B and 124B edges, respectively. We mark experiments where RAC does not finish after 6 hours with a red \times .

incident to x , and as a result some of these edges may cease to be $(1 + \epsilon)$ -heavy.

A key insight is that as long as the size of the cluster represented by x does not grow too much within a single round, the weights of the edges incident to x are very close to what they were in the beginning of the round. Specifically, assume that (in the beginning of a round) x represents a cluster of size c . Then, until the size of this cluster exceeds $(1 + \epsilon) \cdot c$, the edges incident to x that were $(1 + \epsilon)$ -heavy at the beginning of the round remain (at least) $(1 + \epsilon)^2$ -heavy. This allows us to merge multiple vertices with x in a single round, at the cost of increasing the approximation ratio to $(1 + \epsilon)^2$ (which can be reduced to $(1 + \epsilon)$ by scaling ϵ by a constant factor).

We use this observation to give a parallel algorithm that, on average, merges many vertices in a single round. This algorithm and its analysis constitute our main theoretical contribution. At a high level, we argue that the algorithm makes good progress by using the following reasoning. Within each round, for each vertex x with constant probability either a constant fraction of its incident edges decrease weight by a constant factor, or the vertex itself is merged into another vertex representing a cluster of larger size. We observe that either of the two cases can occur at most a logarithmic number of times. As a result, we can guarantee that after a polylogarithmic number of steps, the maximum edge weight in the graph drops by (at least) a constant factor. Assuming the ratio between the largest and the smallest edge weight in the initial graph is $O(\text{poly} n)$, one can easily show that running this algorithm a logarithmic number of times is sufficient to compute $(1 + \epsilon)$ -approximate HAC.

As shown in Figure 13.1, the overall number of rounds required by **ParHAC** is relatively small in practice, which is consistent with our theoretical analysis. In particular, **ParHAC** always runs in several thousand rounds, even on the Hyperlink (HL) graph, which contains 1.7B vertices and 124B edges.

Lower Bound. We omit our lower bounds from this chapter, but we show in our paper [101] that computing average-linkage HAC in poly-logarithmic span is a P-complete problem. In other words, an exact polynomial-work parallel HAC algorithm for average-linkage in poly-logarithmic span would imply poly-logarithmic span algorithms for *all* problems solvable in polynomial time (i.e., show that P = NC). Our lower bound formalizes a commonly held belief that the algorithm is inherently sequential [309] and justifies studying the approximate variant of HAC.

Empirical Evaluation. We provide an efficient parallel implementation of **ParHAC** and compare it with existing graph-based and pointset-based HAC baselines in terms of both quality and scalability. For the Adjusted Rand-Index quality measure, we find that the quality of **ParHAC** is on average 3.1% better than the best approximate sequential baseline, and is on average within 3.69% of the best score for any method on the datasets that we study. We find that **ParHAC** achieves consistently strong quality results on three other quality metrics that we study compared with the best solutions offered by exact and approximate algorithms.

ParHAC also achieves strong speedups relative to other high-quality HAC baselines, achieving 45.2x average speedup over the approximate sequential algorithm from [100] when run on a 72-core machine. We summarize our other empirical results when introducing related work in Section 13.1.1.

Finally, we study the overall running times of using **ParHAC** in the pointset setting, and find that **ParHAC** can cluster significantly larger datasets compared to **fastcluster**, a state-of-the-art pointset clustering algorithm, and achieves up to 417x speedup over **fastcluster** in end-to-end running time.

13.1.1 Related Work

ParHAC is inspired by a recent sequential $(1 + \epsilon)$ -approximate average-linkage HAC algorithm, which runs in $\tilde{O}(m)$ time [100]. However, as outlined in the previous section, obtaining a theoretically-efficient and practical parallel algorithm requires significant new ideas. The empirical evaluation of [100] studied the difference in quality between exact and $(1 + \epsilon)$ -approximate HAC and showed that even moderately small $\epsilon = 0.1$ suffices to roughly maintain the quality of exact HAC. Our results on the quality of **ParHAC** are consistent with these findings.

Parallel HAC Algorithms. Efficient parallelizations of HAC are known in the case of single-linkage (essentially equivalent to maximum spanning forest), and centroid linkage, if one allows $O(\log^2 n)$ -approximation [239]. In other cases, the existing parallelizations of exact HAC either use linear span [248], much larger work [90, 264] or do not come with any bounds on the running time [180, 309, 342]. The ParChain framework [342] for parallel exact HAC on pointsets recently showed using novel caching and pruning techniques that average-linkage HAC can be solved in hours, even for million-point datasets. Running **ParHAC** on a similarity graph built from the pointset yields an average of 39.3x speedup in parallel over ParChain.

Affinity Clustering and SCC. In order to scale hierarchical clustering to large datasets, several HAC-inspired algorithm have been proposed, including Affinity clustering [30] and SCC [237]. Both algorithms are designed for a distributed setting, in which the number of computation rounds that one can afford is highly constrained. In particular, SCC can be seen as a best-effort approximation of HAC, given a fixed (usually small) number of rounds to run.

We implemented both algorithms (using the framework that we built to develop **ParHAC**) and included them in our empirical evaluation. Unsurprisingly, due to the low number of rounds used by these algorithms, they both run faster than **ParHAC**. Specifically, SCC is on average 1.24x faster and Affinity clustering is on average 14.8x faster than **ParHAC**. At the same time, **ParHAC** delivers significantly higher quality results, consistently outperforming Affinity and SCC in almost all cases. This indicates that using an approximate HAC algorithm may have benefits in practical applications. Still, we believe that our highly efficient implementations of Affinity clustering and SCC may be of interest due to their very good running times. We note that all of these algorithms use a similar amount of space in our evaluation, and therefore the main reason to prefer affinity or SCC is for their speed. In particular, Affinity clustering can cluster a 124B-edge Hyperlink graph in under 11 minutes.

RAC. Very recently, it was shown that a distributed implementation of the RAC algorithm can exactly cluster a billion-point dataset in a few hours using 200 machines and 3200 CPUs [309]. However, the span of the algorithm is linear in the worst case, and can be relatively large in practice (see Figure 13.1).

Other HAC Algorithms The theoretical foundations of HAC has been developed in recent years [89, 240, 69], and have motivated using HAC in real-world settings [273, 67, 81, 68, 82]. The version of HAC that takes a graph as input has also been studied before, especially in the context of graphs derived from point sets [155, 188, 135], although without strong theoretical guarantees. Another line of work has focused on breaking the quadratic time barrier for HAC. In a recent breakthrough, Abboud et al. [3] showed that if the input points are in the Euclidean space and the Ward linkage method is used [335], approximate HAC can be solved in subquadratic time.

13.2 Preliminaries

We let $G = (V, E, w)$ denote a weighted graph, where edges are weighted using $w : E \rightarrow \mathbb{R}$. We take all graphs to be weighted, simple (i.e., with no self- or multiple edges), connected, and unless otherwise stated, undirected. We use $\text{Cut}(X, Y)$ to denote the set of edges between two sets of vertices X and Y .

13.2.1 Graph-Based HAC

The *graph-based hierarchical agglomerative clustering (HAC)* problem takes as input a graph $G = (V, E, w)$ and proceeds by repeatedly merging the two most similar

clusters, where the similarity is given by a configurable linkage function.

In this paper we focus on the *average-linkage measure* (sometimes called *unweighted average-linkage (UPGMA-linkage)*), which assumes that the similarity between two clusters (X, Y) is equal to $\sum_{(x,y) \in \text{Cut}(X,Y)} w(x,y) / (|X| \cdot |Y|)$, that is the total weight of edges between X and Y , divided by the maximum number of possible edges between the clusters. A less common variant of average-linkage is *weighted average-linkage (WPGMA-linkage)*. For this measure, the similarity between two clusters depends not only on the current clustering, but also on the sequence of merges that created it. In particular, if a cluster Z is created by merging clusters X and Y , the similarity of the edge between Z and a neighboring cluster U is $\frac{\mathcal{W}(X,U) + \mathcal{W}(Y,U)}{2}$ if both edges (X,U) and (Y,U) exist before the merge, and otherwise just the weight of the single existing edge. We stress that *unweighted* and *weighted* in the linkage measure names refer to the linkage methods. In both cases, throughout this paper we consider graphs with arbitrary non-negative edge weights.

We say that a linkage measure is *reducible* [33], if for any three clusters X, Y, Z , it holds that $\mathcal{W}(X \cup Y, Z) \leq \max(\mathcal{W}(X, Z), \mathcal{W}(Y, Z))$. We note that both unweighted and weighted average-linkage are reducible [33].

When discussing our algorithms, we assume that the graph is mutable (our implementation also follows this idea and uses a mutable graph representation). That is, in each step of our algorithm we merge together the endpoints of a set of (approximately) highest-weight edges. This may affect the weights of the edges incident to the newly created vertices. We note that our algorithm modifies all affected edge weights in the graph to reflect these changes.

In an intermediate step of the algorithm each vertex of the graph corresponds to a cluster containing all vertices that have been merged into it. We say that the *size* of a vertex v , denoted as $|v|$ is the number of vertices in its corresponding cluster. Initially, each vertex has size 1, and the algorithm terminates with a single vertex of size n .

We represent the merges made by the algorithm with a *dendrogram* – a rooted binary tree, which has a single leaf for each vertex in the input graph. Each internal node of the dendrogram represents a cluster obtained in an intermediate step of the algorithm. Merging two vertices (clusters) in the algorithm is reflected in the dendrogram by adding a new node whose two children are the nodes representing the merged clusters. The weight of each internal node in the dendrogram is the linkage similarity used when performing the corresponding merge.

13.3 Sequential HAC

In this section we review the existing exact and $(1 + \epsilon)$ -approximate sequential HAC algorithms which are two baselines that we compare against, and also the starting point for our new theoretical results.

Exact Average-Linkage. A challenge in implementing the average-linkage HAC algorithm is to efficiently maintain edge weights in the graph as vertices (clusters) are *merged*. Since the average-linkage formula includes the cluster sizes of both endpoints

of a (u, v) edge when computing the weight of the edge, an algorithm which eagerly maintains the correct weights of all edges in the graph must update *all* of the edge weights incident to a newly merged cluster. Unfortunately one can easily show that even for sparse graphs with $O(n)$ edges, for some graphs this eager approach requires $\Omega(n^2)$ time.

It was recently shown [100] that exact average-linkage HAC algorithm can be implemented in $\tilde{O}(n\sqrt{m})$ time, using the classic nearest-neighbor chain technique in conjunction with a low-outdegree orientation data structure.

Approximate Average-Linkage (SeqHAC). Our parallel algorithm is inspired by the sequential approximate average-linkage HAC algorithm [100], which runs in near-linear time. The sequential algorithm uses a lazy strategy to avoid updating the weights of all edges incident to each merged vertex. Instead of exactly maintaining the weight of each edge, the algorithm uses the observation that the weights of edges incident to a vertex do not need to be updated every single time the cluster represented by this vertex grows. Namely, if we allow an $(1 + \epsilon)^2$ -approximate algorithm, the weights of edges incident to a vertex do not need to be updated until the cluster-size of this vertex grows by a multiplicative $(1 + \epsilon)$ factor. (The approximation ratio of the algorithm is squared, since the clusters represented by *both* endpoints of an edge may grow by a $(1 + \epsilon)$ factor before its weight is updated.) Since this can happen at most $O(\log_{1+\epsilon} n)$ times, for a constant ϵ , the overall number of edge weight updates is $\tilde{O}(m)$.

The approximate algorithm adapts the folklore *heap-based* approach for computing HAC to the approximate setting. Specifically, the algorithm maintains a heap keyed by the vertices, where the value assigned to a vertex is its current highest-weight incident edge (whose weight is correct up to a $(1 + \epsilon)^2$ factor). In each iteration, the highest-weight edge is chosen from the heap and its endpoints are merged. By setting parameters appropriately one can ensure that the resulting algorithm is $(1 + \epsilon)$ -approximate.

13.4 Parallel Approximate HAC

In this section we present **ParHAC**, a parallel approximate graph-based HAC algorithm for the unweighted average-linkage measure. We start by discussing natural ideas for parallel HAC algorithms and their limitations, and provide a high-level overview of our algorithm.

13.4.1 Overview

Both the sequential exact average-linkage algorithm and the $(1 + \epsilon)$ -approximate average-linkage algorithm discussed in Section 13.3 are in some sense inherently sequential. Specifically, both algorithms use a greedy technique to find the edge to merge. For the exact algorithm this sequential behavior is, as our lower-bound in the paper [101] shows, in some sense unavoidable. For the $(1 + \epsilon)$ -approximate algorithm,

the use of a heap to extract the next pair of vertices to merge causes the span to be $\Omega(n)$ in the worst case, which is far from ideal.

Geometric Bucketing. The starting point for our approach is to observe that as in the $(1 + \epsilon)$ -approximate sequential algorithm, we can merge *any* pair of vertices whose similarity is within a $(1 + \epsilon)$ factor of the current highest-similarity edge. This idea motivates the use of an important technique in parallel approximation algorithms known as *geometric bucketing*, where we group the edges based on their weights and process all edges within the same bucket in parallel. Geometric bucketing is a powerful technique in designing parallel approximation algorithms (e.g., see [35, 50]), but requires care when selecting items within the same bucket to ensure that the approximation quality is preserved.

Let \mathcal{W}_{\max} and \mathcal{W}_{\min} be the maximum-weight and minimum-weight in the graph, respectively. In the geometric bucketing scheme, the i -th bucket contains all edges with weight between $((1 + \epsilon)^{-(i+1)} \cdot \mathcal{W}_{\max}, (1 + \epsilon)^{-i} \cdot \mathcal{W}_{\max}]$. The geometric bucketing algorithm then processes these buckets one-by-one, starting with the heaviest-weight bucket. An **active vertex** is a vertex that has edges in the bucket currently being processed by the algorithm. Under the assumption that the **aspect-ratio** of the graph is polynomially bounded, that is $\mathcal{A} = \mathcal{W}_{\max}/\mathcal{W}_{\min} = O(\text{poly}(n))$, we can show that the maximum number of buckets processed by the geometric bucketing algorithm is $O(\log n)$ for any constant ϵ . We note that on the real-world datasets evaluated in this paper, the aspect ratio is at most 1000. The key challenge for a parallel approximation algorithm based on this paradigm is that it must process all of the elements within each bucket in $\text{polylog}(n)$ rounds, while ensuring that it does not violate the $(1 + \epsilon)$ approximation requirements.

Natural Approaches: Spanning Forest and Affinity. A natural idea is to compute a spanning forest induced by the edges within the current bucket, and to merge together all vertices in each tree in the forest. Using this approach, all edges within the current bucket can be processed in a single spanning-forest step. Furthermore, after applying these merges all remaining edges in the graph fall into buckets of smaller weight. Using a work-efficient spanning forest algorithm, the overall work and span of this approach will be $\tilde{O}(m)$ and $O(\text{polylog}(n))$, respectively [301].

A similar idea is used in the *Affinity Clustering* algorithm of Batani et al. [30]. In this algorithm, each vertex marks its highest-weight incident edge, and the sets of vertices to be merged are the connected components of the graph formed by the marked edges. It is easy to argue that this approach yields a $\tilde{O}(m)$ work and $O(\text{polylog}(n))$ span algorithm in the geometric bucketing scheme.

Unfortunately, both of these natural approaches fail to yield $(1 + \epsilon)$ -approximate algorithms because they both choose “locally good” edges to merge without taking into account how the similarities of these edges change as they are mapped to binary merges in the dendrogram. For example, if the input graph is a path consisting of edges of the same weight, the spanning forest approach would merge all of the vertices together in one step. At the same time it is easy to see that for $\epsilon < 1/2$ an $(1 + \epsilon)$ -approximate algorithm, within a single bucket, must merge sets of vertices of size at most 2.

Our Approach: Capacitated Random Mate. Our approach is inspired by the classic random-mate technique when processing each bucket, but requires a careful modification to yield good approximation guarantees. Our algorithm starts by first randomly coloring the active vertices red and blue with equal probability. Directly applying the random-mate approach (e.g., as applied in the parallel connectivity algorithms of Reif [267] and Phillips [257]) would suggest merging all blue vertices into an arbitrary red neighbor. The number of edges in the bucket decreases by a constant factor in expectation, thus yielding the desired work and span bounds. Unfortunately, this approach does not yield good approximation guarantees for the same reason that the spanning forest and affinity approaches fail to do so. The issue is that the size of a red cluster can become too large, causing the similarity of a merged edge to be much smaller than $W_{\max}/(1 + \epsilon)$, where W_{\max} is the *current largest edge weight*.

Our *capacitated random-mate* approach fixes this issue with the classic random-mate strategy by processing each bucket in multiple *rounds* and enforcing vertex-capacities for the red vertices in each round. These capacities ensure that the sizes of clusters represented by red vertices increase by at most a $(1 + \epsilon)$ factor within a round. As a result, the weight of each edge incident to a red vertex may only decrease by a $(1 + \epsilon)$ factor.

Specifically, consider a red vertex v representing a cluster C . Assume that at the beginning of a round all edge weights in the graph are computed exactly and the size of the cluster C is c_0 . Within the round, vertex v only accepts a merge proposal from a neighboring blue vertex if the total size of the cluster C , including the growth incurred by previous proposals accepted within this round, is smaller than $(1 + \epsilon)c_0$. This approach ensures that the weight of any merged edge is close to W_{\max} , and thus lets us argue that our algorithm is $(1 + \epsilon)$ -approximate. However, using this capacitated approach complicates the analysis of the round-complexity of our algorithm, making it more challenging to bound the overall work and span. Our main theoretical contribution is to show that the capacitated approach results in a near-linear work and poly-logarithmic span algorithm that achieves a $(1 + \epsilon)$ -approximation for any $\epsilon > 0$.

13.4.2 ParHAC Algorithm

Algorithm 13.1 shows the pseudocode for our algorithm. We highlight the parts of the algorithm which utilize parallelism in light blue. The algorithm follows the geometric bucketing scheme described earlier. The main loop of the algorithm (Lines 2–4) computes W_{\max} , the largest weight of an edge in the current graph (Line 3), and processes all edges with weights between $[(1 + \epsilon)^{-1} \cdot W_{\max}, W_{\max}]$ in the current bucket using the **ParHAC-BucketMerge** function (Line 4). We refer to an iteration of this outer loop as a *bucket-processing phase*.

The majority of the work in the algorithm is done in the bucket-processing routine, **ParHAC-BucketMerge**, within the context of one bucket-processing phase. This algorithm repeatedly applies capacitated random-mate steps to the graph induced by edges in the current bucket until the bucket becomes empty. While the maximum-

Algorithm 13.1 – ParHAC($G = (V, E, w), \epsilon$)

Require: Similarity graph G , $\epsilon > 0$.

Ensure: $(1 + \epsilon)$ -approximate dendrogram for unweighted average-linkage HAC.

- 1: Let D be initialized to the identity clustering.
 - 2: **while** $|E| > 0$ **do**
 - 3: Let W_{\max} be the current maximum-weight edge in the graph.
 - 4: Call ParHAC-BUCKETMERGE($G, (1 + \epsilon)^{-1} \cdot W_{\max}, \epsilon, D$).
 - 5: **return** D
-

Algorithm 13.2 – ParHAC-BucketMerge($G = (V, E, w), T_L, \epsilon, D$)

Require: Similarity graph, G , threshold T_L , $\epsilon > 0$, dendrogram D .

Ensure: All edges in G have weight $< T_L$.

- 1: Let W_{\max} be the current maximum-weight edge in G .
 - 2: **while** $W_{\max} \geq T_L$ **do** ▷ Outer round
 - 3: Randomly color active vertices of G either red or blue.
 - 4: Let R, B be the sets of red and blue vertices respectively.
 - 5: Let $G_c = (V, E_c, w)$, where E_c contains all edges in G that have weight $\geq T_L$, and connect a vertex $x \in B$ with a vertex $y \in R$, where the size of y is not smaller than the size of x .
 - 6: **while** $|E_c| > 0$ **do** ▷ Inner round
 - 7: Select a random priority π_b for $b \in B$.
 - 8: Let C_b be a random red neighbor for each $b \in B$.
 - 9: Let $T = \{(C_b, \pi_b, b) \mid b \in B\}$. Sort T lexicographically.
 - 10: Let T_r be triples from T with first component equal to r .
 - 11: For each $r \in R$, select the first prefix of T_r , in which the total size of blue vertices exceeds $\epsilon|r|$
 - 12: Let M be the set of (red, blue) vertex pairs selected.
 - 13: Merge vertices in G and G_c based on the pairs from M , updating edge weights in G_c .
 - 14: Remove edges of G_c that have two red endpoints or weight below T_L
 - 15: Update D based on M . If multiple $b \in B$ merge to a single $r \in R$, merge them into r in the sorted order.
 - 16: Remove $r \in R$ from G_c whose cluster size grew by more than a $(1 + \epsilon)$ factor since the start of the outer round.
 - 17: Recompute W_{\max} based on the current state of G .
-

weight edge in the current graph is larger than the lower-threshold for this bucket, T_L , the algorithm performs an outer-round of the algorithm. An **outer-round** of the algorithm begins by randomly coloring active (i.e., non-isolated) vertices either red or blue (Line 3), assigning each color with probability $1/2$.

Then, it constructs a graph G_c which consists of edges of G of weight belonging to the current bucket (Line 5). Moreover, G_c only contains edges whose both endpoints have different colors, and whose red endpoint has larger size than the blue endpoint. Observe that each edge of G is added to G_c with probability at least $1/4$. The algorithm then performs inner-rounds while the number of edges in G_c is non-zero.

Let us now describe a single *inner-round* (Lines 6–16). For each blue vertex $b \in B$, the algorithm selects a uniformly random priority $\pi_b \in [0, 1]$ (Line 7) which is used to perform symmetry breaking when merging vertices. It then chooses a random red neighbor, C_b , for each $b \in B$ (Line 8). If a blue vertex has no red neighbors it will not participate in the subsequent steps, but for simplicity when describing the algorithm we assume that each $b \in B$ has a valid C_b . The algorithm then builds T , a set of triples for each $b \in B$ containing the value of its candidate red neighbor C_b , its priority π_b , and its id, b (Line 9), and sorts this set lexicographically. We call the elements of T *proposals*. At this point, a red vertex may have proposals to merge from a large number of blue neighbors, and so the algorithm selects for each $r \in R$ the first prefix of T_r (the merges proposing to r) whose total cluster size exceeds $\epsilon|r|$ (Lines 10–11). Note that if no prefix of T_r has this property, the algorithm selects all of T_r .

Finally, the algorithm gathers all of the (r, b) vertex pairs that were selected (Line 12) and applies these merges to update both G and G_c (Line 13). Note that after this update we remove all edges of G_c that have two red endpoints, and as a result we maintain the invariant that each edge of G_c has two endpoints of distinct colors, and the size of the red endpoint is at least the size of the blue one.

The algorithm then applies these merges to the dendrogram D . For each red vertex that participates in a merge in this inner round, if multiple blue vertices merge to it, the algorithm orders these merges in the dendrogram in the sorted order (Line 15). This can be achieved by adding a path to the dendrogram and connecting one leaf to each internal vertex of this path. The last step within an inner-round removes all $r \in R$ whose cluster size has grown by more than a $(1 + \epsilon)$ factor since the start of the current outer round (Line 16).

Theoretical Analysis. Next, we show that our algorithm performs nearly-linear total work and has poly-logarithmic span, and thus polynomial parallelism. We note that for the purpose of this analysis we assume that $\epsilon > 0$ is a constant. We start with a lemma bounding the number of buckets that the algorithm processes.

Lemma 13.1. *Assuming that the aspect-ratio of the graph, $\mathcal{A} = O(\text{poly}(n))$, the number of bucket-processing phases is $O(\text{polylog}(n))$.*

Proof. Since the unweighted average-linkage function is reducible, the largest weight can only decrease over the course of the algorithm. Furthermore, the smallest weight that the algorithm can encounter using the unweighted average-linkage function is proportional to $O(\mathcal{W}_{\min}/n^2)$. Thus the weight range that the algorithm runs over is $\mathcal{W}_{\max}/(\mathcal{W}_{\min}/n^2) = O(\text{poly}(n))$, and only $O(\log n)$ buckets are required to bucket every weight in this weight range. \square

Next, we focus on a single bucket-processing phase and bound the total work and number of rounds required. We start by analyzing a single inner round of Algorithm 13.2 and show that within each inner round, each blue vertex makes progress in expectation either by being merged, or having many edges incident to it be deleted.

Lemma 13.2. Consider an arbitrary blue vertex b within an inner round. Within this round either (a) a constant factor of edges incident to b are deleted, or (b) with constant probability b is merged into one of its red neighbors.

Proof. Fix the random priority π_b for each $b \in B$. We show that the lemma follows for any set of distinct priorities, not necessarily random ones. It is easy to see that our algorithm is equivalent to processing the blue vertices in the order of their priorities and deciding for each of them whether it merges to a red neighbor. We say that a red vertex is *saturated* if its size has increased by more than $(1 + \epsilon)$ factor since the beginning of the outer round. Note that each saturated red vertex is removed at the end of the inner iteration.

Consider a blue vertex b . If more than half of red neighbors of b are saturated, we are in case (a). Otherwise, with constant probability b chooses a non-saturated red neighbor and can merge with it. \square

Next, we bound the number of inner rounds within an outer round.

Lemma 13.3. There are at most $O(\log n)$ inner rounds within each outer round with high probability.

Proof. Fix a blue vertex b . There can be at most $O(\log n)$ inner rounds when the vertex satisfies case (a) of Lemma 13.2, and at most $O(\log n)$ times (with high probability) when it satisfies case (b). After that, the vertex is either merged into its red neighbor or becomes isolated. In both cases, all incident edges are removed. \square

Lemma 13.4. The number of outer rounds within a call to Algorithm 13.2 is $O(\log n)$ with high probability.

Proof. Consider the **while** loop in Line 6. We first prove the following claim: for each edge e of the graph G_c at the beginning of the loop, either (a) the endpoints of e are merged together, or (b) the weight of e drops below T_L , or (c) an endpoint of e increases its size by a factor of $(1 + \epsilon)$.

To prove this fact, we observe that when the loop terminates, all edges of G_c have been removed. Clearly, an edge can be removed when its endpoints are merged together or when its weight drops below T_L , which corresponds to cases (a) and (b). There are two more ways of removing an edge. First, an edge can be removed on Line 16 when its red endpoint grows by a factor of $(1 + \epsilon)$, which leads to case (c). Second, the edge can be removed when it connects two red vertices. This can only happen as a result of a blue vertex merging into a red vertex. However, since for each edge in G_c the red endpoint has size not smaller than the blue one, whenever a blue vertex merges into a red one, the size of its cluster doubles. This finishes the proof of the claim.

From this claim, we immediately conclude that for a fixed weight bucket, any edge e of G can participate in at most $O(\log n)$ inner rounds, as each endpoint of e can increase its size at most $O(\log n)$ times. Since within an outer round, e is included in G_c with constant probability, we conclude there can be at most $O(\log n)$ outer rounds with high probability. \square

We now bound the work and span of an outer round.

Lemma 13.5. *Each outer round can be implemented in $O(m \log n + n \log^2 n)$ expected work and $O(\log^2 n)$ span with high probability.*

Proof. Computing G_c (Line 5) and updating G and G_c within (Line 13) can be done in $O(m)$ work and $O(\log n)$ span using standard parallel primitives such as prefix-sum and filter [47]. Using a parallel comparison sort, Line 9 costs $O(n \log n)$ work and $O(\log n)$ span [47]. The remaining steps can be implemented in $O(n)$ work and $O(\log n)$ span using standard parallel primitives. Combining the work-span analysis with the fact that there are $O(\log n)$ inner rounds w.h.p. by Lemma 13.3 completes the proof. \square

Putting all of the previous results together, we obtain the following theoretical guarantees on the work and span of our algorithm:

Theorem 13.1. *ParHAC performs $\tilde{O}(m + n)$ work in expectation and has $O(\log^4(n))$ span with high probability.*

Approximation Quality. Finally, we show that the algorithm achieves a good approximation with respect to the sequential unweighted average-linkage HAC algorithm.

Theorem 13.2. *ParHAC is an $(1 + \epsilon)$ -approximate unweighted average-linkage HAC algorithm.*

Proof. Let $1 + \delta$ be the approximation parameter used internally in the algorithm, which we will set shortly as a function of ϵ . Consider the merges performed within an inner round in the algorithm. In the algorithm these merges only occur between blue vertices and their red neighbors that they are connected to with an edge with weight at least T_L (the lower bucket-threshold). At the start of a bucket-processing phase the maximum similarity in the graph is W_{\max} and we have by construction that $(1 + \delta)^{-1}W_{\max} \leq T_L$. Furthermore, since the unweighted average-linkage function is reducible the maximum similarity in the graph throughout the rest of this bucket-processing phase is always upper-bounded by W_{\max} .

Consider the merges done within one inner round of the algorithm. We know that for each red vertex r that participates in merges in this inner round, its cluster size never exceeds $(1 + \delta)S_I(r)$ where $S_I(r)$ is the cluster-size of r at the start of the outer round. Furthermore, since we maintain the exact weights of all edges in both G and G_c after every inner round finishes (in Line 13), we know that the weight of each (b_i, r) edge for each blue vertex b_i merging into r in this round is at least T_L . Therefore, the smallest value this weight can attain in a merge is $T_L/(1 + \delta)$, which is lower-bounded by $(1 + \delta)^{-2}W_{\max}$. Therefore, our approach yields a $(1 + \delta)^{-2}$ -approximate algorithm. Now, for $\epsilon \leq 1$ we set $\delta = \epsilon/3$, and for $\epsilon > 1$, we set $\delta = \sqrt{\epsilon}/3$. This way we obtain an $(1 + \epsilon)$ -approximate algorithm for unweighted average-linkage HAC for any $\epsilon > 0$. \square

13.5 ParHAC Implementation

We implemented ParHAC in C++ using the recently developed *CPAM* (Compressed Parallel Augmented Maps) framework [96], which lets users construct compressed and highly space-efficient ordered maps and sets. We build on CPAM’s implementation of the Aspen framework [99, 96], which provides a lossless compressed dynamic graph representation that supports efficient updates (batch edge insertions and deletions). We also use the ParlayLib library for parallel primitives such as reduction, prefix-sum, and sorting [46].

Geometric Bucketing. Instead of explicitly extracting the edges in each bucket and constructing the graph G_c in each outer round, we opted for a simpler approach which works as follows: for each bucket-processing phase in the algorithm we first (i) compute the set of vertices that are active in this phase, i.e., have at least one incident edge in the current bucket and then (ii) repeatedly run the capacitated random-mate steps for an inner round from Algorithm 13.2, recomputing the weights of *all affected edges* in the graph at the end of each inner round. Our implementation modifies Algorithm 13.2 by removing Line 6, which effectively fuses the outer and inner rounds. After each round, we check whether each vertex is still active, and continue within this bucket until no further active vertices remain.

Compressed Clustered Graph Representation. We observed that in practice our ParHAC algorithm can perform a large number of rounds per-bucket in the case where ϵ is very small (e.g., $\epsilon = 0.01$). Although updating the entire graph on each of these rounds is theoretically-efficient (the algorithm will only perform $\tilde{O}(m + n)$ work), many of these rounds only merge a small number of vertices, and leave the majority of the edges unaffected. Hence, updating the weights of all edges in each round can be highly wasteful.

Instead, we designed an efficient *compressed clustered graph representation* using purely-functional compressed trees from the CPAM framework [96]. The new data structure handles all merge operations that affect the underlying similarity graph in work proportional to the number of merged vertices and their incident neighbors rather than proportional to the total number of edges in the graph. Importantly, using CPAM enables lossless compression for integer-keyed maps to store the cluster adjacency information using just a few bytes per edge.³

Our data structure stores the vertices in an array, and stores the current neighbors of each vertex in a weight-balanced compressed purely-functional tree called a PaC-tree [96]. Each vertex also stores several extra variables that store its current cluster size, and variables that help build the dendrogram, such as the current cluster-id of each vertex. There are two key operations that we provide in the compressed clustered graph representation:

- (1) *MultiMerge*: given a sequence M of (r, b) vertex merges where r is a red vertex and b is a blue vertex, update the graph based on all merges in M .

³We find a 2.9x improvement in space-usage by using our CPAM-based implementation over an optimized hashtable-based implementation of a clustered graph, while the running times of both implementations are essentially the same.

- (2) *Neighborhood Primitives*: apply a function f (e.g., map, reduce, filter) in parallel over all current neighbors of a vertex.

For example, our implementation of Algorithm 13.2 uses a parallel map operation over the neighbors of all blue vertices to generate a sequence of merges, which it then supplies to the MultiMerge primitive. Additionally, in our implementation, all of the details of maintaining the dendrogram, keeping track of the size of each cluster, and maintaining the current state of the underlying weighted similarity graph are handled by the compressed clustered graph representation. This enables us to write high-level code for our ParHAC implementation while handling the more low-level details about efficiently merging vertices within the clustered graph code.

Other Parallel Graph Clustering Algorithms. Our new graph representation makes it very easy to implement other parallel graph clustering algorithms. In particular, we developed a faithful version of the Affinity clustering algorithm [30] and the recently proposed SCC algorithm [237] (which is essentially a thresholded version of Affinity) using a few dozens of lines of additional code. Both algorithms are essentially heuristics that are designed to mimic the behavior of HAC, while running in very few rounds (an important constraint for the distributed environments these algorithms are designed for). We note that most of the work done by these algorithms is the work required to merge clusters in the underlying graph, and so by using the same primitives for merging graphs, we eliminate a significant source of differences when comparing algorithms.

The Affinity clustering algorithm [30] is inspired by Boruvka’s MST algorithm [57]. In each round of Affinity clustering, each vertex selects its heaviest incident edge, and all connected components induced by the chosen edges are merged to form new clusters. This process continues until no further edges remain in the graph. After computing best edges, our implementation uses a highly optimized parallel union-find connectivity algorithm from the ConnectIt framework [102] to compute a unique vertex per Affinity tree, which is used as a ‘red’ vertex in the MultiMerge procedure, with all other vertices in its component being ‘blue’ vertices.

The SCC algorithm [237] is closely related to the Affinity algorithm, and can be viewed as running Affinity with different weight thresholds. Specifically, in the i -th round, the SCC algorithm runs a round of Affinity on the graph induced by all edges with weight at least T_i where T_i is the weight threshold on the i -th round. Given the maximum number of rounds R , and the upper threshold U and lower threshold L , the SCC algorithm runs a sequence of R rounds where $T_i = U \cdot (L/U)^{R-i}$. As with Affinity, if the graph becomes empty before all R iterations are run, the algorithm terminates early. We note that our implementation is a best-effort approximation of the SCC algorithm, since SCC was originally designed for the dissimilarity setting and there is no one-to-one mapping to the similarity setting. We refer to our implementation of SCC as SCC_{sim} .

We note that our initial implementations of Affinity and SCC_{sim} were developed in the GBBS framework for static graph processing [98], and did not make use of the compressed clustered graph representation. These initial implementations recomputed the weights of *all* edges in the graph in each round. We found that our new

| Graph Dataset | Num. Vertices | Num. Edges | Avg. Degree |
|-------------------------|---------------|-----------------|-------------|
| <i>com-DBLP (DB)</i> | 425,957 | 2,099,732 | 4.92 |
| <i>YouTube-Sym (YT)</i> | 1,138,499 | 5,980,886 | 5.25 |
| <i>USA-Road (RD)</i> | 23,947,348 | 57,708,624 | 2.40 |
| <i>LiveJournal (LJ)</i> | 4,847,571 | 85,702,474 | 17.6 |
| <i>com-Orkut (OK)</i> | 3,072,627 | 234,370,166 | 76.2 |
| <i>Twitter (TW)</i> | 41,652,231 | 2,405,026,092 | 57.7 |
| <i>Friendster (FS)</i> | 65,608,366 | 3,612,134,270 | 55.0 |
| <i>ClueWeb (CW)</i> | 978,408,098 | 74,744,358,622 | 76.3 |
| <i>Hyperlink (HL)</i> | 1,724,573,718 | 124,141,874,032 | 71.9 |

Table 13.1: Graph inputs, including the number of vertices (n), number of directed edges (m), and the average degree (m/n).

implementations using the compressed clustered graph, which only recompute the weights of edges incident to a merge, are between 7–11x faster across the graphs we evaluate.

13.6 Empirical Evaluation

In this section, we describe our empirical results on a set of real-world graphs and also discuss related empirical work.

Graph Data. We list information about graphs used in our experiments in Table 13.1. ***com-DBLP (DB)*** is a co-authorship network sourced from the DBLP computer science bibliography. ***YouTube (YT)*** is a social-network formed by user-defined groups on the YouTube site. ***LiveJournal (LJ)*** is a directed graph of the social network. ***com-Orkut (OK)*** is an undirected graph of the Orkut social network. ***Friendster (FS)*** is an undirected graph describing friendships from a gaming network. All of the aforementioned graphs are sourced from the SNAP dataset [207].⁴ ***USA-Road (RD)*** is an undirected road network from the DIMACS challenge [93].⁵ ***Twitter (TW)*** is a directed graph of the Twitter network, where edges represent the follower relationship [200].⁶ ***ClueWeb (CW)*** is a web graph from the Lemur project at CMU [54].⁷ ***Hyperlink (HL)*** is a hyperlink graph obtained from the WebDataCommons dataset where nodes represent web pages [233].⁸ We note that the large real-world graphs that we study are not weighted, and so we set the similarity of an edge (u, v) to $\frac{1}{\log(\deg(u)+\deg(v))}$. For the CW and HL graphs, which contain tens to hundreds of billions of edges, due to memory constraints on the machine we use, we set the *initial edge weights* to 1 and use byte-codes to store the weights in a number of bytes proportional to their size [302, 96]. We emphasize that the *aggregate*

⁴Source: <https://snap.stanford.edu/data/>.

⁵Source: <http://www.dis.uniroma1.it/challenge9/>

⁶Source: <http://law.di.unimi.it/webdata/twitter-2010/>.

⁷Source: <https://law.di.unimi.it/webdata/clueweb12/>.

⁸Source: <http://webdatacommons.org/hyperlinkgraph/>.

| Dataset | ParHAC \mathcal{E} | ParHAC $_{0.1}$ | RAC and SeqHAC \mathcal{E} | SeqHAC $_{0.1}$ | Affinity | SCC _{sim} | Sci-Single | Sci-Complete | Sci-Avg | Sci-Ward |
|----------|----------------------|-----------------|------------------------------|-----------------|-----------|--------------------|--------------|--------------|--------------|------------------|
| ARI | <i>iris</i> | 0.892 | <u>0.911</u> | 0.873 | 0.873 | 0.599 | 0.786 | 0.715 | 0.642 | 0.759 |
| | <i>wine</i> | 0.401 | 0.401 | 0.401 | 0.401 | <u>0.416</u> | 0.374 | 0.298 | 0.371 | 0.352 |
| | <i>digits</i> | <u>0.912</u> | 0.896 | 0.895 | 0.895 | 0.625 | 0.851 | 0.661 | 0.479 | 0.690 |
| | <i>cancer</i> | 0.440 | 0.491 | 0.447 | 0.447 | 0.375 | 0.197 | <u>0.561</u> | 0.465 | 0.537 |
| | <i>faces</i> | <u>0.621</u> | 0.618 | 0.610 | 0.610 | 0.460 | 0.607 | 0.468 | 0.472 | 0.529 |
| NMI | <i>iris</i> | 0.858 | <u>0.876</u> | 0.842 | 0.842 | 0.692 | 0.780 | 0.734 | 0.722 | 0.806 |
| | <i>wine</i> | 0.409 | 0.409 | 0.394 | 0.394 | 0.426 | 0.400 | 0.410 | <u>0.442</u> | 0.428 |
| | <i>digits</i> | <u>0.928</u> | 0.916 | 0.913 | 0.931 | 0.768 | 0.859 | 0.562 | 0.711 | 0.830 |
| | <i>cancer</i> | 0.445 | <u>0.464</u> | 0.445 | 0.445 | 0.412 | 0.325 | 0.316 | 0.428 | 0.456 |
| | <i>faces</i> | <u>0.879</u> | 0.874 | 0.873 | 0.873 | 0.858 | 0.867 | 0.848 | 0.849 | 0.861 |
| Purity | <i>iris</i> | 0.931 | <u>0.943</u> | 0.920 | 0.920 | 0.764 | 0.884 | 0.843 | 0.791 | 0.869 |
| | <i>wine</i> | 0.619 | 0.623 | 0.605 | 0.605 | 0.616 | <u>0.630</u> | 0.584 | 0.607 | 0.620 |
| | <i>digits</i> | <u>0.904</u> | 0.883 | 0.891 | 0.891 | 0.729 | 0.811 | 0.737 | 0.562 | 0.755 |
| | <i>cancer</i> | 0.812 | 0.823 | 0.797 | 0.797 | 0.764 | 0.703 | 0.798 | 0.804 | <u>0.829</u> |
| | <i>faces</i> | <u>0.640</u> | 0.613 | 0.618 | 0.621 | 0.538 | 0.601 | 0.566 | 0.502 | 0.623 |
| Dasgupta | <i>iris</i> | 320665 | 320883 | 320665 | 320665 | 362177 | 322308 | 314445 | 323384 | <u>310957</u> |
| | <i>wine</i> | 29114 | 29093 | 29114 | 29114 | 29468 | 27095 | 27891 | 27745 | <u>27324</u> |
| | <i>digits</i> | 243841216 | 243166244 | 243840641 | 243837090 | 244130381 | 244983907 | 244836138 | 243726701 | <u>240476750</u> |
| | <i>cancer</i> | 789808 | 751107 | 789808 | 789808 | 794858 | 952966 | 841428 | 742226 | <u>737071</u> |
| | <i>faces</i> | 4629934 | 4632156 | 4629934 | 4622371 | 4674997 | 4669787 | 4640884 | 4600388 | <u>4569916</u> |

Table 13.2: Adjusted Rand-Index (ARI), Normalized Mutual Information (NMI), Dendrogram Purity, and Dasgupta cost of our new ParHAC implementations (columns 2–3) versus the RAC and SeqHAC implementations (columns 4–5), our Affinity and SCC_{sim} implementations (columns 6–7), and the HAC implementations from SciPy (columns 8–11). Note that both RAC and SeqHAC \mathcal{E} are exact HAC algorithms, and thus compute the same dendrogram. The scores are calculated by evaluating the clustering generated by each cut of the generated dendrogram to the ground-truth labels for each dataset. All graph-based implementations are run over the similarity graph constructed from an approximate k -NN graph with $k = 10$. The Dasgupta cost is computed over the complete similarity graph generated from the all-pairs distance graph, and thus takes into account all pairwise similarities. We display the best quality score for each graph in green and underlined.

weights on edges (the number of edges in the original graph that each edge between clusters represents) as the algorithm progresses grow *significantly larger*, and do not simply stay fixed at 1.

We also consider graphs generated from a pointset by computing the approximate nearest neighbors (ANN) of each point, and converting the distances to similarities. Our graph building process converts distances to similarities using the formula $\text{sim}(u, v) = \frac{1}{1 + \text{dist}(u, v)}$.⁹ It then reweights similarities by dividing each similarity by the maximum similarity. In our implementation, we compute the k -approximate nearest neighbors using ParANN [109] a new shared-memory parallel implementation of the *Vamana* approximate nearest neighbors (ANN) algorithm [179] with parameters $R = 75$, $L = 100$, $Q = \max(L, k)$. We note that this parameter setting yields almost perfect recall on the SIFT datasets for the 10-nearest neighbors. More details about the quality of the Vamana algorithm can be found on ANN-Benchmarks [21]. The datasets that we use are sourced from the `sklearn.datasets` package¹⁰ and the Glove-100 dataset from the ANN-Benchmarks collection [21]. We symmetrized all directed graph inputs studied in this paper.

Experimental Setup. We ran all of our experiments on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with $4 \times 2.4\text{GHz}$ Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use a lightweight parallel scheduler based on the Arora-Blumofe-Plaxton deque [20, 46]. For parallel experiments, we use `numactl -i all` to balance the memory allocations across the sockets.

HAC Algorithms. We compare our **ParHAC** algorithm with several HAC algorithm baselines. **SeqHAC** is the approximate sequential average-linkage algorithm that was recently introduced [100]. **ParHAC** _{ϵ} and **SeqHAC** _{ϵ} are exact versions of the **ParHAC** and **SeqHAC** algorithms, where the **ParHAC** code takes $T_L = W_{\max}$ in each bucket-processing phase and sets $\epsilon = 0$ (i.e., the bucket only consists of equal-weight edges) and the **SeqHAC** code computes the exact HAC dendrogram. The **ParHAC** _{ϵ} and **SeqHAC** _{ϵ} dendograms can differ since there may be multiple neighboring edges within a single bucket-processing phase in **ParHAC** _{ϵ} , which can result in **ParHAC** _{ϵ} making different merge choices compared to **SeqHAC** _{ϵ} . **ParHAC**_{0.1} and **SeqHAC**_{0.1} both use $\epsilon = 0.1$.

We also compare our results with our implementations of Affinity clustering [30] and the recently developed SCC algorithm [237]. We describe the algorithms in detail in Section 13.5. We run the **SCC**_{sim} algorithms for 100 iterations, which we found yielded results that are less noisy than using fewer iterations. We note that affinity clustering requires at most 10 rounds on all graphs we evaluate on. We set U and L in the **SCC**_{sim} algorithm by using the a similarity of 1 as the upper-bound (recall that this is always the maximum similarity in the reweighted graph) and using a lower bound of 0.001.

⁹We also considered other distance-to-similarity schemes, e.g., $\text{sim}(u, v) = 1/(1 + \log^c(\text{dist}(u, v)))$ and $\text{sim}(u, v) = e^{-\text{dist}(u, v)}$. We observed that the first choice, with $c > 1$ yielded slightly better quality results for sparse graph-based methods, but chose the scheme used in this paper for its simplicity.

¹⁰For more detailed information see <https://scikit-learn.org/stable/datasets.html>.

Lastly, we also compare the graph-based implementations of HAC to pointset-based HAC implementations from the `scipy` package using the single-, complete-, average-, and Ward-linkage measures.

13.6.1 Quality

We compare the quality of our clustering results with respect to known ground-truth clusterings for the *iris*, *wine*, *digits*, and *cancer*, and *faces* classification datasets from the UCI dataset repository (found in the `sklearn.datasets` package). We run all of the graph-based clustering algorithms on similarity graphs generated from the input pointsets using $k = 10$ in the approximate k -NN construction. We run the `scipy` pointset clustering algorithms directly on the input pointsets. To measure quality, we use the ***Adjusted Rand-Index (ARI)*** and ***Normalized Mutual Information (NMI)*** scores, which are standard measures of the quality of a clustering with respect to a ground-truth clustering. We also use the ***Dendrogram Purity*** measure [163], which takes on values between $[0, 1]$ and takes on a value of 1 if and only if the tree contains the ground truth clustering as a tree consistent partition (i.e., each class appears as exactly the leaves of some subtree of the tree). Given a tree T tree with leaves V , and a ground truth partition of V into $C = \{C_1, \dots, C_l\}$ classes, define the purity of a subset $S \subseteq V$ with respect to a class C_i to be $\mathcal{P}(S, C_i) = |S \cap C_i|/|S|$. Then, the purity of T is

$$\mathcal{P}(T) = \frac{1}{|Pairs|} \sum_{i=1}^l \sum_{x,y \in C_i, x \neq y} P(\text{lca}_T(x, y), C_i)$$

where $Pairs = \{(x, y) \mid \exists i \text{ s.t. } \{x, y\} \subseteq C_i\}$ and $\text{lca}_T(x, y)$ is the set of leaves of the least common ancestor of x and y in T . Lastly, we also study the unsupervised ***Dasgupta Cost*** [89] measure of our dendograms, which is measured with respect to an underlying similarity graph $G(V, E, w)$ and is defined as:

$$\sum_{(u,v) \in E} |\text{lca}_T(u, v)| \cdot w(u, v)$$

We compute the Dasgupta cost over the complete similarity graph obtained directly by computing all point-to-point distances.

Quality Comparison. Table 13.2 shows the results of our quality study. We observe that our $\text{ParHAC}_{0.1}$ algorithm achieves consistent high-quality results across all of the quality measures that we evaluate. For the ARI measure, $\text{ParHAC}_{0.1}$ is on average within 1.5% of the best ARI score for each graph (and achieves the best score for one of the graphs). For the NMI measure, $\text{ParHAC}_{0.1}$ is on average within 1.3% of the best NMI score for each graph (and again achieves the best score for two of the graphs). $\text{ParHAC}_{0.1}$ also achieves good results for the dendrogram purity and Dasgupta cost measures. For purity, it is on average within 1.9% of the best purity score for each graph, achieving the best score for one of the graphs, and for the unsupervised Dasgupta cost measure it is on average within 1.03% of the smallest

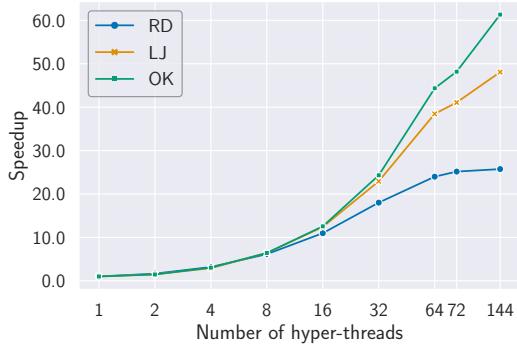


Figure 13.2: Speedups for three of our large real-world graphs on the y -axis versus the number of hyper-threads used on the x -axis.

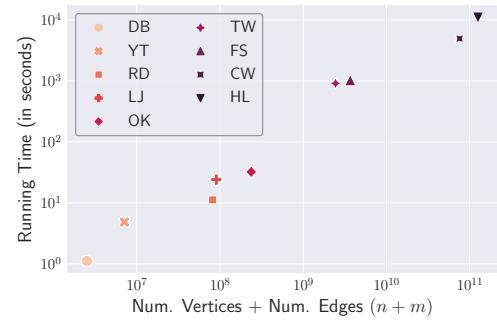


Figure 13.3: Parallel running time of the ParHAC algorithm in seconds on the y -axis in log-scale versus the size of each graph in terms of the total number of vertices and edges on the x -axis.

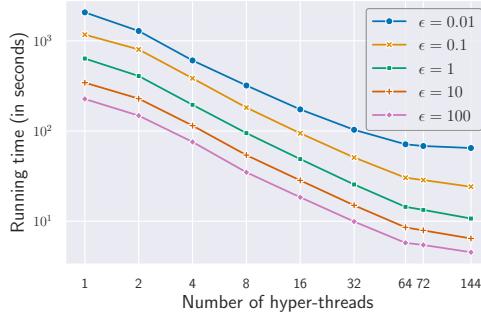


Figure 13.4: Parallel running times of the ParHAC algorithm on the LJ graph in log-scale as a function of the number of threads for varying values of the accuracy parameter, ϵ .

Dasgupta cost score for each graph. Compared with the SciPy unweighted average-linkage which runs on the underlying pointset, ParHAC_{0.1} achieves 14.4% better ARI score on average, 3.6% better NMI score on average, 4.7% better dendrogram purity on average, and 1.02% larger Dasgupta cost on average. Lastly, we observe that out of Affinity and SCC_{sim}, SCC_{sim} nearly always outperforms Affinity on all quality measures. Compared to SCC_{sim}, ParHAC_{0.1} consistently obtains better quality results, achieving 35.6% better ARI score on average, 12.1% better NMI score on average, 6.7% better dendrogram purity on average, and 3.1% better Dasgupta cost on average.

Quality with Varying k . An interesting question is whether the graph-based approach requires a large value of k in graph building to achieve high quality. To study this question, we studied each of the quality measures for the studied algorithms as a function of the k used in the k -NN construction. We present the full results in the full version of the paper, and present the results for the ARI measure on the digits point dataset in Figure 13.5. Our results show that modest values of k yield

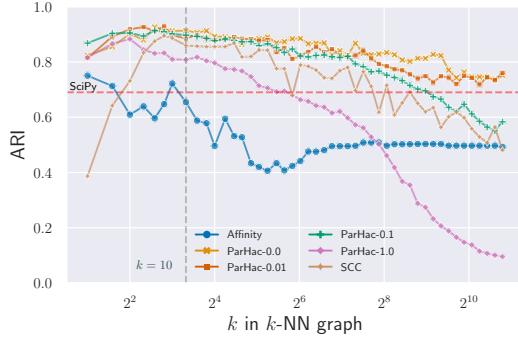


Figure 13.5: ARI score on the *digits* point dataset as a function of the k used in graph building. The gray vertical line highlights the values for $k = 10$, which is the value used across all datasets for the results in Table 13.2. The red horizontal line is the ARI of SciPy’s average-linkage.

very high quality results, and that for example, even tripling, or increasing k an order of magnitude either yields negligible improvement in quality, or actually degrades the quality. For example, for ParHAC with $\epsilon = 0.1$, using $k = 100$ is 10% worse than using $k = 10$, and using $k = 1000$ is 47% worse than $k = 10$. We also observe that even for very small k , e.g., $k = 2$, ParHAC computes very high quality clusterings, which are significantly higher than the corresponding metric-space result for average-linkage. Furthermore, using a smaller value of ϵ when running ParHAC yields results that are consistently high quality, especially compared with Affinity and SCC, which are noisier, likely due to overmerging within the affinity trees computed in each round.

Overall, we find that ParHAC achieves consistently high quality results across the four quality measures that we evaluate. Our results show that being more faithful to the HAC algorithm allows ParHAC to obtain meaningful quality gains over Affinity and SCC_{sim} .

13.6.2 Scalability on Large Real-World Graphs

Next, we study ParHAC’s scalability on large real-world graphs. Table 13.1 shows the statistics of the graphs used in our experiments.

Speedup Results. In Figure 13.2 we show the speedup of our ParHAC implementation on the RD, LJ, and OK graphs. On LJ and OK, our ParHAC implementation achieves 48.0x and 61.3x self-relative speedup respectively. For the RD graph, our ParHAC implementation achieves 25.7x self-relative speedup. The lower self-relative speedups for the RD graph are since ParHAC performs significantly less work on RD than on the LJ graph. In particular, the time spent merging vertices in the Multi-Merge implementation is 3.8x lower than the time spent in the LJ graph, although both graphs have nearly the same total number of vertices and edges. The reason is that the average number of neighbors per cluster at the time of its merge is significantly lower on RD than on the other graphs.

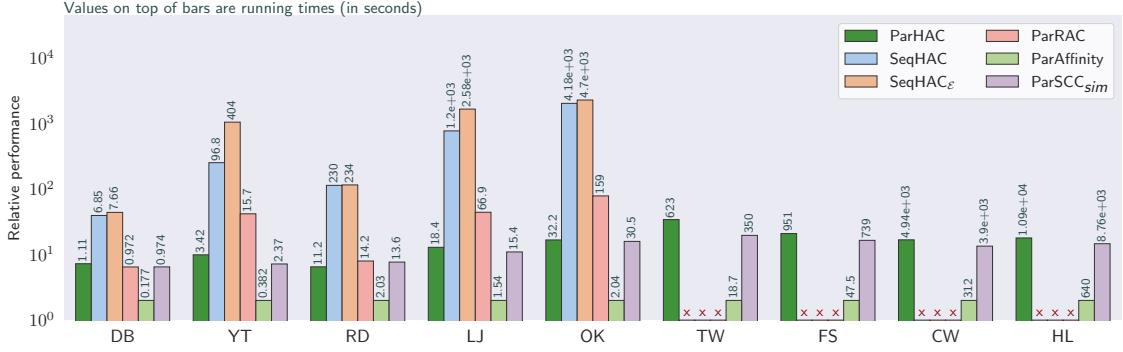


Figure 13.6: Relative performance of our **ParHAC** algorithm compared to the SeqHAC, SeqHAC_E, RAC, Affinity, and SCC_{sim} algorithms. We prefix new implementations developed in this paper with the **Par** prefix. **ParHAC**, SeqHAC, SeqHAC_E, and RAC are HAC-like algorithms that produce full dendograms, whereas Affinity and SCC_{sim} are MST-based methods that produce a (typically small) set of flat clusterings. The values on top of each bar show the running time of each algorithm in seconds. The times shown for the parallel algorithms are 144 hyper-thread times. We run the **ParHAC** and **SeqHAC** algorithm using $\epsilon = 0.1$. We terminated methods that ran for more than 6 hours and mark them with a red x.

Scalability with Increasing Graph Sizes. Figure 13.3 shows the parallel running times of **ParHAC** as a function of the graph size in terms of the total number of vertices and edges in the graph. We observe that the running time of our algorithms grows essentially linearly as a function of the graph size. We noticed that although the total number of vertices and edges for the RD and LJ graphs are similar (LJ has 10% more total vertices and edges), the running time for the LJ graph is 33% larger. The reason is that the RD graph has significantly lower average-degree than the LJ graph, which results in much less work being performed on average when merging vertices.

Scalability for Varying Epsilon. In the next experiment, we study the absolute performance and speedup achieved by our **ParHAC** implementation on a fixed graph, as ϵ is varied. Figure 13.4 shows the results on the LJ graph for $\epsilon \in \{0.01, 0.1, 1, 10, 100\}$. We observe that larger ϵ consistently results in lower running times, and that the value of $\epsilon = 0.1$ which we use in our quality and other scalability experiments requires the second highest running times. We observed similar results on our other graph inputs. On 144 hyper-threads using a value of $\epsilon = 1$ provides a 2.25x speedup, and $\epsilon = 100$ yields a 5.32x speedup over the parallel running time of $\epsilon = 0.1$.

Comparison with Other Algorithms. Figure 13.6 shows the relative performance and parallel running times of **ParHAC** compared with the SeqHAC and SeqHAC_E algorithms, and our implementations of the RAC, Affinity, and SCC_{sim} algorithms. We prefix the names of new implementations (where no shared-memory parallel implementation was previously available) with **Par**.

We observe that our Affinity implementation is always the fastest on our graph inputs. The reason is that on average Affinity requires just 8.7 iterations to com-

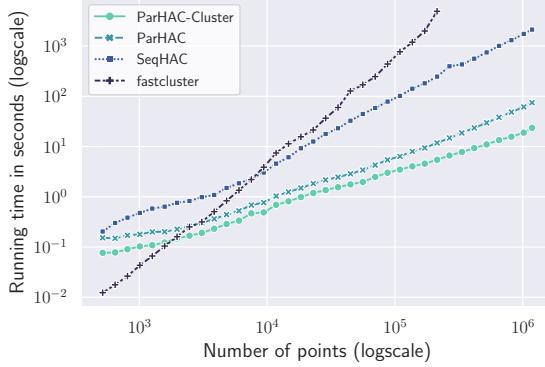


Figure 13.7: End-to-end running times of fastcluster’s unweighted average-linkage, SeqHAC using $\epsilon = 0.1$, and ParHAC using $\epsilon = 0.1$ and 144 hyper-threads on varying-size slices of the glove-100 dataset. The running times shown for SeqHAC and ParHAC include the cost of computing approximate k -NN and generating the input similarity graph. ParHAC-Cluster reports just the time taken to cluster the generated similarity graph. We terminated implementations that run for longer than 3 hours.

plete on these inputs. On the other hand, SCC_{sim} , which runs Affinity with weight thresholding, uses all 100 iterations, and is an average of 11.5x slower than Affinity due to the cost of the additional iterations. Our results show that when carefully implemented, Affinity is a highly scalable algorithms that can cluster graphs with billions of edges in a matter of just tens of minutes, although unlike ParHAC and the other approximate and exact HAC algorithms, they do not provide strong theoretical guarantees.

Compared to Affinity and SCC_{sim} , ParHAC is an average of 14.8x slower than our Affinity implementation and 1.24x slower than our SCC_{sim} implementation. The main reason for the relative speed of ParHAC compared with Affinity is the much larger number of rounds required by ParHAC on our graph inputs (see Figure 13.1). We note that running SCC_{sim} with fewer rounds yields faster results, but used 100 iterations since this setting yielded the highest quality results in our evaluation in Section 13.6.1. Furthermore, for all of our graph inputs, Affinity has one (or a few) rounds where the graph shrinks by a massive amount, suggesting the formation of giant cluster(s) through chaining, a known issue with clustering methods based on Boruvka’s algorithm [227]. For example, the first round of Affinity on the CW graph drops the number of edges from 74.7B to 5.31B (14x lower) and the number of vertices from 955M to 118M. Importantly, the very first round forms a cluster containing 261M vertices. To conclude, our results show that ParHAC achieves a good compromise between running time and quality, as our study in Section 13.6.1 shows that both Affinity and SCC_{sim} can produce sub-optimal dendograms compared to the greedy exact HAC baseline.

Compared with other methods that provide provable guarantees compared with HAC, ParHAC is significantly faster. Compared with SeqHAC and SeqHAC $_{\mathcal{E}}$, ParHAC obtains 45.2x and 72.7x speedup on average respectively. Compared with RAC,

ParHAC achieves an average speedup of 7.1x. Although **RAC** is faster than **ParHAC** on **DB** and **RD**, two of our small graph inputs, it seems to require a very large number of rounds on the remaining graphs. Since each round computes the best edge for each vertex for a total of approximately $O(m)$ work, and the number of rounds for our larger graphs is close to $O(\sqrt{m})$ based on Figure 13.1, the super-linear total work of this algorithm prevents it from achieving good scalability. Similarly, we note that **ParHAC_E** (i.e. using $\epsilon = 0$) only runs within the time limit for **DB**, **YT**, and **RD**, and therefore it is not shown in Figure 13.6. The reason is that it performs $O(mn)$ work when $\epsilon = 0$, since each iteration costs $O(m)$ work and the number of outer-rounds is $O(n)$.

Discussion. To the best of our knowledge, our results are the first to show that graphs with tens to hundreds of billions of edges can be clustered in a matter of tens of minutes (using heuristic methods like **Affinity** and **SCC_{sim}** with fewer iterations) to hours (using **SCC_{sim}** with many iterations or methods with approximation guarantees such as **ParHAC**). We are not aware of other shared-memory clustering results that work at this large scale. Our theoretically-efficient implementations can be viewed as part of a recent line of work showing that theoretically-efficient shared-memory parallel graph algorithms can scale to the largest publicly available graphs using a modest amount of resources [95, 98, 99].

13.6.3 Pointset Clustering: End-to-End Evaluation

We conclude by evaluating the performance of **ParHAC** when clustering larger pointsets. Here we are interested in understanding whether our method yields speedups compared to the fastest unweighted average-linkage HAC implementations for pointsets when we take into account both the similarity graph construction time and the clustering time for our algorithms. We chose to use the unweighted average-linkage implementation from `fastcluster` [241], which we found yields more consistent (and slightly faster) performance for larger numbers of points than the implementations in `sklearn` and `SciPy`. We also compare **ParHAC** to the **SeqHAC** algorithm in this setting, where **SeqHAC** uses the same graph construction method as **ParHAC**, and both algorithms are run using $\epsilon = 0.1$.

We run our end-to-end experiment on the **Glove-100** dataset, which is a 100-dimensional dataset containing vector-embeddings for 1.18 million words. Figure 13.7 shows the results of our experiment, and separates the time spent clustering the graph from the end-to-end time of **ParHAC**. First, we found that for $n > 3000$, the end-to-end times of **ParHAC** are always faster than the time taken by `fastcluster`, and for $n > 9400$, the end-to-end times of **SeqHAC** are always faster than the time taken by `fastcluster`. Comparing `fastcluster` on the largest dataset it can solve in under three hours with the graph-based methods, **ParHAC** is 417x faster and **SeqHAC** is 20x faster.

13.7 Discussion

In this chapter, we have introduced **ParHAC**, the first parallel algorithm for hierarchical agglomerative graph clustering using the average-linkage measure that has strong theoretical-bounds on its work and span, as well as provable approximation guarantees. We have shown that **ParHAC** scales to massive real-world graphs with tens to hundreds of billions of edges on a single machine and achieves high-quality results compared to existing graph-based and pointset-based hierarchical clustering methods.

An interesting question for future work is whether we can reduce the round complexity of our algorithm to make it more suitable for distributed settings. Another question is to study (approximate) graph-based HAC in the dynamic or incremental settings. In particular, we are excited to see if the ideas used to design our parallel approximation algorithm can be extended to design a practical and theoretically-efficient $(1 + \epsilon)$ -approximate dynamic HAC algorithm.

Part IV

Conclusion and Future Directions

Chapter 14

Conclusion and Future Work

14.1 Conclusion

Graph clustering has been an influential area of study in the past decade due to the influx of large-scale data and the demonstrable power of processing such data to extract meaningful information for real-world applications. This thesis addresses a broad class of problems relating to graph clustering, and provides both theoretically and practically efficient solutions that scale to graphs with hundreds of billions of edges and that can operate on publicly accessible commodity multi-core machines. The results in this thesis represent a balance between cost-effectiveness, performance, and scalability, and are achieved using a multi-faceted approach, combining a strong theoretical grounding with performance engineering techniques to overcome the computational challenges of large-scale graph processing. The resulting algorithms and implementations are highly applicable in real-world systems, and are used in production environments in industry to solve real-world problems. This thesis demonstrates the importance of taking an interdisciplinary approach to work-intensive data problems, as theoretical concepts inform practical design and vice versa. Indeed, for some of the graph processing problems in this thesis, our work is the first to scale to the largest publicly available graphs with hundreds of billions of edges.

Part I of this thesis studied efficient parallel algorithms for subgraph counting and listing problems, and introduced new theoretically efficient algorithms and implementations for butterfly (four-cycle), five-cycle, and k -clique counting. We developed exact and approximate counting algorithms, and explored k -clique counting in both static and batch-dynamic settings. The algorithms that we introduced are work-efficient with polylogarithmic span, and the implementations achieved fast running times on large-scale graphs. As far as we know, this work is the first to report four-clique counts on the largest publicly available graph with hundreds of billions of edges. Notably, this thesis also introduced new algorithms for low out-degree orientations, which were vital to reducing the work necessary in our subgraph counting algorithms, and which have broader applicability to classic graph processing algorithms. To the best of our knowledge, in the batch-dynamic setting, this work is the first to compute an approximate k -core decomposition in polylogarithmic depth.

Part II of this thesis introduced parallel algorithms and implementations for a class of computationally intensive subgraph decomposition problems, which discover and hierarchically classify dense substructures within a graph. This thesis developed work-efficient parallel algorithms for bi-core decomposition, butterfly peeling, k -clique peeling, and nucleus decomposition, and proved that many of these problems are P-complete, suggesting that polylogarithmic span solutions are unlikely. This part additionally introduced approximation algorithms for k -clique peeling and nucleus decomposition, and demonstrated orders of magnitude speedups compared to the prior state-of-the-art implementations for subgraph decomposition algorithms.

Finally, Part III of this thesis focused on highly scalable graph clustering algorithms that exhibit high quality in practice compared to ground-truth data. This thesis introduced new parallel algorithms for correlation clustering and hierarchical agglomerative clustering, and developed a polylogarithmic span algorithm for approximate hierarchical agglomerative clustering. We demonstrated that our correlation clustering implementation scales to graphs with billions of edges while maintaining a good precision-recall tradeoff curve compared to ground-truth data, and our hierarchical agglomerative clustering algorithm achieves significant speedups over the state-of-the-art while maintaining roughly the same quality.

Throughout this thesis, by combining deep theoretical understandings with common properties and access behaviors of real-world datasets, we have shown that developing shared-memory parallel clustering algorithms with strong theoretical guarantees and using performance engineering techniques can lead to highly scalable, efficient, and cost-effective implementations on real-world datasets. The work in this thesis addressed core problems in academia and industry, and bridged the gap to bring theory to practice.

14.2 Future Work

There are many interesting directions for future work, including expanding the techniques and methodologies used in this thesis to other problem domains, such as generalizations of classic graph problems to hypergraphs, and computationally intensive problems in metric spaces. It would be interesting to explore other flavors of parallelism across different kinds of hardware, including distributed networks and GPUs, as well as explore different parallel behaviors in nonvolatile memory technologies, such as phase-change memory with asymmetric reads and writes. Some concrete examples of areas for future work include the following.

Practical frameworks. A main focus of this thesis is ensuring that the theoretical algorithms and implementations that we study are grounded in practical applications, and take advantage of the structures of the data that appears in practice. A significant barrier to this is the disconnect between production systems and research code, which makes it arduous to adopt current research in solutions for current problems and makes it difficult to understand which research directions are directly applicable to real-world problems. An ongoing effort and an area with opportunities for further expansion is in developing a shared-memory parallel graph and metric clustering

framework that serves as a bridge between the research at MIT and the work and real-world problems at Google. The concept would be to build a framework on top of our Graph Based Benchmark Suite (GBBS) [106], with an easy-to-use API that seamlessly integrates with the current API used in production at Google. Broadly speaking, it is of both academic and industrial interest to integrate open-source systems with those used in production environments, to make academic code more robust and to introduce cutting-edge research to current applications. The goal for a clustering framework would be to focus on scalable clustering implementations with different theoretical guarantees, and elucidate how these implementations compare to one another and in which settings each implementation is preferable, with a parallel evaluation pipeline that collects metrics and comparisons to ground-truth data. In the future, we plan to develop scalable and modular frameworks for other graph and data mining problems, such as subgraph counting and mining, with integration into production environments, and with a focus on real-world data inputs and evaluation metrics.

Complementarily, another interesting direction for future work is to bridge the gap between spatial data processing and graph processing. In particular, mapping spatial data to sparse graphs using classic graph building techniques allows us to apply efficient sparse graph algorithms that can translate meaningfully to results on spatial data. For applications such as clustering, handling spatial data in this manner can even improve metrics such as clustering quality, due to the noise reduction given by classic k -nearest neighbor graph building techniques, as based on our initial findings in Chapter 13. An important goal would be to develop a unified approach to spatial data and graph processing, and explore the interplay between algorithms and techniques used in each setting.

Dynamic and streaming settings. Additionally, today’s graphs are constantly evolving and changing in real-time, requiring fast computation to satisfy incoming queries with low latency. As a result, an interesting avenue for future work would be to develop efficient graph processing algorithms for dynamic and streaming settings, and an important future step is to explore both new theoretical guarantees and practical implementations of computationally intensive graph algorithms in these settings. Notably, even single edge insertions in graphs can cause cascading effects in graph metrics and even dynamic graph storage, and additional sequential dependencies can limit the parallelism available in processing such changes, as demonstrated in Chapter 6; strong theoretical guarantees limit the worst-case behaviors of such modifications, and coupled with practical optimizations such as for good cache locality, we can develop scalable and efficient implementations that significantly outperform static or sequential counterparts.

Generalizations of graphs. Furthermore, the classic conception of graphs is limited in terms of the kinds of relational data that they can encode and represent. One facet is that classic graphs only represent pairwise relationships, whereas data today may have more complicated multi-way relationships. More powerful representations include generalizations of graphs such as hypergraphs, which have widespread applications in areas including document recommendation and social network analysis,

but for the most part classic graph algorithms and implementations do not trivially extend to efficient hypergraph algorithms and implementations. Moreover, static and evolving graph data may be augmented with metadata or other properties such as probabilistic decay functions on edges, which are normally not taken into account in classic graph algorithms. An interesting direction for future work would be to challenge the current limitations of graph algorithms and frameworks, and explore ways to efficiently generalize or optimize these algorithms to broader classes of data.

Other parallel hardware. Finally, an important next step would be to develop algorithms with considerations for emerging hardware innovations, to fully take advantage of newer machines and behaviors. In particular, graph algorithms often display poor spatial locality due to arbitrary vertex and edge accesses, which makes cache behavior essential in terms of practical efficiency. Modern multiprocessor architectures take into account non-uniform memory accesses (NUMA), but modifying current algorithms to exploit locality in NUMA-aware systems is non-trivial. Additionally, there are technologies in non-volatile memory, with different properties compared to the classic DRAM in terms of asymmetric read and write operations in terms of energy and latency, motivating the development of algorithms designed specifically for these asymmetries. An interesting direction would be to develop graph algorithms and implementations in the future optimizing for performance and memory accesses on these state-of-the-art systems and architectures, and more broadly in generalizing current graph frameworks to other kinds of parallel hardware.

Bibliography

- [1] Amazon EC2 instance types. <https://aws.amazon.com/ec2/instance-types/>. Accessed: 2022-03-28.
- [2] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 434–443, 2014.
- [3] Amir Abboud, Vincent Cohen-Addad, and Hussein Houdrouge. Subquadratic high-dimensional hierarchical clustering. In *Advances in Neural Information Processing Systems*, pages 11580–11590, 2019.
- [4] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.
- [5] Charu C. Aggarwal and Haixun Wang. *A Survey of Clustering Algorithms for Graph Data*, pages 275–301. Springer US, Boston, MA, 2010.
- [6] Adel Ahmed, Vladimir Batagelj, Xiaoyan Fu, Seok-hee Hong, Damian Merrick, and Andrej Mrvar. Visualisation and analysis of the internet movie database. In *International Asia-Pacific Symposium on Visualization*, pages 17–24, 2007.
- [7] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, Nick G. Duffield, and Theodore L. Willke. Graphlet decomposition: Framework, algorithms, and applications. *Knowl. Inf. Syst.*, 50(3):689–722, 2017.
- [8] Nesreen K. Ahmed, Theodore L. Willke, and Ryan A. Rossi. Estimation of local subgraph counts. In *IEEE International Conference on Big Data (BigData)*, pages 586–595, 2016.
- [9] Nesreen K. Ahmed, Theodore L. Willke, and Ryan A. Rossi. Exact and estimation of local edge-centric graphlet counts. In *International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 1–17, 2016.
- [10] Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: Ranking and clustering. *J. ACM*, 55(5), November 2008.

- [11] Esra Akbas and Peixiang Zhao. Truss-based community search: A truss-equivalence based indexing approach. *Proc. VLDB Endow.*, 10(11):1298–1309, August 2017.
- [12] Sinan G. Aksoy, Tamara G. Kolda, and Ali Pinar. Measuring and modeling bipartite graphs with community structure. *J. Complex Networks*, 5:581–603, 2017.
- [13] Mohammad Almasri, Omer Anjum, Carl Pearson, Zaid Qureshi, Vikram S. Mailthody, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Update on k-truss decomposition on GPU. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019.
- [14] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.
- [15] J. I. Alvarez-Hamelin, Luca Dall’asta, Alain Barrat, and Alessandro Vespignani. Large scale networks fingerprinting and visualization using the k -core decomposition. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 41–50. 2005.
- [16] Altaf Amin, Yoko Shinbo, Kenji Mihara, Ken Kurokawa, and Shigehiko Kanaya. Development and implementation of an algorithm for detection of protein complexes in large interaction networks. *BMC bioinformatics*, 7:207, 02 2006.
- [17] Richard Anderson and Ernst W. Mayr. A P-complete problem and approximations to it. Technical report, 1984.
- [18] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. Distributed k -core decomposition and maintenance in large dynamic graphs. In *ACM International Conference on Distributed and Event-Based Systems*, pages 161–168, 2016.
- [19] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. PATRIC: A parallel algorithm for counting triangles in massive networks. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 529–538, 2013.
- [20] Nima S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2):115–144, 2001.
- [21] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *International Conference on Similarity Search and Applications*, pages 34–49. Springer, 2017.
- [22] H. Avron. Counting triangles in large graphs using randomized matrix trace estimation. In *Workshop on Large-scale Data Mining: Theory and Applications*, 2010.

- [23] A. Azad, A. Buluç, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, 2015.
- [24] Gary D. Bader and Christopher WV Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4(1):2, January 2003.
- [25] Seung-Hee Bae, Daniel Halperin, Jevin D. West, Martin Rosvall, and Bill Howe. Scalable and efficient flow-based community detection for large-scale graph analysis. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(3), March 2017.
- [26] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proc. VLDB Endow.*, 5(5):454–465, January 2012.
- [27] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Mach. Learn.*, 56(1–3):89–113, June 2004.
- [28] Luis Barba, Jean Cardinal, Matias Korman, Stefan Langerman, André van Renssen, Marcel Roeloffzen, and Sander Verdonschot. Dynamic graph coloring. *Algorithmica*, 81(4):1319–1341, 2019.
- [29] Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. *Distributed Comput.*, 22(5–6):363–379, 2010.
- [30] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. Affinity clustering: Hierarchical clustering at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6864–6874, 2017.
- [31] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 16–24, 2008.
- [32] Austin R. Benson, David F. Gleich, and Jure Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.
- [33] Jean-Paul Benzécri. Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques. *Cahiers de l'analyse des données*, 7(2):209–218, 1982.
- [34] Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. Linear Time Subgraph Counting, Graph Degeneracy, and the Chasm at Size Six. In *Proceedings of the Innovations in Theoretical Computer Science Conference*, volume 151, pages 38:1–38:20, 2020.

- [35] Bonnie Berger, John Rompel, and Peter W. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. *J. Computer and System Sciences*, 49(3):454–477, 1994.
- [36] Edvin Berglin and Gerth Stølting Brodal. A simple greedy algorithm for dynamic graph orientation. *Algorithmica*, 82(2):245–259, feb 2020.
- [37] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *International Colloquium on Automata, Languages, and Programming*, volume 9134, pages 167–179, 2015.
- [38] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 692–711, 2016.
- [39] Jonathan W. Berry, Luke K. Fostvedt, Daniel J. Nordman, Cynthia A. Phillips, C. Seshadhri, and Alyson G. Wilson. Why do simple algorithms for triangle enumeration work in the real world? In *Innovations in Theoretical Computer Science (ITCS)*, pages 225–234, 2014.
- [40] Maciej Besta, Armon Carigiet, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, and Torsten Hoefer. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.
- [41] Alex Beutel, WanHong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. Copycatch: Stopping group attacks by spotting lockstep behavior in social networks. In *International World Wide Web Conference (WWW)*, page 119–130, 2013.
- [42] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *ACM Symposium on Theory of Computing (STOC)*, pages 173–182, 2015.
- [43] Divya Bhattachari. Towards scalable parallel Fibonacci heap implementation. Master’s thesis, St. Cloud State University, 2018.
- [44] M. A. Bhuiyan, M. Rahman, M. Rahman, and M. Al Hasan. GUISE: Uniform sampling of graphlets for large graph analysis. In *IEEE International Conference on Data Mining (ICDM)*, pages 91–100, 2012.
- [45] Mark Blanco, Tze Meng Low, and Kyungjoo Kim. Exploration of fine-grained parallelism for load balancing eager k -truss on GPU and CPU. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019.

- [46] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib - a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 507—509, 2020.
- [47] Guy E. Blelloch and Laxman Dhulipala. Introduction to parallel algorithms. <http://www.cs.cmu.edu/~realworld/slidesS18/parallelChap.pdf>, 2018. Carnegie Mellon University.
- [48] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 189–199, 2010.
- [49] Guy E. Blelloch and Bruce M. Maggs. Parallel algorithms. *Communications of the ACM*, 39:85–97, 1996.
- [50] Guy E. Blelloch, Richard Peng, and Kanat Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 23–32, 2011.
- [51] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *J. Stat. Mech. Theory Exp.*, 2008(10):P10008, October 2008.
- [52] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1), 1998.
- [53] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.
- [54] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. In *International World Wide Web Conference (WWW)*, pages 595–602, 2004.
- [55] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. Core decomposition of uncertain graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1316–1325, 2014.
- [56] Stephen P. Borgatti and Martin G. Everett. Network analysis of 2-mode data. *Social Networks*, 19(3):243 – 269, 1997.
- [57] Otakar Boruvka. O jistém problému minimálním. *Práce Mor. Přírodověd. Spol. v Brně III*, 3:37–58, 1926.
- [58] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

- [59] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. Motif counting beyond five nodes. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 12(4):48:1–48:25, 2018.
- [60] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. Motivo: Fast motif counting via succinct color coding and adaptive sampling. *Proc. VLDB Endow.*, 12(11):1651–1663, July 2019.
- [61] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representations of sparse graphs. In *Proc. 6th International Workshop on Algorithms and Data Structures (WADS)*, pages 342–351, 1999.
- [62] Ronald S. Burt. Structural holes and good ideas. *American Journal of Sociology*, 110(2):349–399, 2004.
- [63] Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt, and Eran Shir. A model of Internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences*, 104(27):11150–11154, 2007.
- [64] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining (SDM)*, pages 442–446, 2004.
- [65] T.-H. Hubert Chan, Mauro Sozio, and Bintao Sun. Distributed approximate k-core decomposition and min-max edge orientation: Breaking the diameter barrier. *J. Parallel Distributed Comput.*, 147:87–99, 2021.
- [66] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In *International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*, pages 84–95, 2000.
- [67] Moses Charikar and Vaggos Chatziafratis. Approximate hierarchical clustering via sparsest cut and spreading metrics. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–854, USA, 2017.
- [68] Moses Charikar, Vaggos Chatziafratis, and Rad Niazadeh. Hierarchical clustering better than average-linkage. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2291–2304, 2019.
- [69] Evangelos Chatziafratis. *Hierarchical clustering with global objectives: Approximation algorithms and hardness results*. Stanford University, 2020.
- [70] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. Accelerating truss decomposition on heterogeneous processors. *Proc. VLDB Endow.*, 13(10):1751–1764, June 2020.
- [71] Pei-Ling Chen, Chung-Kuang Chou, and Ming-Syan Chen. Distributed algorithms for k-truss decomposition. In *IEEE International Conference on Big Data (BigData)*, pages 471–480, 2014.

- [72] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, February 1985.
- [73] Flavio Chierichetti, Nilesh Dalvi, and Ravi Kumar. Correlation clustering in MapReduce. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, page 641–650. Association for Computing Machinery, 2014.
- [74] M. Chrobak and D. Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theor. Comput. Sci.*, 86:243–266, 1991.
- [75] Deming Chu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang, Yinglong Xia, and Chenyi Zhang. Finding the best k in core decomposition: A time and space optimal solution. In *IEEE International Conference on Data Engineering (ICDE)*, pages 685–696, 2020.
- [76] Deming Chu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. Hierarchical core decomposition in parallel: From construction to subgraph search. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1138–1151, 2022.
- [77] Martino Ciaperoni, Edoardo Galimberti, Francesco Bonchi, Ciro Cattuto, Francesco Gullo, and Alain Barrat. Relevance of temporal cores for epidemic spread in temporal networks. *Scientific Reports*, 10(1), July 2020.
- [78] Michael Cochez and Hao Mou. Twister tries: Approximate hierarchical agglomerative clustering for average distance in linear time. In *ACM SIGMOD International Conference on Management of Data*, pages 505–517, 2015.
- [79] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 16(3.1), 2008.
- [80] Jonathan Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Eng.*, 11(4):29–41, July 2009.
- [81] Vincent Cohen-Addad, Varun Kanade, and Frederik Mallmann-Trenn. Hierarchical clustering beyond the worst-case. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, pages 6201–6209. Curran Associates, Inc., 2017.
- [82] Vincent Cohen-Addad, Varun Kanade, Frederik Mallmann-trenn, and Claire Mathieu. Hierarchical clustering: Objective functions and algorithms. *J. ACM*, 66(4), 2019.
- [83] Alessio Conte, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Verzani. Discovering k -trusses in large-scale networks. In *IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018.
- [84] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

- [85] Bruce Croft and Jamie Callan. The Lemur project. <https://www.lemurproject.org/>, 2016.
- [86] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k -cliques in sparse real-world graphs. In *The Web Conference (WWW)*, page 589–598, 2018.
- [87] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: Scalable online collaborative filtering. In *International World Wide Web Conference (WWW)*, page 271–280. Association for Computing Machinery, 2007.
- [88] N. S. Dasari, R. Desh, and M. Zubair. ParK: An efficient algorithm for k -core decomposition on multicore processors. In *IEEE International Conference on Big Data (BigData)*, pages 9–16, 2014.
- [89] Sanjoy Dasgupta. A cost function for similarity-based hierarchical clustering. In *ACM Symposium on Theory of Computing (STOC)*, page 118–127, New York, NY, USA, 2016. Association for Computing Machinery.
- [90] Manoranjan Dash, Simona Petrucci, and Peter Scheuermann. ppop: Fast yet accurate parallel hierarchical clustering using partitioning. *Data Knowl. Eng.*, 61(3):563–578, 2007.
- [91] V. S. Dave, N. K. Ahmed, and M. Hasan. PE-CLoG: Counting edge-centric local graphlets. In *IEEE International Conference on Big Data (BigData)*, pages 586–595, 2017.
- [92] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48, 2013.
- [93] Camil Demetrescu, Andrew Goldberg, and David Johnson. 9th DIMACS implementation challenge: Shortest paths. <http://www.dis.uniroma1.it/challenge9/>, 2019.
- [94] Inderjit S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 269–274, 2001.
- [95] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.
- [96] Laxman Dhulipala, Guy E Blelloch, Yan Gu, and Yihan Sun. Pac-trees: Supporting parallel and compressed purely-functional collections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2022.

- [97] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. <https://github.com/paralg/gbbs>.
- [98] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2018.
- [99] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 918–934, 2019.
- [100] Laxman Dhulipala, David Eisenstat, Jakub Łącki, Vahab Mirrokni, and Jessica Shi. Hierarchical agglomerative graph clustering in nearly-linear time. In *International Conference on Machine Learning (ICML)*, volume 139, pages 2676–2686. PMLR, 2021.
- [101] Laxman Dhulipala, David Eisenstat, Jakub Łącki, Vahab Mirrokni, and Jessica Shi. Hierarchical agglomerative graph clustering in poly-logarithmic depth. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 22925–22940. Curran Associates, Inc., 2022.
- [102] Laxman Dhulipala, Changwan Hong, and Julian Shun. ConnectIt: A framework for static and incremental parallel graph connectivity algorithms. *Proc. VLDB Endow.*, 14(4):653–667, dec 2020.
- [103] Laxman Dhulipala, Quanquan Liu, Sofya Raskhodnikova, Jessica Shi, Julian Shun, and Shangdi Yu. Differential privacy from locally adjustable graph algorithms: k-core decomposition, low outdegree ordering, and densest subgraphs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2022.
- [104] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k -clique counting. In *2nd Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 129–143, 2021.
- [105] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy Blelloch, Phillip Gibbons, and Julian Shun. Sage: Parallel semi-asymmetric graph algorithms for nvrams. *Proc. VLDB Endow.*, 2020.
- [106] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. The graph based benchmark suite (GBBS). In *Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 11:1–11:8. ACM, 2020.
- [107] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. Efficient fault-tolerant group recommendation using alpha-beta-core. In *ACM on Conference on Information and Knowledge Management*, page 2047–2050, 2017.

- [108] T. N. Dinh, X. Li, and M. T. Thai. Network clustering via maximizing modularity: Approximation algorithms and theoretical limits. In *IEEE International Conference on Data Mining (ICDM)*, pages 101–110, 2015.
- [109] Magdalen Dobson, Zheqi Shen, Guy E Blelloch, Laxman Dhulipala, Yan Gu, and Yihan Sun. Working paper: ParANN: Fast batch-incremental high-dimensional graph-based approximate nearest neighbors. 2022.
- [110] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. Extraction and classification of dense implicit communities in the web graph. *ACM Trans. Web*, 3(2), April 2009.
- [111] Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.
- [112] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11):1343–1354, November 1988.
- [113] Xiaoxi Du, Ruoming Jin, Liang Ding, Victor E. Lee, and John H. Thornton. Migration motif: A spatial-temporal pattern mining approach for financial markets. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, page 1135–1144. Association for Computing Machinery, 2009.
- [114] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. Accessed on 07.27.2021.
- [115] Zdenek Dvorák and Vojtech Tuma. A dynamic data structure for counting subgraphs in sparse graphs. In *International Workshop on Algorithms and Data Structures (WADS)*, pages 304–315, 2013.
- [116] Talya Eden, Dana Ron, and C. Seshadhri. Faster sublinear approximation of the number of k-cliques in low-arboricity graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 1467–1478, 2020.
- [117] Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, October 2004.
- [118] Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. Beyond triangles: A distributed framework for estimating 3-profiles of large graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 229–238, 2015.
- [119] Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. Distributed estimation of graph 4-profiles. In *International World Wide Web Conference (WWW)*, pages 483–493, 2016.

- [120] Micha Elsner and Warren Schudy. Bounding and comparing methods for correlation clustering beyond ilp. In *Workshop on Integer Linear Programming for Natural Language Processing*, pages 19–27, 2009.
- [121] 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*, pages 403–414. Springer, 2010.
- [122] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Efficient computation of probabilistic core decomposition at web-scale. In *International Conference on Extending Database Technology*, pages 325–336, 2019.
- [123] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. Nucleus decomposition in probabilistic graphs: Hardness and algorithms. In *IEEE International Conference on Data Engineering (ICDE)*, pages 218–231, 2022.
- [124] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. Parallel and streaming algorithms for k -core decomposition. In *International Conference on Machine Learning (ICML)*, pages 1397–1406, 2018.
- [125] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. Effective community search over large spatial graphs. *Proc. VLDB Endow.*, 10(6):709–720, February 2017.
- [126] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. A survey of community search over big graphs. *Proc. VLDB Endow.*, 29(1):353–392, jan 2020.
- [127] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V. S. Lakshmanan, and Xuemin Lin. Efficient algorithms for densest subgraph discovery. *Proc. VLDB Endow.*, 12(11):1719–1732, July 2019.
- [128] Martin Farach-Colton and Meng-Tsung Tsai. Computing the degeneracy of large graphs. In *Latin American Symposium on Theoretical Informatics*, pages 250–260, 2014.
- [129] Katherine Faust. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks*, 32(3):221–233, 2010.
- [130] Mahmood Fazlali, Ehsan Moradi, and Hadi Tabatabaei Malazi. Adaptive parallel louvain community detection on a multicore platform. *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)*, 54:26 – 34, 2017.
- [131] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261 – 272, 1995.
- [132] Alexandre Fender, Nahid Emad, Serge Petiton, and Maxim Naumov. Parallel modularity clustering. *Procedia Computer Science*, 108:1793–1802, 2017.

- [133] Irene Finocchi, Marco Finocchi, and Emanuele G. Fusco. Clique counting in mapreduce: Algorithms and experiments. *J. Exp. Algorithmics*, 20, October 2015.
- [134] Valeria Fionda, Luigi Palopoli, Simona Panni, and Simona E. Rombo. Bi-grappin: bipartite graph based protein-protein interaction network similarity search. In *IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 355–361, 2007.
- [135] P. Franti, O. Virmajoki, and V. Hautamaki. Fast agglomerative clustering using a k-nearest neighbor graph. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1875–1881, 2006.
- [136] Eugene Fratkin, Brian T. Naughton, Douglas L. Brutlag, and Serafim Batzoglou. MotifCut: Regulatory motifs finding with maximum density subgraphs. *Bioinformatics*, 22(14):e150–e157, 2006.
- [137] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.
- [138] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic shortest paths in digraphs with arbitrary arc weights. *J. Algorithms*, 49(1):86–113, 2003.
- [139] Kasimir Gabert, Ali Pinar, and Ümit V. Çatalyürek. Shared-memory scalable k-core maintenance on dynamic graphs and hypergraphs. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 998–1007, 2021.
- [140] Edoardo Galimberti, Francesco Bonchi, Francesco Gullo, and Tommaso Lanckiano. Core decomposition in multilayer networks: Theory, algorithms, and applications. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 14(1), January 2020.
- [141] David García-Soriano, Konstantin Kutzkov, Francesco Bonchi, and Charalampos Tsourakakis. Query-efficient correlation clustering. In *The Web Conference (WWW)*, page 1468–1478, New York, NY, USA, 2020. Association for Computing Machinery.
- [142] Ullas Gargi, Wenjun Lu, Vahab Mirrokni, and Sangho Yoon. Large-scale community detection on youtube for topic discovery and exploration. In *AAAI Conference on Weblogs and Social Media (ICWSM)*, volume 5, pages 486–489, 2011.
- [143] Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM Journal on Computing*, 20(6):1046–1067, 1991.

- [144] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrović. Improved parallel algorithms for density-based network clustering. In *International Conference on Machine Learning (ICML)*, pages 2201–2210, 2019.
- [145] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarrià-Miranda, A. Khan, and A. Gebremedhin. Distributed Louvain algorithm for graph community detection. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895, 2018.
- [146] Christos Giatsidis, Fragkiskos Malliaros, Dimitrios Thilikos, and Michalis Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. *AAAI Conference on Artificial Intelligence*, 1, 07 2014.
- [147] David Gibson, Ravi Kumar, and Andrew Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. VLDB Endow.*, pages 721–732. VLDB Endowment, 2005.
- [148] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991.
- [149] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences (PNAS)*, 99(12):7821–7826, 2002.
- [150] Michael T. Goodrich and Paweł Pszona. External-memory network analysis algorithms for naturally sparse graphs. In *European Symposium on Algorithms (ESA)*, page 664–676, 2011.
- [151] Oded Green, Luis M. Munguia, and David A. Bader. Load balanced clustering coefficients. In *Workshop on Parallel Programming for Analytics Applications*, pages 3–10, 2014.
- [152] Oded Green, Pavan Yalamanchili, and Luis M. Munguia. Fast triangle counting on the GPU. In *Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2015.
- [153] E. Gregori, L. Lenzini, and S. Mainardi. Parallel k -clique community detection on large-scale networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(8):1651–1660, Aug 2013.
- [154] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015.
- [155] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. ROCK: A robust clustering algorithm for categorical attributes. In *IEEE International Conference on Data Engineering (ICDE)*, page 512, USA, 1999.

- [156] Lee M. Gunderson and Gecia Bravo-Hermsdorff. Introducing graph cumulants: What is the variance of your social network? 2020.
- [157] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning (ICML)*, pages 3887–3896, 2020.
- [158] Mahantesh Halappanavar, Hao Lu, Ananth Kalyanaraman, and Antonino Tumeo. Scalable static and dynamic community detection using grappolo. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2017.
- [159] Shuo Han, Lei Zou, and Jeffrey Xu Yu. Speeding up set intersections in graph algorithms using simd instructions. In *ACM SIGMOD International Conference on Management of Data*, pages 1587–1602, 2018.
- [160] Kathrin Hanauer, Monika Henzinger, and Qi Cheng Hua. Fully dynamic four-vertex subgraph counting. *CoRR*, abs/2106.15524, 2021.
- [161] Meng He, Ganggui Tang, and Norbert Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 128–140, 2014.
- [162] John Healy, Jeannette Janssen, Evangelos Milius, and William Aiello. Characterization of graphs using degree cores. In *International Workshop on Algorithms and Models for the Web-Graph (WAW)*, pages 137–148, 2007.
- [163] Katherine A. Heller and Zoubin Ghahramani. Bayesian hierarchical clustering. In *International Conference on Machine Learning (ICML)*, pages 297–304, 2005.
- [164] Monika Henzinger, Stefan Neumann, and Andreas Wiese. Explicit and implicit dynamic coloring of graphs with bounded arboricity. *CoRR*, abs/2002.10142, 2020.
- [165] Tomaz Hocevar and Janez Demsar. A combinatorial approach to graphlet counting. *Bioinformatics*, pages 559–65, 2014.
- [166] Paul W. Holland and Samuel Leinhardt. A method for detecting structure in sociometric data. *American Journal of Sociology*, 76(3):492–513, 1970.
- [167] Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. Cyclic pattern kernels for predictive graph mining. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, page 158–167. Association for Computing Machinery, 2004.

- [168] Yang Hu, Hang Liu, and H. Howie Huang. TriCore: Parallel triangle counting on gpus. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 14:1–14:12, 2018.
- [169] Q. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen. Faster parallel core maintenance algorithms in dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 31(6):1287–1300, 2020.
- [170] Qin Huang and W. E. Weihl. An evaluation of concurrent priority queue algorithms. In *IEEE Symposium on Parallel and Distributed Processing*, pages 518–525, 1991.
- [171] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. Querying k -truss community in large and dynamic graphs. In *ACM SIGMOD International Conference on Management of Data*, page 1311–1322, 2014.
- [172] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. Approximate closest community search in networks. *Proc. VLDB Endow.*, 9(4):276–287, December 2015.
- [173] Yihao Huang, Claire Wang, Jessica Shi, and Julian Shun. Efficient algorithms for parallel bi-core decomposition. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 17–32, 2023.
- [174] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, September 1999.
- [175] Shweta Jain and C. Seshadhri. A fast and provable method for estimating clique counts using Turán’s theorem. In *International World Wide Web Conference (WWW)*, pages 441–449, 2017.
- [176] 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.
- [177] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.
- [178] Siddhartha V. Jayanti and Robert E. Tarjan. A randomized concurrent algorithm for disjoint set union. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 75–82, 2016.
- [179] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravisankar Krishnawamy, and Rohan Kadekodi. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.
- [180] Yongkweon Jeon and Sungroh Yoon. Multi-threaded hierarchical clustering by parallel nearest-neighbor chaining. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 26(9):2534–2548, 2015.

- [181] Rut Jesus, Martin Schwartz, and Sune Lehmann. Bipartite networks of Wikipedia’s articles and authors: a meso-level approach. In *International Symposium on Wikis and Open Collaboration*, pages 1–10, 2009.
- [182] Madhav Jha, C. Seshadhri, and Ali Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *International World Wide Web Conference (WWW)*, pages 495–505, 2015.
- [183] Biaobin Jiang, Jiguang Wang, Jingfa Xiao, and Yong-Cui Wang. Gene prioritization for type 2 diabetes in tissue-specific protein interaction networks. *Syst Biol*, 11, 09 2009.
- [184] H. Jin, N. Wang, D. Yu, Q. Hua, X. Shi, and X. Xie. Core maintenance in dynamic graphs: A parallel approach based on matching. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 29(11):2416–2428, 2018.
- [185] H. Kabir and K. Madduri. Parallel k -core decomposition on multicore platforms. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1482–1491, May 2017.
- [186] Humayun Kabir and Kamesh Madduri. Parallel k-truss decomposition on multicore systems. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.
- [187] Haim Kaplan and Shay Solomon. Dynamic representations of sparse distributed networks: A locality-sensitive approach. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 33–42, 2018.
- [188] George Karypis, Eui-Hong (Sam) Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, August 1999.
- [189] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. k -core decomposition of large networks on a single PC. *Proc. VLDB Endow.*, 9(1):13–23, 2015.
- [190] Benjamin King. Step-wise clustering procedures. *Journal of the American Statistical Association*, 69:86–101, 1967.
- [191] Maksim Kitsak, Lazaros K. Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H. Eugene Stanley, and Hernán A. Makse. Identification of influential spreaders in complex networks. *Nature Physics*, 6(11):888–893, November 2010.
- [192] Ton Kloks, Dieter Kratsch, and Haiko Muller. Finding and counting small induced subgraphs efficiently. *Information Processing Letters*, 74(3):115–121, 2000.

- [193] Tamara G. Kolda, Ali Pinar, Todd Plantenga, C. Seshadhri, and Christine Task. Counting triangles in massive graphs with MapReduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014.
- [194] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 532–543, 2014.
- [195] Łukasz Kowalik. Short cycles in planar graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 284–296, 2003.
- [196] Łukasz Kowalik. Adjacency queries in dynamic sparse graphs. *Inf. Process. Lett.*, 102(5):191–195, 2007.
- [197] Łukasz Kowalik. Fast 3-coloring triangle-free planar graphs. *Algorithmica*, 58(3):770–789, 2010.
- [198] Dexter C Kozen. *Theory of Computation*, volume 121. Springer, 2006.
- [199] Jérôme Kunegis. KONECT: the Koblenz network collection. pages 1343–1350, 05 2013.
- [200] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *International World Wide Web Conference (WWW)*, pages 591–600, 2010.
- [201] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow.*, 12(10):1099–1112, June 2019.
- [202] Kartik Lakhotia, Rajgopal Kannan, Viktor K. Prasanna, and César A. F. De Rose. RECEIPT: Refine coarse-grained independent tasks for parallel tip decomposition of bipartite graphs. *Proc. VLDB Endow.*, 14(3):404–417, 2020.
- [203] G. N. Lance and W. T. Williams. A general theory of classificatory sorting strategies 1. Hierarchical systems. *Computer Journal*, 9(4):373–380, February 1967.
- [204] Matthieu Latapy, Clémence Magnien, and Nathalie Del Vecchio. Basic notions for the analysis of large two-mode networks. *Social Networks*, 30(1):31 – 48, 2008.
- [205] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. *A Survey of Algorithms for Dense Subgraph Discovery*, pages 303–336. Springer US, Boston, MA, 2010.
- [206] Charles E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3), 2010.

- [207] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, 2019.
- [208] Conggai Li, Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Efficient progressive minimum k -core search. *Proc. VLDB Endow.*, 13(3):362–375, November 2019.
- [209] R. Li, J. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge & Data Engineering (TKDE)*, 26(10):2453–2465, oct 2014.
- [210] Rong-Hua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. Ordering heuristics for k -clique listing. *Proc. VLDB Endow.*, 13(12):2536–2548, 2020.
- [211] Rong-Hua Li, Lu Qin, Fanghua Ye, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. Skyline community search in multi-valued networks. In *ACM SIGMOD International Conference on Management of Data*, page 457–472. Association for Computing Machinery, 2018.
- [212] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(10):2453–2465, 2014.
- [213] Don R Lick and Arthur T White. k -degenerate graphs. *Canadian Journal of Mathematics*, 22(5):1082–1096, 1970.
- [214] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. Hierarchical core maintenance on large dynamic graphs. *Proc. VLDB Endow.*, 14(5):757–770, 2021.
- [215] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. Efficient (α, β) -core computation in bipartite graphs. *Proc. VLDB Endow.*, 29(5):1075–1099, 2020.
- [216] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic algorithms for k -core decomposition and related graph problems. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 191–204. Association for Computing Machinery, 2022.
- [217] Ying Liu, Ming Tang, Tao Zhou, and Younghae Do. Core-like groups result in invalidation of identifying super-spreader by k -shell decomposition. *Scientific Reports*, 5:9602–9602, May 2015.
- [218] Tze Meng Low, Daniele G. Spampinato, Anurag Kutuluru, Upasana Sridhar, Doru Thom Popovici, Franz Franchetti, and Scott McMillan. Linear algebraic formulation of edge-centric k -truss algorithms with adjacency matrices. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2018.

- [219] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19 – 37, 2015. Special Issue: Graph Analysis for Scientific Discovery.
- [220] Shangqi Lu and Yufei Tao. Towards Optimal Dynamic Indexes for Approximate (and Exact) Triangle Counting. In *International Conference on Database Theory (ICDT)*, volume 186, pages 6:1–6:23, 2021.
- [221] Qi Luo, Dongxiao Yu, Zhipeng Cai, Xuemin Lin, and Xiuzhen Cheng. Hypercore maintenance in dynamic hypergraphs. In *IEEE International Conference on Data Engineering (ICDE)*, pages 2051–2056, 2021.
- [222] Qi Luo, Dongxiao Yu, Xiuzhen Cheng, Zhipeng Cai, Jiguo Yu, and Weifeng Lv. Batch processing for truss maintenance in large dynamic graphs. *IEEE Transactions on Computational Social Systems*, 7(6):1435–1446, 2020.
- [223] Qi Luo, Dongxiao Yu, Feng Li, Zhenhao Dou, Zhipeng Cai, Jiguo Yu, and Xiuzhen Cheng. Distributed core decomposition in probabilistic graphs. In *Computational Data and Social Networks*, pages 16–32, 2019.
- [224] Qi Luo, Dongxiao Yu, Hao Sheng, Jiguo Yu, and Xiuzhen Cheng. Distributed algorithm for truss maintenance in dynamic graphs. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 104–115, 2021.
- [225] D. Makkar, D. A. Bader, and O. Green. Exact and parallel triangle counting in dynamic graphs. In *IEEE International Conference on High Performance Computing (HiPC)*, pages 2–12, 2017.
- [226] Fragkiskos D. Malliaros, Maria-Evgenia G. Rossi, and Michalis Vazirgiannis. Locating influential nodes in complex networks. *Scientific Reports*, 6(1), 2016.
- [227] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.
- [228] D. Marcus and Y. Shavitt. Efficient counting of network motifs. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 92–98, 2010.
- [229] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.
- [230] D. Mawhirter, B. Wu, D. Mehta, and C. Ai. ApproxG: Fast approximate parallel graphlet counting through accuracy control. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 533–542, 2018.
- [231] Sourav Medya, Tianyi Ma, Arlei Silva, and Ambuj Singh. A game theoretic approach for k -core minimization. In *19th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1922–1924, 2020.

- [232] B. Menegola. An external memory algorithm for listing triangles. *Tech. report, Universidade Federal do Rio Grande do Sul*, 2010.
- [233] Robert Meusel, Sebastiano Vigna, Oliver Lehberg, and Christian Bizer. The graph structure in the web—analyzed on different aggregation levels. *The Journal of Web Science*, 1(1):33–47, 2015.
- [234] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, July 2019.
- [235] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. Scalable large near-clique detection in large-scale networks via sampling. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 815–824, 2015.
- [236] Robert J. Mokken. Cliques, clubs and clans. *Quality & Quantity*, 13(2):161–173, 1979.
- [237] Nicholas Monath, Avinava Dubey, Guru Guruganesh, Manzil Zaheer, Amr Ahmed, Andrew McCallum, Gokhan Mergen, Marc Najork, Mert Terzihan, Bryon Tjanaka, et al. Scalable agglomerative clustering. *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2021.
- [238] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k -core decomposition. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(2):288–300, 2012.
- [239] Benjamin Moseley, Kefu Lu, Silvio Lattanzi, and Thomas Lavastida. A framework for parallelizing hierarchical clustering methods. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2019.
- [240] Benjamin Moseley and Joshua R. Wang. Approximation bounds for hierarchical clustering: Average linkage, bisecting k -means, and local search. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3094–3103, 2017.
- [241] Daniel Müllner. fastcluster: Fast hierarchical, agglomerative clustering routines for r and python. *Journal of Statistical Software, Articles*, 53(9):1–18, 2013.
- [242] C St JA Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 1(1):445–450, 1961.
- [243] Crispin Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, s1-39(1):12–12, 1964.
- [244] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. on Alg. (TALG)*, 12(1):1–15, 2015.

- [245] Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 026(2):415–419, 1985.
- [246] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.
- [247] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, March 2018.
- [248] Clark F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Comput.*, 21(8):1313–1325, 1995.
- [249] Naoto Ozaki, Hiroshi Tezuka, and Mary Inaba. A simple acceleration method for the Louvain algorithm. *International Journal of Computer and Electrical Engineering*, 8:207–218, 01 2016.
- [250] Rasmus Pagh and Francesco Silvestri. The input/output complexity of triangle enumeration. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 224–233, 2014.
- [251] Rasmus Pagh and Charalampos E. Tsourakakis. Colorful triangle counting and a MapReduce implementation. *Inf. Process. Lett.*, 112(7):277–281, March 2012.
- [252] Xinghao Pan, Dimitris Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael I. Jordan. Parallel correlation clustering on big graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 1, page 82–90. MIT Press, 2015.
- [253] Ha-Myung Park and Chin-Wan Chung. An efficient MapReduce algorithm for counting triangles in a very large graph. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 539–548, 2013.
- [254] Ha-Myung Park, Francesco Silvestri, U Kang, and Rasmus Pagh. MapReduce triangle enumeration with guarantees. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 1739–1748, 2014.
- [255] Ha-Myung Park, Francesco Silvestri, Rasmus Pagh, Chin-Wan Chung, Sung-Hyon Myaeng, and U Kang. Enumerating trillion subgraphs on distributed systems. *ACM Trans. Knowl. Discov. Data*, 12(6), October 2018.
- [256] Merav Parter, David Peleg, and Shay Solomon. Local-on-average distributed tasks. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 220–239, 2016.
- [257] C. A. Phillips. Parallel graph contraction. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 148–157, 1989.

- [258] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. ESCAPE: Efficiently counting all 5-vertex subgraphs. In *International World Wide Web Conference (WWW)*, pages 1431–1440, 2017.
- [259] Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. High quality, scalable and parallel community detection for large real graphs. In *International World Wide Web Conference (WWW)*, number 23, page 225–236. Association for Computing Machinery, 2014.
- [260] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, August 2018.
- [261] X. Que, F. Checconi, F. Petrini, and J. A. Gunnels. Scalable community detection with the louvain algorithm. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 28–37, 2015.
- [262] U. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E Stat. Nonlin. Soft Matter Phys.*, 76, 2007.
- [263] M. Rahman, M. A. Bhuiyan, and M. Al Hasan. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(10):2466–2478, 2014.
- [264] Sanguthevar Rajasekaran. Efficient parallel hierarchical clustering algorithms. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16(6):497–502, 2005.
- [265] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.
- [266] Jörg Reichardt and Stefan Bornholdt. Statistical mechanics of community detection. *Phys. Rev. E*, 74:016110, Jul 2006.
- [267] J. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. Technical Report TR-08-85, Harvard University, 1985.
- [268] Jason Riedy, David A Bader, and Henning Meyerhenke. Scalable multi-threaded community detection in social networks. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, number 26, pages 1619–1628. IEEE, 2012.
- [269] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, page 4292–4293, 2015.
- [270] Ryan A. Rossi, Nesreen K. Ahmed, and Eunyee Koh. Higher-order network representation learning. In *Companion Proceedings of the The Web Conference (WWW)*, page 3–4. International World Wide Web Conferences Steering Committee, 2018.

- [271] Ryan A. Rossi, Rong Zhou, and Nesreen K. Ahmed. Estimation of graphlet counts in massive networks. *IEEE Trans. Neural Netw. Learning Syst.*, 30(1):44–57, 2019.
- [272] Randolph Rotta and Andreas Noack. Multilevel local search algorithms for modularity clustering. *ACM J. Exp. Algorithms*, 16, July 2011.
- [273] Aurko Roy and Sebastian Pokutta. Hierarchical clustering via spreading metrics. In *Advances in Neural Information Processing Systems (NIPS)*, volume 29, pages 2316–2324. Curran Associates, Inc., 2016.
- [274] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, USA, 2nd edition, 2003.
- [275] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, et al. Static graph challenge: Subgraph isomorphism. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2017.
- [276] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, et al. Graphchallenge.org: Raising the bar on graph analytic performance. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2018.
- [277] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sarıyüce, and Srikantha Tirthapura. Butterfly counting in bipartite networks. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 2150–2159, 2018.
- [278] Ahmet Erdem Sarıyüce. Motif-driven dense subgraph discovery in directed and labeled networks. In *The Web Conference (WWW)*, pages 379–390, 2021.
- [279] Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. Incremental k -core decomposition: Algorithms and evaluation. *Proc. VLDB Endow.*, 25(3):425–447, 2016.
- [280] Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. Streaming algorithms for k -core decomposition. *Proc. VLDB Endow.*, 6(6):433–444, April 2013.
- [281] Ahmet Erdem Sarıyüce and Ali Pinar. Fast hierarchy construction for dense subgraphs. *Proc. VLDB Endow.*, 10(3):97–108, November 2016.
- [282] Ahmet Erdem Sarıyüce and Ali Pinar. Peeling bipartite networks for dense subgraph discovery. In *International Conference on Web Search and Data Mining (WSDM)*, pages 504–512, 2018.
- [283] Ahmet Erdem Sarıyüce, C Seshadhri, and Ali Pinar. Local algorithms for hierarchical dense subgraph discovery. *Proc. VLDB Endow.*, 12(1):43–56, 2018.

- [284] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. Nucleus decompositions for identifying hierarchy of dense subgraphs. *ACM Trans. Web*, 11(3):16:1–16:27, July 2017.
- [285] N. S. Sattar and S. Arifuzzaman. Parallelizing Louvain algorithm: Distributed memory challenges. In *IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC)*, pages 695–701, 2018.
- [286] EN Sawardecker, CA Amundsen, M Sales-Pardo, and Luis AN Amaral. Comparison of methods for the detection of node group membership in bipartite networks. *The European Physical Journal B*, 72(4):671–677, 2009.
- [287] Saurabh Sawlani and Junxing Wang. Near-optimal fully dynamic densest subgraph. In *ACM SIGACT Symposium on Theory of Computing*, pages 181–193, 2020.
- [288] Satu Elisa Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, August 2007.
- [289] T. Schank. Algorithmic aspects of triangle-based network analysis. *PhD Thesis, Universitat Karlsruhe*, 2007.
- [290] Stephen B. Seidman. Network structure and minimum degree. *Soc. Networks*, 5(3):269–287, 1983.
- [291] Stephen B Seidman and Brian L Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical Sociology*, 6(1):139–154, 1978.
- [292] C. Seshadhri, Ali Pinar, Nurcan Durak, and Tamara G. Kolda. The importance of directed triangles with reciprocity: Patterns and algorithms. *CoRR*, abs/1302.6220, 2013.
- [293] Jianyin Shao, Stephen W. Tanner, Nephi Thompson, and Thomas E. Cheatham. Clustering molecular dynamics trajectories: 1. Characterizing the performance of different clustering algorithms. *Journal of Chemical Theory and Computation*, 3(6):2312–2334, 2007.
- [294] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In David van Dyk and Max Welling, editors, *Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 5, pages 488–495. PMLR, 16–18 Apr 2009.
- [295] Jessica Shi, Laxman Dhulipala, David Eisenstat, Jakub Łacki, and Vahab Mirrokni. Scalable community detection via parallel correlation clustering. *Proc. VLDB Endow.*, 14(11):2305–2313, jul 2021.
- [296] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 135–146, 2021.

- [297] Jessica Shi, Laxman Dhulipala, and Julian Shun. Theoretically and practically efficient parallel nucleus decomposition. *Proc. VLDB Endow.*, 15(3):583–596, feb 2022.
- [298] Jessica Shi and Julian Shun. Parallel algorithms for butterfly computations. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 16–30, 2020.
- [299] Julian Shun and Guy E Blelloch. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.
- [300] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012.
- [301] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 143–153, 2014.
- [302] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *Data Compression Conference (DCC)*, pages 403–412, 2015.
- [303] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W. Mahoney. Parallel local graph clustering. *Proc. VLDB Endow.*, 9(12):1041–1052, 2016.
- [304] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *IEEE International Conference on Data Engineering (ICDE)*, pages 149–160, 2015.
- [305] Shaden Smith, Xing Liu, Nesreen K Ahmed, Ancy Sarah Tom, Fabrizio Petrini, and George Karypis. Truss decomposition on shared-memory parallel systems. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2017.
- [306] Peter H.A. Sneath and Robert R. Sokal. *Numerical Taxonomy: The Principles and Practice of Numerical Classification*. W.H. Freeman, San Francisco, 1973.
- [307] Shay Solomon and Nicole Wein. Improved dynamic graph coloring. *ACM Trans. on Alg. (TALG)*, 16(3), June 2020.
- [308] C. L. Staudt and H. Meyerhenke. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(1):171–184, 2016.

- [309] Baris Sumengen, Anand Rajagopalan, Gui Citovsky, David Simcha, Olivier Bachem, Pradipta Mitra, Sam Blasiak, Mason Liang, and Sanjiv Kumar. Scaling hierarchical agglomerative clustering to billion-sized datasets. *CoRR*, abs/2105.11653, 2021.
- [310] Bintao Sun, T.-H. Hubert Chan, and Mauro Sozio. Fully dynamic approximate k -core decomposition in hypergraphs. *ACM Trans. Knowl. Discov. Data (TKDD)*, 14(4), May 2020.
- [311] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proc. VLDB Endow.*, 13(2):211–225, 2019.
- [312] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *International World Wide Web Conference (WWW)*, pages 607–614, 2011.
- [313] Kanat Tangwongsan, A. Pavan, and Srikanta Tirthapura. Parallel triangle counting in massive streaming graphs. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 781–786, 2013.
- [314] V. A. Traag. Faster unfolding of communities: Speeding up the louvain algorithm. *Phys. Rev. E*, 92:032801, September 2015.
- [315] V. A. Traag, L. Waltman, and N. J. van Eck. From Louvain to Leiden: Guaranteeing well-connected communities. *Scientific Reports*, 9(1):5233, March 2019.
- [316] Alok Tripathy, Fred Hohman, Duen Horng Chau, and Oded Green. Scalable k -core decomposition for static graphs using a dynamic graph data structure. In *IEEE International Conference on Big Data (BigData)*, pages 1134–1141, 2018.
- [317] Charalampos Tsourakakis. The k -clique densest subgraph problem. In *International World Wide Web Conference (WWW)*, page 1122–1132, 2015.
- [318] Charalampos E. Tsourakakis. Counting triangles in real-world networks using projections. *Knowl. Inf. Syst.*, 26(3):501–520, 2011.
- [319] Charalampos E. Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 1(2):75–81, Apr 2011.
- [320] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. DOULION: Counting triangles in massive graphs with a coin. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 837–846, 2009.

- [321] Charalampos E. Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. Scalable motif-aware graph clustering. In *International World Wide Web Conference (WWW)*, page 1451–1460. International World Wide Web Conferences Steering Committee, 2017.
- [322] Virginia Vassilevska. Efficient algorithms for clique problems. *Inf. Process. Lett.*, 109(4):254–257, 2009.
- [323] Nate Veldt, David F. Gleich, and Anthony Wirth. A correlation clustering framework for community detection. In *The Web Conference (WWW)*, page 439–448. International World Wide Web Conferences Steering Committee, 2018.
- [324] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proc. VLDB Endow.*, 5(9):812–823, May 2012.
- [325] Jia Wang, Ada Wai-Chee Fu, and James Cheng. Rectangle counting in large bipartite graphs. In *IEEE International Conference on Big Data (BigData)*, pages 17–24, 2014.
- [326] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 501–508, 2006.
- [327] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. Efficient computing of radius-bounded k -cores. In *IEEE International Conference on Data Engineering (ICDE)*, pages 233–244, 2018.
- [328] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Vertex priority based butterfly counting for large-scale bipartite networks. *Proc. VLDB Endow.*, 12(10), June 2019.
- [329] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Efficient bitruss decomposition for large-scale bipartite graphs. In *IEEE International Conference on Data Engineering (ICDE)*, pages 661–672, 2020.
- [330] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *Proc. VLDB Endow.*, pages 1–24, 03 2021.
- [331] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, Lu Qin, and Yuting Zhang. Efficient and effective community search on large-scale bipartite graphs. In *IEEE International Conference on Data Engineering (ICDE)*, pages 85–96, 2021.
- [332] Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. Parallel algorithm for core maintenance in dynamic graphs. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2366–2371, 2017.

- [333] P. Wang, J. Zhao, X. Zhang, Z. Li, J. Cheng, J. C. S. Lui, D. Towsley, J. Tao, and X. Guan. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 30(1):73–86, Jan 2018.
- [334] Yiqiu Wang, Yan Gu, and Julian Shun. Theoretically-efficient and practical parallel DBSCAN. In *ACM SIGMOD International Conference on Management of Data*, pages 2555–2571, 2020.
- [335] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.
- [336] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, 1998.
- [337] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. Yu. I/O efficient core graph decomposition: Application to degeneracy ordering. *IEEE Transactions on Knowledge & Data Engineering (TKDE)*, 31(01):75–90, jan 2019.
- [338] Sebastian Wernicke and Florian Rasche. FANMOD: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.
- [339] Xiangzhou Xia. Efficient and scalable listing of four-vertex subgraphs. Master’s thesis, Texas A&M University, 2016.
- [340] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, January 2015.
- [341] Hao Yin, Austin R Benson, and Jure Leskovec. Higher-order clustering in networks. *Physical Review E*, 97(5), 2018.
- [342] Shangdi Yu, Yiqiu Wang, Yan Gu, Laxman Dhulipala, and Julian Shun. Par-chain: A framework for parallel hierarchical agglomerative clustering using nearest-neighbor chain. *Proc. VLDB Endow.*, 15(2):285–298, 2021.
- [343] Wayne W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33(4):452–473, 1977.
- [344] J. Zeng and H. Yu. Parallel modularity-based community detection on large-scale graphs. In *IEEE International Conference on Cluster Computing*, pages 1–10, 2015.
- [345] Bin Zhang and Steve Horvath. A general framework for weighted gene co-expression network analysis. *Statistical Applications in Genetics and Molecular Biology*, 4(1), 2005.
- [346] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. When engagement meets similarity: Efficient (k,r) -core computation on social networks. *Proc. VLDB Endow.*, 10(10):998–1009, June 2017.

- [347] Haohua Zhang, Hai Zhao, Wei Cai, Jie Liu, and Wanlei Zhou. Using the k-core decomposition to analyze the static structure of large-scale software systems. *J. Supercomput.*, 53(2):352–369, 2010.
- [348] Y. Zhang, H. Jiang, F. Wang, Y. Hua, D. Feng, and X. Xu. LiteTE: Lightweight, communication-efficient distributed-memory triangle enumerating. *IEEE Access*, 7:26294–26306, 2019.
- [349] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. In *IEEE International Conference on Data Engineering (ICDE)*, pages 337–348, 2017.
- [350] Yang Zhang and Srinivasan Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1049–1060, 2012.
- [351] Yikai Zhang and Jeffrey Xu Yu. Unboundedness and efficiency of truss maintenance in evolving graphs. In *ACM SIGMOD International Conference on Management of Data*, page 1024–1041, 2019.
- [352] Yunhao Zhang, Rong Chen, and Haibo Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 614–630, 2017.
- [353] Feng Zhao and Anthony KH Tung. Large scale cohesive subgraphs discovery for social network visual analysis. *Proc. VLDB Endow.*, 6(2):85–96, 2012.
- [354] R. Zhu, Z. Zou, and J. Li. Fast rectangle counting on massive networks. In *IEEE International Conference on Data Mining (ICDM)*, pages 847–856, 2018.
- [355] Zhaonian Zou. Bitruss decomposition of bipartite graphs. In *Database Systems for Advanced Applications (DASFAA)*, pages 218–233, 2016.