On Designing Structure-Aware High-Performance Graph Algorithms

A thesis submitted for the degree of
Doctor of Philosophy

by
Mohsen Koohi Esfahani
Master of Computer Engineering, Isfahan University of Technology, Iran

at
The School of Electronics, Electrical Engineering, and Computer Science
Queen's University Belfast

Supervisors:
Hans Vandierendonck
Peter Kilpatrick

Examiners:
Paul Kelly, Imperial College London
Ivor Spence

October, 2022

On Designing Structure-Aware High-Performance Graph Algorithms

Abstract

Graph algorithms find several usages in industry, science, humanities, and technology. The fast-growing size of graph datasets in the context of the processing model of the current hardware has resulted in different bottlenecks such as memory locality, work-efficiency, and load-balance that degrade the performance. To tackle these limitations, high-performance computing considers different aspects of the execution in order to design optimized algorithms through efficient usage of hardware resources.

The main idea in this thesis is to analyze the structure of graphs to exploit special features that are key to introduce new graph algorithms with optimized performance.

First, we study the structure of real-world graph datasets with skewed degree distribution and the applicability of graph relabeling algorithms as the main restructuring tools to improve performance and memory locality. To that end, we introduce **novel locality metrics** including *Cache Miss Rate Degree Distribution*, *Effective Cache Size*, *Push Locality and Pull Locality*, and *Degree Range Decomposition*.

Based on this structural analysis, we introduce the **Uniform Memory Demands** strategy that (i) recognizes diverse memory demands and behaviours as a source of performance inefficiency, (ii) separates contrasting memory demands into groups with uniform behaviours across each group, and (iii) designs bespoke data structures and algorithms for each group in order to satisfy memory demands with the lowest overhead.

We apply the Uniform Memory Demands strategy to design three graph algorithms with optimized performance: (i) the **SAPCo Sort** algorithm as a parallel counting sort algorithm that is faster than comparison-based sorting algorithms in degree-ordering of power-law graphs, (ii) the **iHTL** algorithm that optimizes locality in Sparse Matrix-Vector (SpMV) Multiplication graph algorithms by extracting dense subgraphs containing incoming edges to in-hubs and processing them in the push direction, and (iii) the **LOTUS** algorithm that optimizes locality in Triangle Counting by separating different caching demands and deploying specific data structure and algorithm for each of them.

Acknowledgements

I am grateful to my great supervisors, Prof. Hans Vandierendonck and Prof. Peter Kilpatrick, for their comprehensive and continuous support in all steps of this project.

I am grateful to The Department for Economy, Northern Ireland and to The Queen's University Belfast for the scholarship of this project. Also, this work has been partially supported by two projects: Kelvin-2 Supercomputer (EPSRC grant EP/T022175/1) and DiPET (CHIST-ERA-18-SDCDN-002, EPSRC grant EP/T022345/1).

Contents

Chapter 1

Introduction

1.1 High-Performance Graph Processing and Its Challenges

Graphs are simply-defined data structures to represent relationships between objects. Graph algorithms extract information from these data and are used in a wide range of different fields of science, industry, and humanities. Finding the shortest path in computer networks and map applications, searching and ranking web pages [92], recommendation engines [161], job scheduling [113], and graph databases [5] are samples of graph applications. In intelligence analysis, graphs are used to identify suspicious activities and threats [40, 141]. In social sciences, graphs are used to represent groups and social circles, and to model exchange networks and interpersonal communications [27, 101, 129]. In biology, graphs are used to represent protein-protein complex structure [47], to model human and yeast protein interaction [165], and for drug discovery [1].

The size of graph datasets, similar to the size of data produced in different fields, is growing fast and public graph datasets have up to hundreds of billions of edges. This large size of graph datasets in the context of the processing model and current hardware has resulted in long execution times for graph algorithms.

On the other hand, the performance of graph algorithms is also affected by the **structure of graph datasets**. Many real-world graphs derived from social networks, the internet and the world-wide web, or from bio-informatics, show a skewed degree distribution, following a **power-law distribution**: a small fraction of vertices with very large degrees are connected to a disproportionately large fraction of edges. This special structure and the large size of graph datasets have negative effects on the performance

of graph algorithms:

- The memory access patterns of graph algorithms are specified by the structure of graph datasets. This makes it hard to use hardware caches (as the main resources designed to reduce memory accesses) efficiently for graph algorithms as the size of cache is much smaller than the total size of data that is continuously accessed, i.e., only a small fraction of the data can be efficiently accessed through cache. As a result, it is necessary to improve **cache reuse** to provide better performance.
- As cache cannot satisfy a large portion of memory accesses, and since a low number of computation instructions per memory access (load and store instructions) is performed, memory access is the bottleneck of graph processing. This shows that reducing the number of memory accesses directly impacts the **work-efficiency** and, subsequently, performance of the graph algorithms.
- The skewed degree distribution of these graphs and the dependency of the work performed for each vertex/edge on the type of graph traversal necessitate designing effective partitioning algorithms to optimize **load-balance** (i.e., to keep all concurrent processors busy during the total execution time) of graph algorithms and to reduce the execution time.

To deal with these challenges two general algorithmic approaches exist: (i) theoretical analysis of algorithms and (ii) experimental analysis of algorithms in the context of real execution environment.

In the first approach, a simple processing model is assumed and algorithms are analyzed and designed for this model. However, as mathematical models of the modern processors with their various complicated facilities (like memory prefetching, multi-level caching, and branch prediction) are not accessible either theoretically or practically, **pure algorithm analysis** cannot take the execution environment and its implications into account and cannot provide more details than complexity analysis of algorithms.

In the second approach, high-performance computing researchers use **experimental methods** to evaluate different factors that affect the execution time of graph algorithms.

They use the results of these studies to extract **falsifiable insights**[1] that are used to introduce new algorithms and processing environments (such as processor, cache, memory, and network architectures) with optimized performance as a result of better adjustment of algorithms to the computing environments and vice versa.

As examples, they measure execution time and collect the number of memory accesses, hardware instructions, and cache misses (e.g., [137]); they simulate execution of graph algorithms to extract the special metrics (e.g., [149, 179]); they measure the idle time of parallel processors (e.g., [7, 173]); they evaluate the scalability of implementations by changing the number of processors/machines involved in processing (e.g., [130]); they measure and compare the communication and computation times in distributed graph algorithms (e.g., [177]); they measure the effects of different prefetching algorithms, cache sizes, and cache replacement algorithms (e.g., [12, 171]); they consider effects of different types of memory accesses and/or different types of memory architectures (e.g., [153, 173]).

The **diversity of graph algorithms** and their **memory access patterns**, however, has made it difficult (or still impossible) to experience the best performance in reusing an optimized graph algorithm (or an optimized generalization/abstraction of graph algorithms) for other ones [137]: in some graph algorithms data and/or weights are assigned to vertices, while in others to edges; some graph algorithms require accessing data of neighbours of vertices, while others may also need to access the neighbours of each neighbour of a vertex; in some graph algorithms vertices are processed in the order

[1] According to the Philosophy of Science, we deploy (at least) two "discovery" methods to achieve new "knowledge": (i) the "Justification" method, in which, we use induction to infer new results from the previous deductions or (imagined-as-true) assumptions, and (ii) the "Falsification" method, in which, we prune the falsifiable insights by experiments in order to get closer to knowledge. In this approach, it is necessary for an insight (i.e., temporary knowledge) to be falsifiable: an experiment exists or can be designed to evaluate its correctness or wrongness (for each individual case in its application domain).

In the Falsification approach, experiments are used to prune the insights; in other words, the main role of the experiments is not to prove the correctness of a theory (i.e., insight) as experiments are used as counterexamples to identify the special conditions where a theory is wrong (that is why the insights should be falsifiable). However, the sequence of experiments are continued until we reach the last (in our timing manner) finest theory that is closer to the reality/knowledge. While these theories are falsifiable, the last-level experiment shows the adaptability of the last theory with the environment of the experiments, and does not act as a counterexample. This also shows that we need to extend the domain of our experiments to cover more cases in order to better refine our insights.

For further explanations, we refer the reader to the philosophies of Popper and Descartes [91, 116, 126] and their applications in Computer Science studies [136].

derived by their IDs, while in others the order of processing is specified by the previously processed vertices; in some graph algorithms the size of data that is assigned to a vertex/edge, is constant during the execution but it may vary in other algorithms; in some graph algorithms cache is mostly dedicated to data of vertices, while in others cache may be used mainly to store the topology data. On the other hand, some graph algorithms have unique features that are effective in delivering a better performance, but that opportunity may be lost by deploying abstraction[2].

Different efforts have been performed to tackle these problems:

- A group of studies have concentrated on providing programming abstractions or processing models for graph algorithms to design and implement **graph processing frameworks** that efficiently use hardware resources. These frameworks can be divided into (i) shared-memory graph frameworks that operate on a single machine (e.g. [18,51,159,173]), (ii) distributed-memory graph frameworks that deploy multiple machines to process graph datasets (e.g., [38,67,69,73,88,112,146,177]), (iii) out-of-core frameworks that use external memory as an intermediate storage to process graphs with memory requirements larger than the available memory (e.g., [2,26,105,132,178]), (iv) Graphics Processing Unit (GPU)-based graph frameworks (e.g. [21,62,70,75,76,117]), or (v) combinations of the previous items (e.g. [172]).
- On the other hand, some studies concentrate on optimizing **individual graph algorithms** (e.g., [13,60,109,151,155]).

1.2 Research Questions

While effective efforts have been performed to accelerate graph algorithms, these efforts are, mainly, focused on optimizing graph algorithms without considering the implications of the structure of graph datasets on the performance of graph algorithms. The previous studies on improving performance by using the structure of graph datasets are divided into two categories: (i) improving graph partitioning [38,154,172] and (ii) designing graph relabeling algorithms (e.g. [6,108,163]) that change the order of vertices

[2]As an example, in some graph algorithms we can process an edge twice, in others we cannot do that as the results may be wrong. Now, if we want to make an abstraction, we need to restrict all edges to be processed once and this does not allow the implementation of the first group to benefit from the flexibility of processing some edges multiple times.

aiming to provide better memory locality (therefore, better performance) during graph traversal (i.e., traversing vertices and their neighbours). We review these works in Section 2.4.3 and Section 2.4.7.

Our insight is that we need more investigations about the functionality and effectiveness of the previous works on real-world graphs with skewed degree distribution. Moreover, the structure of graphs has unidentified implications on the performance of graph algorithms and requires more analysis in order to understand its effects on different performance bottlenecks. This analysis helps us to design structure-aware algorithms with optimized performance.

We try to answer the following **research questions** in this thesis:

- How do relabeling algorithms affect performance of different real-world graphs with skewed degree distribution? What are the limitations of relabeling algorithms to improve memory locality?
- What are the structural differences between different types of these real-world graph datasets?
- How can we exploit the features of these graphs to design algorithms with better performance?

1.3 Research Scope

In order to specify the **research scope**, we consider the following aspects:

- In this thesis, we study the performance of graph algorithms for shared-memory CPU-based systems. However, the insights gained from our study explain features of data and efficient processing strategies that apply in equal measure to different processing platforms such as GPU-based and distributed-memory computing.
- We consider the impacts of real-world graph datasets with skewed degree distributions close to the power-law distribution. They are the largest public graph datasets. While mathematical formulations of power-law degree distribution exist, since most real-world graphs are not accurately power-law [31], we consider the real-world graphs with skewed degree distributions and without limiting the domain of our study to an accurate mathematical model.
- We limit our study to static graphs. Dynamic graph algorithms [48, 72] (where

edges and vertices are gradually added or removed) and streaming graph algorithms [114] (that assume a limited size of available memory) are not considered.

- We assume the graph topology is completely stored in the main memory and there is no interaction with a secondary storage (like disk, SSD, NAS/SAN, or NVMM) or network devices during the execution of the graph algorithms. We also do not consider different methods of loading graphs to the main memory (like reading from disk, collecting from databases, or calling APIs) and the execution time of a graph algorithm does not include the loading time.

1.4 Contributions & Publications

First, in Chapter 3, we investigate the effects of graph relabeling algorithms as the main restructuring tools that are used to improve performance of graph algorithms. Graph relabeling algorithms assign new IDs to vertices in order to change the order of memory accesses aiming to provide better locality in memory accesses and increasing cache hits. We study the structure of power-law graph datasets and the effects of relabeling algorithms on these graphs by introducing new locality metrics including *Cache Miss Rate Degree Distribution*, *Push Locality and Pull Locality*, and *Degree Range Decomposition*. This work has been presented in:

- M. Koohi Esfahani, P. Kilpatrick and H. Vandierendonck, "How Do Graph Relabeling Algorithms Improve Memory Locality?," 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '21) [95], and
- M. Koohi Esfahani, P. Kilpatrick and H. Vandierendonck, "Locality Analysis of Graph Reordering Algorithms," 2021 IEEE International Symposium on Workload Characterization (IISWC '21) [96].

The structural analysis of graph datasets in Chapter 3 has a clear message: different subgraphs of real-world graphs with skewed degree distribution have contrasting behaviours even in a simple graph traversal. Some vertices have a low number of neighbours and their processing benefits from caching. The others with great numbers of neighbours flush cache contents and reduce cache reuse.

To tackle this problem, we introduce the **Uniform Memory Demands** strategy in Chapter 4 that addresses the diversity of memory demands in graph algorithms and divides the vertices and/or edges into a number of groups, where each group has uniform

memory demands and behaviours. Then, by designing special data structures and algorithms for each group, their uniform memory demands are satisfied without sacrificing performance.

We design three graph algorithms by deploying the Uniform Memory Demands strategy. In Chapter 5, we design the **SAPCo Sort** algorithm that optimizes performance in degree-ordering of power-law graphs and is faster than the comparison-based sorting algorithms. SAPCo Sort, assigns shared data structures for high-degree vertices, while assigning private data structures for low-degree vertices. This work has been presented in:

- M. Koohi Esfahani, P. Kilpatrick and H. Vandierendonck, "SAPCo Sort: optimizing Degree-Ordering for Power-Law Graphs," 2022 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS '22 [100].

In Chapter 6, we use the Uniform Memory Demands strategy to design the **iHTL** algorithm that optimizes memory locality in Sparse Matrix-Vector (SpMV) Multiplication graph algorithm. iHTL extracts subgraphs containing incoming edges to in-hubs and processes them in the push direction instead of the pull direction. In this way, cache is dedicated to destination of these edges that are much lower in count (than their source vertices) and data of destinations can be easily maintained in cache. This work has been presented in:

- M. Koohi Esfahani, P. Kilpatrick and H. Vandierendonck, "Exploiting in-Hub Temporal Locality in SpMV-based Graph Processing," 50th International Conference on Parallel Processing (ICPP '21) [94].

Chapter 7 presents the **LOTUS** algorithm that optimizes memory locality in Triangle Counting by deploying the Uniform Memory Demands Strategy. Lotus divides the execution into a number of steps (based on the presence of hub edges in the triangles) and in each step uses a bespoke data structure and algorithm to concentrate compact memory accesses into the cache. This work has been presented in:

- M. Koohi Esfahani, P. Kilpatrick and H. Vandierendonck, "LOTUS: Locality optimizing Triangle Counting," 27th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '22) [98].

While the Uniform Memory Demands strategy concentrates on optimizing graph algorithms that are affected by the degree distribution of the real-world graphs, we

have also considered the special connectivity of these graphs and its impacts on the connectivity-based graph algorithms to introduce work-efficient algorithms that do not traverse all edges. These works are not covered in this dissertation and have been published in:

- M. Koohi Esfahani, P. Kilpatrick and H. Vandierendonck, "Thrifty Label Propagation: Fast Connected Components for Skewed-Degree Graphs," 2021 IEEE International Conference on Cluster Computing (CLUSTER '21) [97] and
- M. Koohi Esfahani, P. Kilpatrick and H. Vandierendonck, "MASTIFF: Structure-Aware Minimum Spanning Tree/Forest," Proceedings of the 36th ACM International Conference on Supercomputing (ICS '22) [99].

1.5 Thesis Structure

We review the preliminary definitions and concepts and also the related works in Chapter 2.

In Chapter 3, we investigate the application of graph reordering algorithms for different power-law graph datasets.

Chapter 4 introduces the Uniform Memory Demands strategy. We deploy this strategy to design three algorithms. Chapter 5 introduces the SAPCo algorithm, a structure-aware parallel counting sort with optimized performance for degree-ordering of power-law graphs. Chapter 6 introduces the iHTL algorithm that optimizes locality in SpMV-based graph processing. The LOTUS algorithm that optimizes locality of triangle counting is designed and evaluated in Chapter 7.

Chapter 8 summarizes the efforts and proposes some directions for future research.

The experimental setup is explained in Appendix A.

Chapter 2

Background

In this chapter, we lay the foundation of the discussions. We start by defining graph types and representations in Section 2.1 that is followed by explanation of the skewed-degree distribution in Section 2.2. Then, we explain the performance bottlenecks in Section 2.3 and we review the literature in Section 2.4.

2.1 Terminology and Graph Representations

A **graph** $G = (V,\ E)$ has a set of vertices V, and a set of *directed* edges E. E is a set of ordered pairs such as $(i,\ j)$ that represents an edge from vertex i to vertex j. The **adjacency matrix** of a graph is a binary matrix representation of the graph: the element at row i and column j is 1 if E contains an edge from vertex i to j, and 0 otherwise.

N_v^-, and N_v^+ are the set of **in-neighbours** and **out-neighbours** of vertex v, respectively [143]. N_v is the set of neighbours of vertex v, i.e., $N_v = N_v^- \cup N_v^+$. In the adjacency matrix, row i represents the out-neighbours of vertex i, i.e., N_i^+ and column j represents in-neighbours of vertex j, i.e., N_j^-.

An **undirected graph** $G_u = (V,\ E_u)$ has a set of vertices V, and a set of *undirected* edges E_u between these vertices. E_u is a set of unordered pairs such as $\{i, j\}$ that represents an edge between vertices i and j. In other words, for each undirected edge $\{i, j\}$, two directed edges exist: (i, j) and (j, i). As such, if we represent the undirected graph $G_u = (V, E_u)$ as the directed graph $G = (V, E)$, we have $|E| = 2|E_u|$.

In an undirected graph, for each directed edge (i, j), the directed edge (j, i) also exists. So, for each vertex v; $N_v^- = N_v^+ = N_v$. The adjacency matrix of an undirected graph is a symmetric matrix and contains $2|E_u|$ non zero elements.

The **average degree** of a graph is defined as $\frac{|E|}{|V|}$. Real-world graphs have an average degree which is much smaller than |V|; in other words, each vertex has a very small number of edges to other vertices. As a result, a row or column in the adjacency matrix has a very small number of 1 values. In this way, these graphs are called **sparse**.

Since the adjacency matrix representation of a sparse graph with a few billion vertices requires a memory size of ExaBytes, data structures with better memory utilization are required. The **Compressed Sparse Columns (CSC)**, and **Compressed Sparse Rows (CSR)** formats [133] are used for representing the in-neighbours and out-neighbours of vertices, respectively. The idea is to store the ID of neighbours of vertices instead of assigning a bit for every possible edge in the adjacency matrix.

CSC and CSR use two arrays: (1) an *offsets* array containing $|V| + 1$ elements, and (2) an *edges* array of $|E|$ elements. The offsets array is indexed by a vertex ID (a number between 0 and $|V|$) and specifies the index of the first edge of that vertex in the edges array. The edges array specifies the IDs of source endpoints of edges in CSC and IDs of destination endpoints of edges in CSR.

Since CSR and CSC representations are the same for a symmetric adjacency matrix, we use the **CSX** notation for the representation of an undirected graph.

2.2 Skewed Degree Distribution

The **degree distribution** of a graph is a plot that shows the frequency of vertices with the same degree. As an example, in Figure 2.1 the plot named "Frequency" (shown on the left y-axis) shows the degree distribution of the undirected ClueWeb12 graph (Appendix A) with 1 billion vertices and 75 billion edges. This figure shows that for each degree greater than 10K, the number of vertices is smaller than 100.

The plot named "Cumulative Frequency" (shown on the right y-axis) in Figure 2.1 shows the percentage of cumulative number of vertices with degrees less than or equal to a degree on the x-axis. This plot shows that vertices with degrees less than 1000 include more than 99.3% of the vertices.

This **skewed degree distribution**, that is mathematically modeled as a **power-law distribution**, is observed in real-world graphs: a very small percentage of vertices have very large degrees while, most of vertices have very small degrees.

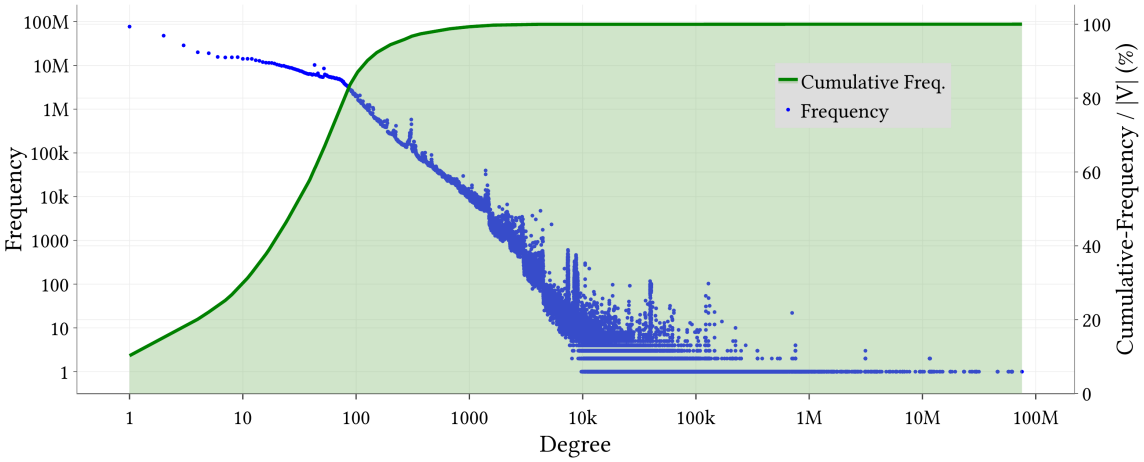


Figure 2.1: Degree distribution of undirected ClueWeb12 ($|V| = 1B$, $|E| = 75B$)

2.3 Performance Bottlenecks

Performance of parallel graph algorithms is affected by a number of factors:

- **Memory Locality**. Locality is defined as *"the tendency for programs to cluster references to subsets of address space for extended periods"* [49]. By improving locality, **cache reuse** is increased and the time processors wait for fetching data from memory is reduced. Memory locality plays an important role in data-intensive applications (such as graph algorithms) as memory accesses represent a substantial portion of the overall processing time.
- **Load-Balance**. In order to process algorithms by multiple processors (and/or machines), **partitioning algorithms** are used to divide the total task into smaller subtasks that are assigned to processors. The optimal condition is to divide the work in a way that all processors are busy for the same amount of time during the execution time.
- **Work-Efficiency**. A task may be performed in multiple ways, and the amount of total work may be different in each way. The algorithm with the lower number of hardware instructions is more work-efficient.

2.4 Literature Review

Various programming abstractions and models have been introduced in in high-performance graph processing literature that we review in this section.

2.4.1 Matrix-Vector Multiplication

The iterative matrix-vector multiplication model is an abstraction for implementing graph algorithms [65,88,134]. In this model, each vertex has a data (or property) and the graph algorithms are modeled as multiplication of the adjacency matrix of the graph by the data of the vertices (as a vector).

For a graph algorithm that visits neighbours of each vertex and processes their data, the matrix-vector multiplication uses the adjacency matrix representation to model the graph traversal. Each row of the adjacency matrix is multiplied by the vector to identify value of the corresponding index in the result. An element on column i and row j of the adjacency matrix is 1, only if, there is an edge between j and i. Moreover, the elements with 0 value on the adjacency matrix do not affect the multiplication. As a result, multiplication of row j by the vector will be sum of those indices of the vector that are neighbours of j. In other words, matrix-vector multiplication is equal to traversing neighbours of vertices.

In Pegasus [88], the user defines three functions as operators to specify the actions to be performed for (i) processing each edge of a vertex, (ii) for merging the output of processing different edges of a vertex, and (iii) for specifying the new value for data of a vertex.

This algebraic model is used for implementing a number of graph algorithms including Breadth First Search (BFS), Connected Components (CC), Single Source Shortest Path (SSSP), Hyperlink Induced Topic Search [92], Belief Propagation [86], PageRank [29], Community Detection [176], and HADI [87].

If the graph is sparse, the adjacency matrix is sparse, and the Matrix-Vector Multiplication is called **Sparse Matrix-Vector Multiplication (SpMV)** graph processing that operates on the sparse representations of the graph.

Pegasus [88] uses the MapReduce programming model to implement each iteration of the Matrix-Vector multiplication: after processing edges, the output is merged based on the neighbourhood of vertices to identify new data of vertices.

Cache-blocking is a technique to improve locality of SpMV [78,79]. It works based on dividing the vector into blocks and applying multiplication in multiple steps in order to limit the range of data that is accessed in each step that consequently improves locality. The effects of blocking may be improved by ordering vertices by their degrees [174] and by applying the blocking only for vertices with the highest degrees [169]. We discuss

these techniques in more detail in Section 6.5.

2.4.2 Pregel's Bulk Synchronous Parallel

Bulk Synchronous Parallel (BSP) model [158] divides the execution into a number of steps where machines perform parallel computation during each step and communications are performed at the end of steps.

The BSP model was first used in graph processing by Pregel [112] for distributed-memory graph processing. Pregel performs parallel processing of vertices in each step using user-defined functions and performs communication at the end of each step. In Pregel, vertices may send messages to other vertices and the messages will be accessible to the destination in the next step.

Pregel's BSP, like SpMV, follows an iterative model; however, it is not required to traverse all edges in each step of BSP. Pregel assigns a *state* to each vertex and in each step, processes only vertices with *active* states. A vertex can deactivate itself or reactivate other vertices. The steps continue while at least one active vertex exists. The same semantic is used in shared-memory graph processing frameworks such as [51, 143, 153, 159, 173] where the set of active vertices of each iteration is called a *frontier* or *worklist*.

In this way, Pregel's BSP provides more efficient graph traversals than SpMV for graph algorithms that do not require processing of all edges in each step (like BFS, CC, SSSP). For these algorithms Sparse Matrix-Sparse Vector (SpMSpV) Multiplication [32, 64] is used as the vector in the multiplication is not permanently dense.

2.4.3 Structure-Aware Graph Partitioning

Pregel divides vertices between machines. This one-dimensional graph partitioning incurs load-imbalance as the work performed per partition may vary, especially, in power-law graphs whose vertices have widely differing numbers of edges.

PowerGraph [67] introduces a two-dimensional partitioning algorithm that is applied on edges (and not vertices) in order to provide better load-balance. PowerGraph minimizes ghost data (local copy of data and neighbourhood of some vertices that are stored on different machines instead of only one machine) [69] of high-degree vertices by limiting the number of machines that process their edges between.

PowerLyra [38] reduces the communication cost of PowerGraph (Section 2.4.2) by a partitioning algorithm that distributes edges of high-degree vertices between machines

and keeps edges of low-degree vertices in the same machine. In this way, PowerLyra ensures that low-degree vertices do not incur high communication costs (as only high-degree vertices are replicated which are much smaller in count in comparison to low-degree vertices) and processing of high-degree vertices experiences better load balance.

In order to provide better load balance in using CPU and GPU integrated devices, FinePar [172] assigns high-degree vertices to CPU and processes low-degree vertices using GPU processors as it is easier to provide load balance for low-degree vertices since (i) the number of low-degree vertices is much greater than the number of high-degree vertices, and (ii) the amount of work performed for each low-degree vertex is smaller.

VEBO [154] introduces a partitioning algorithm that distributes high-degree vertices on different partitions, while trying to keep equal number of edges for each partition. In this way, VEBO provides better load balance as it prevents high-degree vertices from being assigned to a small number of partitions.

2.4.4 Asynchronous and Partially Asynchronous

To reduce the overhead of global synchronization at the end of each BSP step, it is required to have sufficient amount of work in each step such that all machines are fully occupied. In graph algorithms with varying amounts of work in different steps, it is not straightforward to reach that target and load-imbalance happens. This becomes worse if we need a large number of steps as global synchronization at the end of a step may require more time than its computation.

To reduce the load imbalance of synchronization, asynchronous graph algorithms [74, 124] were introduced that do not require global synchronization. However, asynchronous algorithms may perform excess work as new works are requested as soon as new updates are found, but newer updates may come and invalidate the older updates and make the work performed for those older updates redundant[1]. These multiple updates for a vertex could happen in the same step of BSP without imposing excess work as the latest updates of vertices are used for triggering new updates (i.e., new edges to be processed) in the next step.

As a result, there is a trade-off between load imbalance of synchronous algorithms

[1] As an example, in finding the shortest path, assume a new shorter path is found for a vertex, and all of its neighbours need to update their shortest distances. Then, a new shorter path is found for that vertex that results in new updates for its neighbours. The new updates makes the older updates redundant.

and work inefficiency of asynchronous algorithms. KLA [73] introduces a distributed-memory graph framework with limited number of asynchronous steps after each global synchronization. In this way, KLA keeps the processors busy while it controls the amount of excess work. KLA dynamically adjusts the number of asynchronous steps by considering the visited vertices and costs of performed excess work.

2.4.5 Out-of-Core Graph Processing

Out-of-core graph processing frameworks use secondary storage to process graphs that are too large to fit in the main memory. The main challenges in out-of-core processing are to design a partitioning algorithm that reduces the number of times data is loaded from/stored on disk.

GraphChi [105] partitions edges based on their destinations and writes incoming edges to vertices of each part in a file. In each file, edges are sorted by their sources. In this way, updates for the outgoing edges of the consecutive vertex IDs are written sequentially in the files. For processing a partition, the related file is read completely, and the new data of vertices in this partition is calculated. Then, the file of each partition is partially updated for the outgoing edges from vertices in the current partition to edges in that partition.

X-Stream [132] introduces a **buffering** mechanism in disk-based graph processing. X-Stream divides edges based on their sources. In the scatter phase, each vertex processes its outgoing edges and writes the related updates in a buffer. Buffers are stored on disk after being partially sorted based on the partition of the relevant vertices. In the gather phase, the buffers containing updates of each partition are read and updates of vertices are merged to specify their new data that are stored on the relevant file.

The buffering mechanism of X-Stream benefits from **hardware prefetching** that provides better bandwidth for sequential accesses in comparison to random accesses either for main memory or for secondary storage.

GridGraph [178] uses 1D and 2D partitioning to reduce the overheads of reading and writing data in out-of-core processing.

2.4.6 NUMA-Aware Optimizations

To accelerate memory accesses in shared-memory graph processing in NUMA architectures, Polymer [173] allocates memory required for topology data on local NUMA

nodes and assigns NUMA-interleaved memory for vertex data.

Polymer uses the **partitioning of edges by destinations** of GraphChi (Section 2.4.5) for push traversal that changes random memory accesses in writing data of vertices from remote (on different NUMA nodes) to local (on the processor's NUMA node). This partitioning does not require protecting data of vertices from concurrent writes of parallel threads as each partition is assigned to a thread. However, this partitioning requires reading data of source vertices several times; in the worst case, for processing each partition.

Polymer introduces **edge-balanced partitioning** for improving load balance in shared-memory graph processing: each partition contains consecutive vertices and the number of edges in different partitions is close to $\frac{|E|}{P}$, where P is number of partitions.

GraphGrind [153] deploys NUMA-aware work-stealing in shared-memory graph processing and introduces Compressed CSR and CSC (CCSR and CCSC) topology data representation that facilitate better memory space utilization by storing vertex IDs in the offsets list to prevent allocating memory for zero-degree vertices.

2.4.7 Optimizing Locality

Since memory accesses are the bottleneck of graph processing, it is important to improve locality in a way that more reuse of cache contents happens.

The graph relabeling (also called "reordering" or "renumbering") algorithms try to increase the cache hit rate by changing the order in which vertices and their edges are processed, i.e., the order in which random memory accesses are made. In other words, a relabeling algorithm assigns new IDs to vertices to improve the clustering of memory accesses into a range that can be mostly satisfied by cache contents.

Figure 2.2 shows an example of a graph and its relabeled version and compares cache reuse in processing incoming edges to vertices.

On the left side of this figure, we see the main graph and a timetable that shows the contents of cache in processing this graph. In processing vertex 1, edges 3, 4, and 7 are accessed in steps 1–3 and cache contains data of vertices 4,7 at the end of processing vertex 1. We see that only one reuse of cache contents happens for this graph.

On the right side of Figure 2.2 a relabeled version of the graph is shown. The timetable of the reordered graph shows that in processing incoming edges to the vertices, we have 5 reuses of cache contents.

We present a comprehensive study of reordering algorithms in Chapter 3.

Time	Source	Destin.	Cache	Reuse
1	3	1	3	✗
2	4	1	4,3	✗
3	7	1	7,4	✗
4	5	2	5,7	✗
5	6	2	6,5	✗
6	1	4	1,6	✗
7	3	4	3,1	✗
8	6	5	6,3	✗
9	8	5	8,6	✓
10	8	6	8,6	✗
11	3	7	3,8	✗
12	1	8	1,3	✗

Time	Source	Destin.	Cache	Reuse
1	2	1	2	✗
2	3	1	3,2	✗
3	3	2	3,2	✓
4	4	2	4,3	✗
5	4	3	4,3	✓
6	6	4	6,4	✗
7	6	5	6,4	✓
8	7	5	7,6	✗
9	5	6	5,7	✗
10	7	6	5,7	✓
11	8	6	7,8	✗
12	7	8	7,8	✓

Figure 2.2: Effects of relabeling on processing incoming edges to vertices - Cache size = 2

On the other hand, space filling curves can improve temporal locality without relabeling the graph. These techniques have first been investigated for dense linear algebra [37, 52, 115]. More recently they have been applied to graph processing [122, 152]. They are most easily applied when the graph is stored in a coordinate list representation (that is, the graph is represented as an array of edges and each edge contains IDs of its source and destination vertices).

Chapter 3

Analysis of Graph Relabeling Algorithms and Graph Datasets

3.1 Introduction

As the first step of our study on designing structure-aware graph algorithms, in this chapter, we analyze the functionality of graph relabeling algorithms as the main restructuring tools designed to improve memory locality. This study helps us to understand the limitations of graph relabeling algorithms, the main structural features of real-world graphs with skewed degree distribution, and how these features affect the functionality of relabeling algorithms and partitioning algorithms. Moreover, we will identify the design requirements of graph algorithms that should be taken into account in order to design efficient algorithms for these graph datasets.

In Section 2.4.7, we explained that a **relabeling algorithm (RA)** improves locality of a graph traversal by assigning new IDs to vertices in order to cluster memory accesses into a shorter range that can be better satisfied by cache contents.

However, as identifying the optimal order is a NP-complete problem [163], different heuristics are employed in RAs based on assumptions about graph structure or execution environment [6, 22, 25, 45, 71, 90, 108, 125, 142, 145, 150, 163, 166].

Some studies investigate the impact of RAs on graph analytics [10, 11, 44, 59] by evaluating the general effect of RAs based on the execution time of graph analytics and do not explain how RAs work or how they affect locality of different graphs in different ways, useful for some, neutral or destructive for others. In order to reach effective and applicable locality optimizing algorithms, there is still a need to understand the

strengths and weaknesses of previous efforts.

A key stumbling block to analyzing RAs is the availability of suitable metrics and tools. Numerous metrics are available, but none is fully effective in providing insight into vertex relabeling. Graph topology metrics [28, 110, 111] summarize the characteristics of graphs independently of execution properties like processing order and vertex ID assignment. As such, they are great for analysing the graph, but do not reflect on execution efficiency. Reuse distance curves [19, 53, 54, 85] are an established means to assess the general degree of locality in programs. In the case of graph processing, reuse distance distributions are determined by the processing order and vertex IDs; however, they do not facilitate analysing the effectiveness (or shortcomings) of RAs. Moreover, reuse distance curves are practical only for comparing locality of a graph as a whole and do not reveal detailed information about the impact of RAs. The large size of graphs is another source of problems that makes it highly time-consuming to visualize graphs, to simulate execution, or to apply Monte Carlo-style trial and error methods to find patterns in the execution. **What is lacking are light-weight metrics and techniques to analyze locality at a finer scale than the whole graph**.

In this chapter, we study three state-of-the-art RAs: SlashBurn [108], GOrder [163], and Rabbit-Order [6]. We identify different locality types in a parallel graph processing environment. Then we introduce a bespoke technique for each RA to explain how it affects locality. We use real execution metrics (such as execution time, last level cache misses, and DTLB misses), and by introducing a new cache simulation method, Effective Cache Size and novel structural metrics (such as Asymmetricity distribution, Degree Decomposition, and Push and Pull Locality), we compare contrasting effects of RAs on different graphs and present a structural analysis of graph datasets.

In the experiments in this chapter, we use the SkyLakeX machine (Appendix A). As we use different methods in this chapter (simulation, real execution, and calculation of identifying special metrics in the graph datasets), tables and figures start with a word that specifies the method used for extracting the data.

Section 3.2 explains the background materials. We introduce the locality types, the graph-specific cache simulation technique, and the AID metric in Section 3.3. Section 3.4 analyzes the RAs. Section 3.5 demonstrates the structural analysis of datasets.

3.2 Prerequisites

3.2.1 SpMV Graph Traversal

In Section 2.4.1, we explained the SpMV graph traversal that processes all edges of the graph. In this section, we explain more details of SpMV.

Traversal Direction. SpMV can be performed in two directions:

- In the **pull direction**, each vertex aggregates data of its **in-neighbours**. As we see in Algorithm 1, the outer loop (Lines 1-5) traverses vertices and the inner loop (Lines 3-4) traverses all incoming edges to the vertex. In iteration i, the data of a vertex ($\mathbf{D}_v^{i+1}$) is calculated using the vertex data ($\mathbf{D}_u^i$) of its in-neighbours.
- In the **push direction**, vertices update the data of their **out-neighbours**. Algorithm 2 shows the push direction. The outer loop (Lines 3-5) traverses vertices and the inner loop (Lines 4-5) traverses all outgoing edges from a vertex. In iteration i, the data of all out-neighbours of a vertex ($\mathbf{D}_u^{i+1}$) is updated by its data ($\mathbf{D}_v^i$).

Topology and Vertex Data. Based on the adjacency matrix definition (Section 2.1), visiting incoming edges in a pull traversal corresponds to a *column-major* traversal of the adjacency matrix and visiting outgoing edges in a push traversal corresponds to a *row-major* traversal. Consequently, the pull traversal uses the CSC format and the push traversal uses the CSR format.

We use graph average degree ($\frac{|E|}{|V|}$) as the threshold between **low-degree vertices (LDV)** and **high-degree vertices (HDV)**. Vertices with degree greater than $\sqrt{|V|}$ are called **hubs** (borrowed from huge node definition in GOrder). Hubs are divided into in-hubs and out-hubs. A vertex is an **in-hub** if its in-degree (the number of vertices that have edges to that vertex) is greater than $\sqrt{|V|}$ and is an **out-hub** if the out-degree is greater than $\sqrt{|V|}$.

As we saw in Algorithm 1 and Algorithm 2, in addition to topology data, the **data of vertices** (old data: $\mathbf{D}^i$ and the new data: $\mathbf{D}^{i+1}$) are stored in arrays. Each array contains $|V|$ elements and is indexed by a vertex ID.

Parallelisation. In push direction, each vertex updates data of its outgoing edges. Therefore, it is possible that data of a vertex with multiple in-neighbours to be updated concurrently. As a result, it is required to protect data of vertices in the push direction through (1) atomic instructions, (2) buffering (Section 2.4.5), or (3) partitioning edges by

Algorithm 1: Pull SpMV

Input: $G(V, E)$, $\mathbf{D}^i$
Output: $\mathbf{D}^{i+1}$
1 **for** $v \in V$ **do**
2 $\quad sum = 0$;
3 $\quad$ **for** $u \in N_v^-$ **do**
4 $\quad\quad sum$ += $\mathbf{D}^i[u]$;
5 $\quad \mathbf{D}^{i+1}[v] = sum$;

Algorithm 2: Push SpMV

Input: $G(V, E)$, $\mathbf{D}^i$
Output: $\mathbf{D}^{i+1}$
1 **for** $v \in V$ **do**
2 $\quad \mathbf{D}^{i+1}[v]$=0;
3 **for** $v \in V$ **do**
4 $\quad$ **for** $u \in N_v^+$ **do**
5 $\quad\quad \mathbf{D}^{i+1}[u]$+= $\mathbf{D}^i[v]$;

destination (Section 2.4.6).

Pull traversal, on the other hand, does not require protection from concurrent accesses as write memory access for each vertex is performed only once. As a result, pull traversal is faster than push by traversing edges. In this chapter, we use the pull direction SpMV.

3.2.2 Sequential vs Random Memory Accesses

Here, we explain memory access types using the concepts introduced by Zhang, et al. [173]. In SpMV traversal, memory accesses for reading neighbours of a vertex (Line 3 of Algorithm 1 and Line 4 of Algorithm 1) are performed **sequentially** and are accelerated by hardware prefetchers. Moreover, each edge in the *edges* array is accessed only once during a SpMV traversal. So, **accesses to each element of the edges array in the topology data are not repeated** and cache lines containing these elements show **little locality** for a number of consecutive accesses to adjacent elements inside each cacheline.

In Line 4 of Algorithm 1, a memory access is made for reading data of vertex u ($\mathbf{D}^i[u]$) which is an in-neighbour of vertex v. Similarly, in Line 5 of Algorithm 2, a memory access is made for updating data of vertex u ($\mathbf{D}^i[u]$) which is an out-neighbour of vertex v. So, **accessing data of a vertex such as u is repeated** in processing each of its neighbours. In total, SpMV makes $|E|$ accesses to $|V|$ elements of the data array. On average, each vertex data is accessed $\frac{|E|}{|V|}$ times; however, accesses to the data of a vertex are dispersed among $|E|$ accesses. Since $|E| \gg |V|$ and accesses are too unstructured to predict the next accesses accurately using typical modern hardware predictors and are called **random**.

In the **pull** direction each vertex reads the old data ($\mathbf{D}^i$) of its in-neighbours and writes its new data ($\mathbf{D}^{i+1}$). So, **random read memory accesses are made to the old data of vertices**.

In the **push** direction, each vertex updates the new data of its out-neighbours by its old data. So, **random memory accesses are made for writing the new data of vertices**.

3.2.3 Graph Relabeling

In order to accelerate execution by preventing expensive memory accesses, the processor cache tries to store the frequently and/or recently accessed data; however, the skewed degree distribution of large real-world graphs results in a huge number of random accesses that cannot be fully satisfied by the cache (with limited capacity). As a result, it is necessary to improve locality of random accesses to accelerate the graph traversal.

Random accesses to data of vertices are specified by (1) the order in which vertices are processed and (2) edges of vertices. RAs keep the second factor unchangeable and concentrate on the first factor: RAs rearrange the relative order between vertices to change the order of edges and, consequently, the order of random accesses.

As an example, Figure 2.2 shows that vertices 2,5,6, and 8 become vertices 1–4 after relabeling and the vertices with consecutive IDs have more common neighbours that increases the opportunity of reuse in processing vertices (i.e., accessing data of their neighbours).

An RA receives a graph as its input and creates a **relabeling array** of size $|V|$ to permute vertex IDs. The relabeling array is indexed by the old ID of a vertex to specify the new ID. Using the relabeling array, the new CSC/CSR representations are created.

3.2.4 Relabeling Algorithms

This section explains the RAs studied in this chapter.

SlashBurn (SB) [108] considers hubs of the graph as the main connector between vertices and uses this feature to detect communities[1] of vertices by removing hubs and finding connected components that represent communities. This process continues in the next iteration for the giant connected component (**GCC**) - the community with the greatest number of edges. SB assigns consecutive IDs to hubs of the main graph and the giant communities starting from 0 (based on their degrees) and vertices in a community also receive contiguous IDs.

[1] A number of vertices that are tightly connected to each other form a community.

We selected SB as it targets specifically real-world graphs; moreover, it is a representative of degree-ordering RAs. The original implementation of SlashBurn[2] is in MATLAB and we implemented a parallel version of SB in the C language that uses "basic hub-ordering" and selecting $0.02|V|$ vertices (as suggested in the paper) in each iteration.

GOrder (GO) [163] prioritizes neighbours of vertices by defining a "score" function between two vertices:

$$S(u,v) = S_s(u,v) + S_n(u,v).$$

The sibling score ($S_s(u,v)$) is the number of common in-neighbours between u and v, and the neighbourhood score ($S_n(u,v)$) is the number of edges between u and v (that is 0, 1, or 2).

GO starts from the vertex with the maximum degree and uses a sliding window to find the vertex with maximum score (between neighbours of recently assigned IDs) to assign the next ID.

We selected GO because of its special algorithm that concentrates on increasing temporal reuse instead of identifying communities. We used commit `7ccdfe9` of GOrder[3] with its default window size (5). This is a single-threaded implementation for graphs with $|E| < 2^{31}$.

Rabbit-Order (RO) [6] develops communities using neighbours of vertices. By starting from the vertices with the lowest degree, it searches for the neighbour with maximum "gain" that can be reached through merging. The gain function is defined as:

$$\Delta Q_{u,v} = 2\Big(\frac{w_{uv}}{2|V|} - \frac{|N_u||N_v|}{(2|V|)^2}\Big),$$

where w_{uv} is the weight of edge between u and v and N_u is the degree of u. The vertex and its max-gain neighbour are temporarily merged for the purposes of reordering and the weight of the new vertex is calculated as $2w_{uv} + w_{uu} + w_{vv}$. After merging two vertices, the weights of their common edges are also added up. The initial weights of a vertex and an edge are 0 and 1, respectively.

The merging process continues while there is a neighbour u of v with $\Delta Q_{u,v} > 0$; otherwise, the vertex v is added to the top level set which contains the root of communities. Finally, a parallel Depth First Search (DFS) is performed starting from members of the top level set to assign new IDs.

[2] http://datalab.snu.ac.kr/~ukang/SlashBurn-1.0.zip

[3] https://github.com/datourat/Gorder

Dataset	Pre-processing Time (Seconds)			Memory Footprint (GigaBytes)		
	SlashBurn	GOrder	RabbitOrder	SlashBurn	GOrder	RabbitOrder
WebB	232	327	37	35	19	62
TwtrMpi	46	5,697	67	29	23	88
Frndstr	75	4,894	139	38	30	108
SK	90	588	35	36	31	117
WbCc	81	6,587	72	46	33	122
UKDls	810		67	78		236
UU	647		80	105		329
UKDmn	1,040		69	116		394
ClWb9	591			407		

Table 3.1: [Real execution] Preprocessing overheads - Blank cells indicate failed attempts

We selected RO as it is the fastest community detection RA. We used commit `f67a79e` of Rabbit-Order[4]. RO produces different permutations in different executions and we observed that results may vary up to $\pm 5\,\%$. For each graph, one output of RO has been stored and used for all experiments in this chapter. RO did not complete its execution for the `ClWb9` graph because of an "out of memory" error.

3.2.5 Preprocessing Overheads

Table 3.1 shows the preprocessing time (in Seconds) and memory footprint (in Giga-Bytes) of the RAs.

3.3 Locality Types and Metrics

In this section, we introduce different locality types; then, we introduce N2N AID degree distribution and cache miss rate degree distribution that helps to measure different locality types.

3.3.1 Locality Types

Considering random memory accesses in Line 4 of Algorithm 1, the following patterns of reuse of vertex data are identified.

- **Type I**: The consecutive neighbours of vertex v are close so that accesses to the neighbours benefit from **spatial reuse**. This means that proximity of IDs of consec-

[4] https://github.com/araij/rabbit_order

utive neighbours results in placing their data on the same cache line that provides reuse in accessing data of neighbours.

- **Type II**: Subsequently processed vertices v and $v + \delta$ have common neighbours whose data is **temporally reused**. If vertex u is a neighbour of v and $v + \delta$, then proximity of IDs allows cache to reuse the data of u in processing $v + \delta$ after using it for processing v.
- **Type III**: Subsequently processed vertices v and $v + \delta$ have distinct neighbours, but the IDs of the neighbours are close together and causes **spatio-temporal reuse**. If u is a neighbour of v, and $u + \theta$ is a neighbour of $v + \delta$, and θ is small enough that data of u and $u + \theta$ are on the same cache line, then proximity of v and $v + \delta$ results in reuse of this cache line.
- **Types IV and V**: These types happen for reusing a cache line that has been loaded by another thread into a shared cache: a cache line contains data of vertices u and $u + \theta$ and is (re)used in semi-concurrent processing of distinct vertices by **different threads**. It is type IV, if $\theta = 0$ (similar to type II); otherwise, it is type V (similar to type III).

Types IV and V are not directly targeted by RAs as they mainly depend on (1) vertex partitioning algorithm and scheduling method of the runtime environment, and (2) availability of the shared caches in the processor architecture. Types I, II and III are determined by the graph and are controlled by RAs.

GO aims for improving type II and III by selecting the vertex with maximum gain based on current contents of cache. RO targets type I and tries to improve clustering based on neighbourhood of vertices that also results in type III. SB aims to improve locality types I and III by identifying communities, and types II and III by assigning consecutive IDs to hubs.

3.3.2 Neighbour to Neighbour Average ID Distance

Community detection algorithms such as RO try to form clusters based on the neighbourhood of vertices. By assigning consecutive IDs to vertices in the same community, they aim to increase reuse of neighbours' data. To investigate how an RA succeeds in **bringing neighbours close to each other (spatial locality, type I)**, we calculate the distance between neighbours.

Using $N_{v,i}$ to show the ID of the i-th neighbour of vertex v (with sorted neighbours

list in ascending order), **Neighbour to Neighbour Average ID Distance (AID)** is defined as:

$$AID_v = \frac{\sum_{i=2}^{i=|N_v|} |N_{v,i} - N_{v,i-1}|}{|N_v|}$$

When an RA assigns close IDs to neighbours of a vertex, the difference between IDs of consecutive neighbours is reduced and AID is reduced. In this way, **lower AID values, generally, relate to better spatial locality**. For a SpMV in the pull direction, AID considers only the in-neighbours of a vertex.

We study the effects of RO on spatial locality of different vertex classes, using AID degree distribution in Section 3.4.3 (Figure 3.3). AID degree distribution is computed in $\mathcal{O}(|E|)$ time and $\mathcal{O}(max\text{-}degree)$ space complexity.

It is useful to compare N2N AID to "average gap profile" [11] that calculates average of the differences between the IDs of two endpoints of each edge to provide a summary of the spatial locality of the graph. The neighbours of a vertex do not need to be close to the main vertex as they should be only close to each other to maximize spatial locality.

It is important to note that AID measures clustering efficacy of an RA and is independent of the architecture. In this way, **AID is not a deterministic spatial locality metric**. As an example, assume a vertex has neighbours with IDs 1600, 3200, and 6400. If an RA changes the IDs of neighbours to 400, 800, and 1600, AID is reduced but the spatial locality is not changed (as the neighbours are still on different cache lines).

3.3.3 Cache Miss Rate Degree Distribution

In order to collect detailed information about RAs, we collect cache miss rates based on the degree of vertices. This shows how RAs affect locality types II and III of different vertex classes. We use simulation for this purpose; however, detailed simulation of processor and memory hierarchy (in simulators like Gem5 [20]) is time-consuming for large graphs.

Since (i) graph analytics are memory intensive (as an example, in Algorithm 1 we have one add as computation in Line 4), and (ii) computation instructions are much faster than memory instructions, we ignore simulating execution of instructions except time-consuming memory instructions (load and store instructions) to make the simulation process efficient and fast.

We designed a trace-based simulator based on the cache simulator of SimpleScalar[5] [33] and equipped it with an accurate implementation of the dueling BRRIP and SRRIP [83] cache replacement policies. We use this implementation for level 3 cache shared between the cores of each NUMA node and for the same configuration (number of sets and ways of associativity) as the real CPU, our SkyLakeX machine (Appendix A). We instrumented Algorithm 1 at source code level to call the simulator for every load/store.

We performed the parallel simulation in two phases: (1) logging memory accesses during graph processing by each of the parallel threads, and (2) dividing execution duration between threads where for each interval a thread simulates all logged accesses by parallel threads in a round robin way.

Figure 3.1 shows the degree distribution of cache miss rate for RAs. We will interpret these results in Section 3.4.

For datasets used in this chapter the average simulation time of one SpMV iteration is 151 seconds. Compared to the real machine, the total cache misses of the simulation has an average 15% error, and the average relative error (for comparing misses between two RAs) is 1.4%. This means that differences greater than 1.4% between miss rates of relabeled versions of a graph in Figure 3.1 are valid (presuming all degrees experience the same rate of error).

3.4 Locality Analysis of RAs

3.4.1 SlashBurn

SB has been designed for power-law graphs: *"We propose to envision graphs as a collection of hubs connecting spokes, with super-hubs connecting the hubs, and so on, recursively"* [108]. The main idea is to iteratively remove hubs of power-law graphs; however, the practicality of this method depends on (i) whether power-law graphs are destroyed recursively and also (ii) if hubs are the main communication points of the graph.

To assess the first hypothesis, we depict the degree distribution of GCC for different iterations of SB in Figure 3.2. **Over different iterations of SB, the degree distribution of the GCC does not maintain the power-law property**. After a few iterations, the remaining network shows an almost-uniform degree distribution with low degrees. Further iterations of the SB separate these LDV from their neighbours in what are per-

[5] http://simplescalar.com/

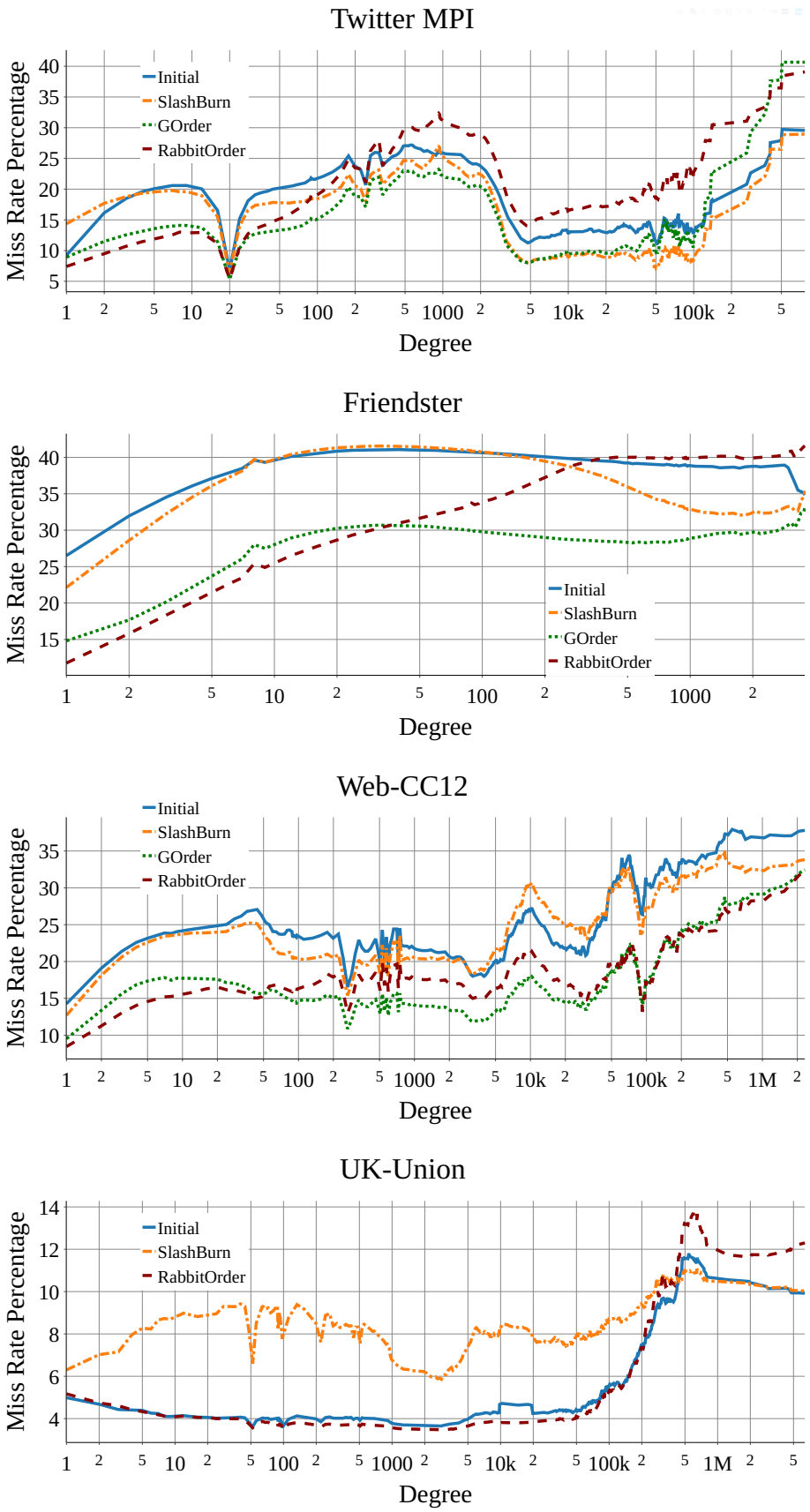


Figure 3.1: [Simulation] Cache miss rate degree distribution

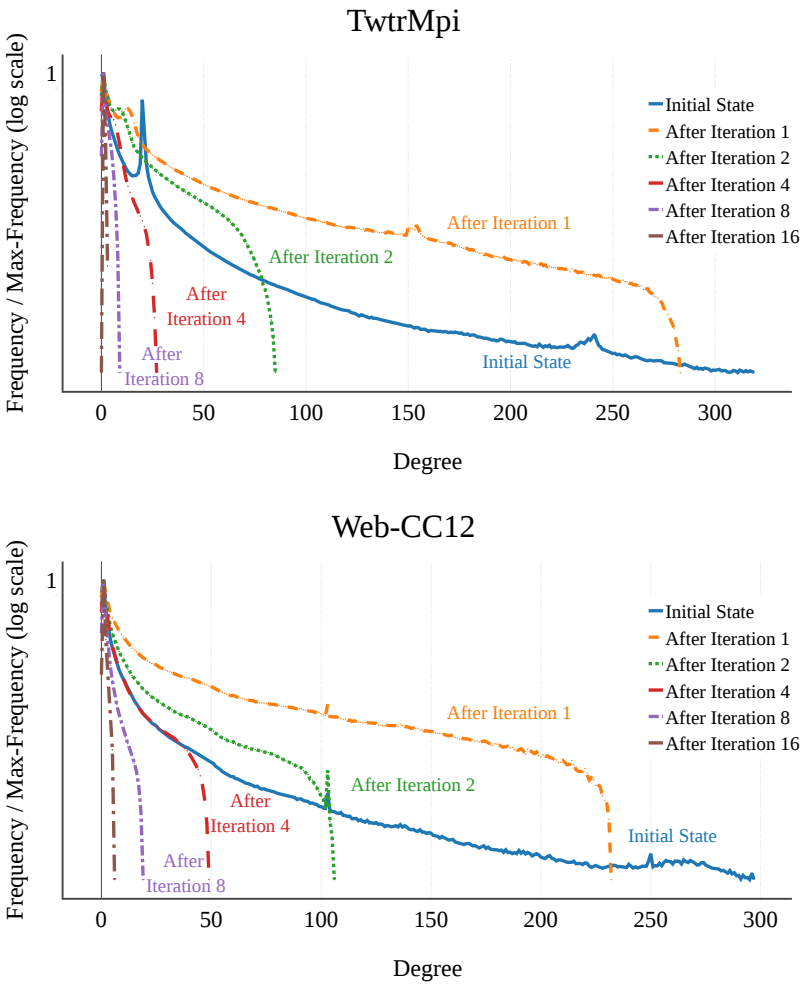


Figure 3.2: [Real execution] Degree distribution of initial graph and GCC after SB iterations

ceived as different communities. As a result, neighbours are assigned widely distinct IDs that reduces locality types I and III.

To evaluate the second hypothesis, at least for some graphs like protein-protein interaction networks, it has been shown that **flow bottlenecks** (vertices that frequently appear in communication paths between different vertices, i.e., with greatest centrality) may not be the same vertices as hubs [170]. This shows that flow bottlenecks are better candidates to produce communities by being removed from the graph.

SB is partly similar to **degree-ordering** as a number of HDV receive initial consecutive IDs that increases temporal reuse (types II and III) in accessing data of out-hubs. SB improves locality types IV and V (Section 3.4.6).

3.4.2 GOrder

GO tries to optimize locality by maximising reuse of the current content of the cache (types II and III). It uses a sliding window and searches for a neighbour with the greatest score (Section 3.2.4). For a HDV in the sliding window the sibling score is dominant and the vertex with more common in-neighbours will have more chance to be selected. For a LDV in the sliding window the neighbourhood score is dominant.

GO considers common neighbours with only a limited number of already-placed vertices (a window size of the past 5 vertices). There are numerous LDV in power-law graphs and many of them appear equally "close" to the 5 last labeled vertices. As such, GOrder cannot properly distinguish which LDV to select. This is reflected in the cache miss rate (Figure 3.1) where **GOrder decreases cache miss rate well on HDV but cannot perform well for LDV**.

To further investigate GOrder's strategy towards HDV, we use cache simulation to count the number of misses occurring in accessing data of HDV. Table 3.2 shows that GO and SB have the lowest reloads of HDV. For `Twitter MPI` and `Friendster` SB has lower reloads of vertices with $degree > 2000$; but, for vertices with $degree > 20$, GO has the lower reloads. As such, **GOrder increases the number of reloads of HDV to provide space in cache for LDV (to reduce its reloads). The latter are exponentially more frequent in power-law graphs**.

Dataset	Min. Degree	Initial	SlashBurn	GOrder	Rabbit-Order
WebB	2,000	10	21	2	10
TwtrMpi	2,000	8	0.4	4	11
TwtrMpi	20	345	260	230	377
Frndstr	2,000	2	0.04	1	3
Frndstr	20	1,177	1,110	818	1,060
SK	100	8	26	7	11

Table 3.2: [Simulation] Total number of misses (in millions) for accessing data of vertices with $degree > Min.\ Degree$

As we explained in Section 3.4.1, **partial degree-ordering in SB keeps data of out-hubs in cache**; but, the score function of GO selects vertices with more temporal reuse based on the current contents of the cache and prevents filling cache with HDV. In other words, GO allocates cache space to vertices with lower degree but with more temporal

reuse in short durations of processing and in this way, **GOrder reduces the presence of HDV in the cache to increase the total reuse**. As a consequence, GO fills the cache with vertices of different degrees (but with more temporal reuse) rather than dedicating the cache capacity to vertices with the highest degree.

3.4.3 Rabbit-Order

RO builds communities bottom-up and starts from low degree vertices and merges neighbouring vertices while trying to maximise the gain function (Section 3.2.4). This results in a set of trees that reflect the communities and are used in the second phase to assign IDs by DFS traversal of each tree.

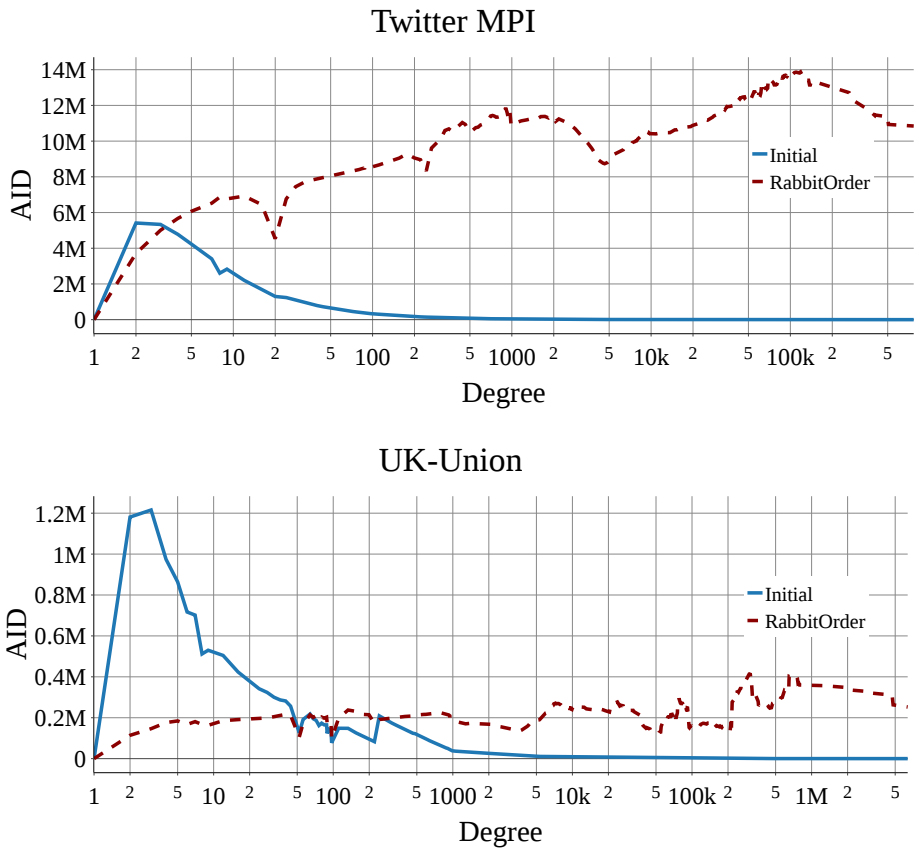


Figure 3.3: [Calculation] AID degree distribution

We use degree distribution of AID (Section 3.3.2) to assess changes made by RO in spatial locality. As Figures 3.1 and 3.3 show, **Rabbit-Order reduces AID of LDV and improves their spatial locality** by using DFS in the second phase that assigns spatially close IDs between neighbouring LDV in clusters. However, as degree of vertices is

increased, DFS cannot assign consecutive IDs to the neighbours (because each neighbour has itself a number of neighbours). So, **AID and cache miss rate of Rabbit-Order are increased for HDV**.

3.4.4 Observation on Hubs

Figure 3.1 shows that all RAs incur higher miss rates for hub vertices. Processing an in-hub requires accessing data of several in-neighbours, and only a fraction of that data can be held in cache in any one moment. For other ones, memory accesses are required. While RAs change the order of edges of hubs, they cannot change the topology of a graph. So, **locality of hubs is not improved by RAs as much as other vertices**.

Locality of hubs is important as they involve a large fraction of the edges (therefore, a large fraction of traversal time) and this observation demonstrates that **hubs of skewed-degree graphs suffer from a structural problem in relation to locality that cannot be solved by RAs**.

Moreover, since the data of subsequent neighbours of an in-hub should be read from memory and be placed in cache, the formerly read neighbour's data is flushed by the newer ones before that in-hub is processed completely. This does not allow the data to have an opportunity to be used in processing other vertices. In this way, **reuse of cache contents is reduced as a side effect of processing in-hubs**.

3.4.5 Real Execution Performance Metrics

Table 3.3 shows the real execution of SpMV. The number of misses and idle time are averaged between threads.

Last level (L3) cache misses show the number of memory accesses that are not satisfied by caches and are sent to main memory. The number of L3 misses is the main locality metric. Table 3.3 shows that SB may destroy locality and, therefore, increase the execution time. GO reduces L3 misses and execution time of social networks. RO reduces the cache misses and the execution times of web graphs.

DTLB misses is the number of misses that occur in looking up translations of virtual addresses to physical addresses. While a DTLB miss results in (possibly multiple) memory accesses, DTLB misses are not usually a bottleneck as the total size of huge memory pages that are cached by TLB is much greater than the aggregate CPU cache capacity. DTLB misses show locality of RA at larger granularity, i.e., at longer reuse distances

Dataset	**Time** (MilliSeconds)				**Idle** (%)				**L3 Misses** (Millions)				**DTLB Misses** (Kilos)			
	Bl	SB	GO	RO	Bl	SB	GO	RO	Bl	SB	GO	RO	Bl	SB	GO	RO
WebB	90	145	89	79	1.5	2.1	2.2	2.3	4.3	6.8	4.3	3.7	0.6	1.7	1.8	1.6
TwtrMpi	354	339	299	366	1.8	2	1.1	1.7	15.7	14.2	12.6	16.3	4.7	2.3	3.1	3.1
Frndstr	771	761	578	667	1.2	1.5	1.4	1.2	40.8	39.2	29.1	34.9	9.3	9.4	7.1	7.6
SK	117	168	109	109	8.2	1.5	1.6	4.1	5.7	8.8	5.5	5.3	0.8	1.4	0.5	0.6
WbCc	438	414	311	297	1.9	2.3	2.3	3.1	20.5	19.3	13.5	12.6	8.6	6.8	6.9	4.5
UKDls	194	317		180	1.9	1.9		2.5	10.1	16.5		9.3	1.8	4.4		1.4
UU	282	486		285	1.9	1.9		6	14.6	24.9		13.8	2.8	7.8		2.4
UKDmn	297	459		281	1.4	2.1		2.7	15.7	23.5		14.7	4.4	5.6		2.7
ClWb9	2,221	2,811			1.3	1.4			100.9	139.3			39M	181		

Table 3.3: [Real execution] SpMV execution results (Bl: Baseline without relabeling)

than L3 misses.

RO interleaves HDV between LDV during the ID assignment phase. By applying DFS on independent clusters whose data are placed in a few memory pages, **Rabbit-Order minimizes intra-cluster edges that reduces DTLB misses**.

Idle time shows the average percentage of execution time that each thread is idle. Comparison between RO and the baseline for `UU` in Table 3.3 shows that RO reduces L3 misses, but the execution time is not better than the baseline. Increased idle time is one of the reasons and shows that **improving locality does not necessarily translate to improved performance**.

Since RAs do not change the locality of consecutive vertices (that form partitions that are assigned to or stolen by threads) evenly, as Table 3.3 shows, **improving locality of a graph dataset by an RA may increase the idle time**. Increased idle time indicates poorer load balance and opens further opportunities for optimization, e.g., it may be possible to further improve load balance by increasing the number of partitions.

3.4.6 How Much of Cache Capacity Is "Effectively" Used?

We introduce the **Effective Cache Size (ECS)** as "the percentage of cache capacity dedicated to caching randomly accessed data". In the pull direction SpMV (Algorithm 1), this measures the proportion of cache used to cache $\mathbf{D}^i$. The ECS is important in graph processing as cache lines of topology data are sequentially accessed and have a limited reuse. So, there is no merit in keeping topology data in cache; but, randomly accessed vertex data are reused and dedicating more cache space to them is beneficial to improve

performance.

We use functional (timing-less) simulation to estimate ECS. We periodically scan the cache contents during execution to identify cache lines containing old data of vertices. Table 3.4 shows the results for the baseline (without relabeling) and for different relabeling algorithms. Table 3.4 shows that **RAs do not utilize all capacity of the cache to satisfy random memory accesses**.

Dataset	Baseline	SlashBurn	GOrder	Rabbit-Order
WebB	27.4	50.9	26.2	20.4
TwtrMpi	68.3	65.5	63.4	68.9
Frndstr	77.5	76.9	75.2	76.7
SK	37.3	55.2	37.3	42.9
WbCc	64.1	64.9	57.5	58.9
UKDls	22.0	48.1		20.6
UU	29.7	52.4		30.6
UKDmn	18.2	41.5		18.3

Table 3.4: [Simulation] Average effective cache size (%)

Moreover, SB usually has the greatest ECS while it makes the most cache misses (Figure 3.1 and Table 3.3). In other words, **by reducing locality, the effective cache size is increased**. To explain this, we need to review the arrangement of vertices in the SB algorithm. By separating LDV from their parents in the last iterations of SB, the locality types I and III of LDV are reduced. This means more memory requests are performed and cache lines with lower reuse are evicted faster (as a greater number of new cache lines are fetched from the main memory and should be placed somewhere in the cache). Therefore, **cache lines of topology data are removed faster from cache** and the number of cache lines of vertex data is increased.

To have a better illustration, we compare this status to when all random memory accesses are clustered on a small number of vertex data because of better locality. So, a smaller portion of cache capacity is dedicated to those frequently accessed vertices and ECS is reduced. Comparison of L3 misses in Table 3.3 and ECS in Table 3.4 also shows that **the RA with the best locality for a dataset, has the lowest ECS**.

This observation has two important repercussions for hardware design: (i) the full cache capacity remains unused in the current state of the art. So, improving locality will mean **caches are even more over-sized**, and (ii) in order to optimize a multi-level caching system, we need to have **new and different cache inclusion policies for memory access**

types: the topology data (that is prefetched) is required to be cached only in the highest level, but the vertex data benefits from multi-level caching, especially from a shared last level cache where only vertex data cache lines are stored and accessed by multiple processors.

On the other hand, this shows that **we need software algorithms that are capable of making good use of all capacity of the cache**.

Increasing ECS in SB results in filling the cache with a large number of vertex data and **locality types IV and V are improved in processing numerous neighbours of hubs**. So, as Figure 3.1 shows, **the miss rate in processing hubs is reduced by SlashBurn**.

3.5 Locality Analysis of Graph Datasets

This section investigates the structure of different types of real-world graph datasets and their effects on RAs.

3.5.1 Web Graphs vs. Social Networks

Table 3.3 shows that the RA that performs well for social networks is GO and for web graphs it is RO. Section 3.4.2 explains that GO improves locality of HDV and Section 3.4.3 demonstrates how RO improves locality of LDV. So, HDV of social networks and LDV of web graphs are the main sources of improving locality by GO and RO, respectively.

To explain this, we compare the connection between HDV in social networks and web graphs by defining the **Asymmetricity** of a vertex as the fraction of in-neighbours that are not out-neighbours:

$$Asymmetricity_{(v)} = \frac{|\{(u,v) \in E | (v,u) \notin E\}|}{|\{(u,v) \in E\}|}$$

Figure 3.4 compares the degree distribution of asymmetricity of `TwtrMpi` (a social network) to `UK-Union` (a web graph). It shows that `TwtrMpi` has highly symmetric vertices with high in-degrees. In other words, **in-hubs are almost symmetric in social networks (in-hubs are out-hubs), while web graphs do not have symmetric in-hubs**.

For further investigation, we analyze the connection between vertices by defining degree classes: "1-10", "10-100", "100-1K", ... Figure 3.5 represents the **degree range decomposition** as the correlation between the degrees of neighbouring vertices: all edges

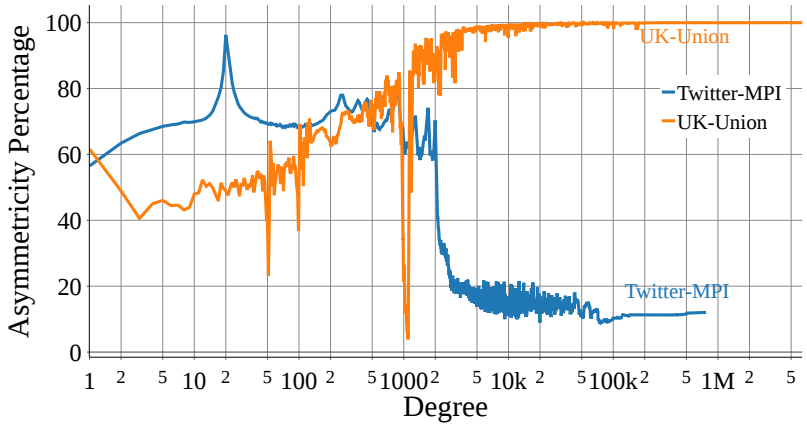


Figure 3.4: [Calculation] Asymmetricity degree distribution

to vertices in a degree class are binned based on the degree class of their source vertex. As an example, vertices with in-degree between 10-100 in `TwtrMpi` receive 29% of their incoming edges from vertices with out-degree 100K-1M.

For vertices with degree greater than $1K$ in `TwtrMpi`, HDV form more than half of the neighbours while, in `SK-Domain` LDV are dominant in forming neighbours of HDV. This shows that **HDV have close connection to each other in social networks**. On the other hand, **LDV are the main constituents of all degree classes of the web graphs**.

For this tight connection of HDV in social networks, RO cannot form independent clusters (with relatively small number of intra-cluster edges) and therefore RO cannot improve locality of the HDV. Table 3.2 also shows that RO has the most reloads, but GO manages hubs based on their temporal reuse (Section 3.2.4). In this way, **GOrder optimizes reuse of a large number of fully connected HDV of social networks** that cannot be kept simultaneously in the cache by giving priority to temporal reuse of vertices with lower degree.

On the other hand, web graphs do not have a tight connection between HDV and the important factor for locality is spatial locality between low-degree neighbours. As a result, **Rabbit-Order efficiently groups LDV to reduce AID and improves their locality** (Figures 3.1 and 3.3).

3.5.2 Push Locality vs. Pull Locality

Section 3.2.1 explained push and pull traversal directions. In this section we explain how different datasets benefit from a special traversal direction.

Push and pull traversals differ in two aspects: (1) using CSC in pull and CSR in push,

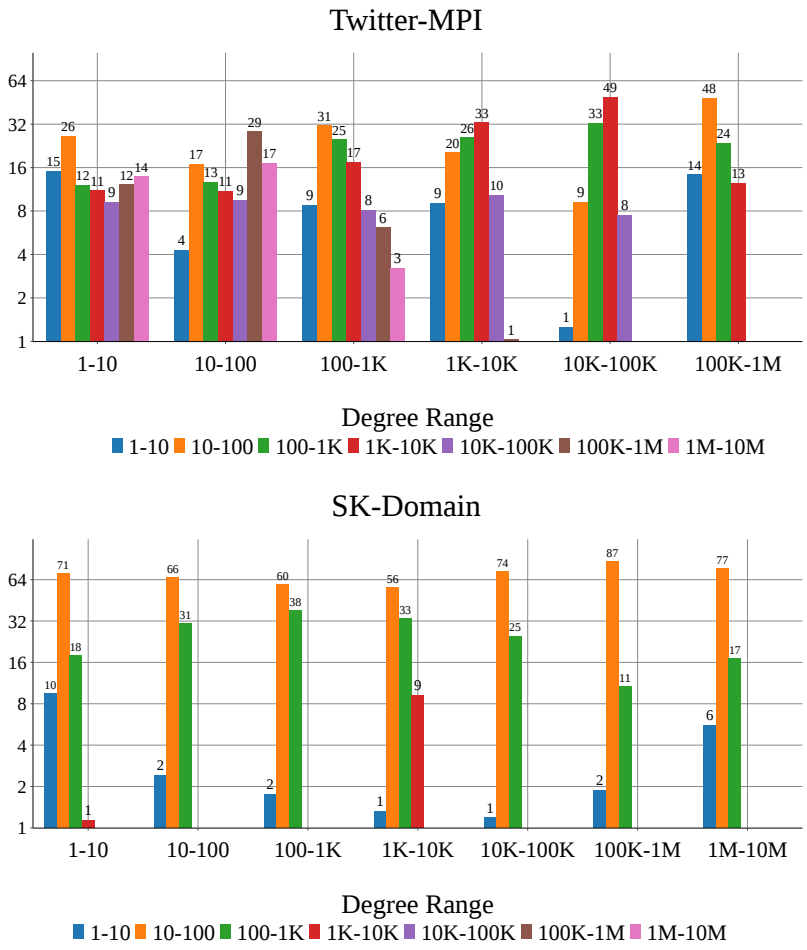


Figure 3.5: [Calculation] Degree range decomposition of neighbours of vertices (in percent)

and (2) reading data of vertices in pull and writing it in push. So, the comparison of push and pull traversals should be performed in two steps: (1) investigating the impact of CSC and CSR formats of the graph by considering the **same operation** (e.g. read) for both formats (instead of read in CSC and write in CSR), and (2) identifying how read and write instructions affect the CSC and CSR traversals.

The second step depends on the analytic algorithm. So, we concentrate on the first step to understand the impacts of different real-world graphs with skewed degree distribution on locality of push and pull traversals. Table 3.5 compares CSC and CSR traversals for the **read operation**, i.e., each vertex makes a sum of data of its in-neighbours (in CSC traversal) and its out-neighbours (in CSR traversal). It shows that there is a fundamental difference: **web graphs have faster CSR traversal, but CSC traversal is faster for social networks**.

Dataset	L3 Misses (M)		Traversal Time (ms)	
	CSC	CSR	CSC	CSR
WebB	4.3	3.8	90	81
TwtrMpi	15,7	21.7	354	439
SK	5.7	4.6	117	88
UKDls	10.1	9.3	194	177
ClWb9	100.9	96.5	2,221	2,129

Table 3.5: [Real execution] CSC vs. CSR read traversals

To explain the differences in CSR and CSC locality, we study the structure of power-law graphs. The effect of hubs becomes more important in CSR and CSC traversals by noting two points discussed in Section 3.5.1: (1) real-world graphs with skewed degree distribution may have both in-hubs and out-hubs or only one of them, and (2) in-hubs are not always out-hubs and vice versa. Moreover, in a **pull** traversal using CSC format, **out-hubs have a constructive effect** on locality as their data is frequently accessed and is **reused** in processing several vertices; but, in a **push** traversal using CSR **in-hubs are locality improving**.

In order to explain locality of push and pull traversals, we consider the number of edges where accessing their data results in a cache hit by keeping H hubs with maximum degrees in the cache. This shows what fraction of the total edges (as an indicator of total random memory accesses) are processed with no cache miss as a result of keeping data of H hubs in the cache. Figure 3.6 illustrates the percentage of edges covered by hubs while increasing the number of hubs for a social network (`Twitter MPI`) and a web graph (`SK-Domain`).

Figure 3.6 shows that in a pull traversal of `Twitter MPI`, accesses to data of 44% of the edges are hit in cache if we keep $100K$ out-hub data in the cache, but in a push traversal only 23% of the edges are covered. For the `SK-Domain` it is vice versa: a pull traversal covers only 4% of the edges for keeping $100K$ out-hub data in the cache while, push traversal covers 64% of the edges. We found the same trend across all graphs of the same types.

This shows that **web graphs benefit from Push Locality as they have more powerful in-hubs than out-hubs while, social networks benefit from Pull Locality because of their more powerful out-hubs**.

This difference between push and pull locality in social networks and web graphs is reflected in the applicability of Direction-Optimizing BFS (DO-BFS) for social net-

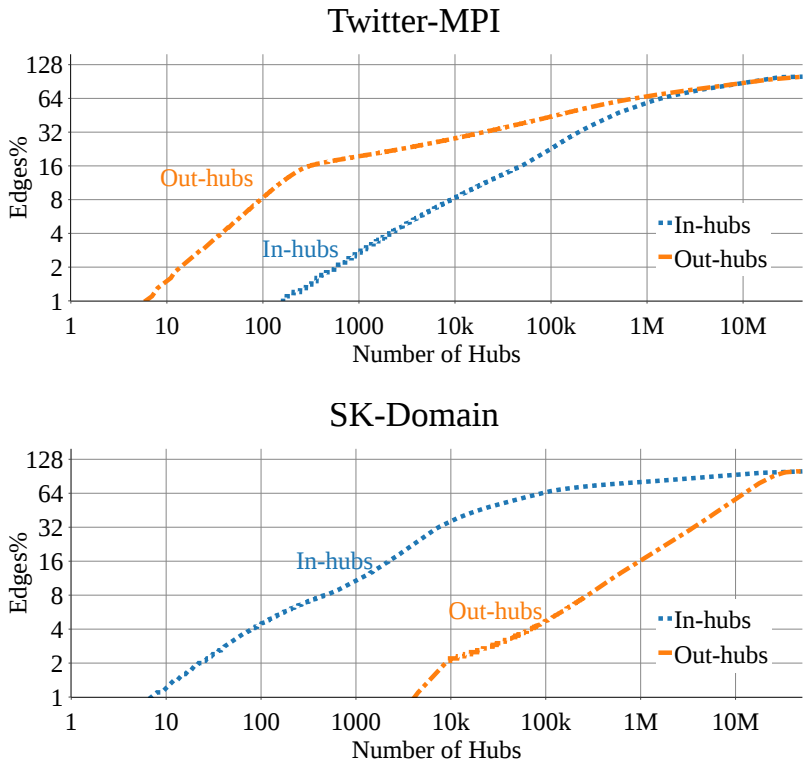


Figure 3.6: [Calculation] Comparison of percentage of edges covered by in-hubs in CSR traversal vs. out-hubs in CSC traversal

works [13]. DO-BFS uses push traversal for sparse iterations (where a small portion of vertices are active) and uses pull traversal for dense iterations (where a large number of vertices are active) as suggested by [151]; however, pull traversal is accelerated only for graph datasets like social networks that have pull locality. For web graphs, on the other hand, pull traversal is not supported by pull traversal as they have push locality.

3.6 Conclusion

In this chapter, we discussed key mechanisms that create or destroy locality. We introduced a number of techniques to analyze graph reordering algorithms (RA) and their effects on real-world graphs with skewed degree distribution. We introduced Locality Types to enrich the terminology required for the discussion and we presented an accurate graph-specific simulation technique that allowed us to investigate locality conditional on the degree of vertices. We presented N2N AID as a spatial locality metric. Using these techniques and metrics we studied three state-of-the-art locality optimizing RAs: SlashBurn, GOrder, and Rabbit-Order to identify how they affect locality of

different vertices.

We presented a structural analysis of real-world graphs that explains the contrasting behaviours of datasets in relation to RAs. We identified a tight network of high-degree vertices in social networks that suffers from temporal locality and we discussed the functionality of GOrder that enhances temporal locality of these datasets. Analysis of web graphs showed that low-degree vertices are the dominant class of vertices. As a result, clustering low-degree vertices of web graphs by Rabbit-Order improves the spatial locality. We introduced Effective Cache Size as a metric of cache capacity utilization and we see that improving locality by relabeling algorithm reduces the ECS.

We considered differences of the locality of push and pull traversals as consequences of the structure of datasets and we showed that web graphs benefit from push locality but, social networks benefit from pull locality. This reveals the necessity of considering the structure of datasets in selecting a suitable direction for processing and also in interpreting results.

We observed that while hubs dedicate a large portion of edges of a graph, they experience low locality in graph traversal even after relabeling by different graph reordering algorithms. This shows that different vertex classes in power-law graphs have different performance and locality demands that are required to be considered in designing high-performance algorithms.

Chapter 4

Uniform Memory Demands Strategy

4.1 Introduction

In Chapter 3, we analyzed the structure of graph datasets and we observed that relabeling algorithms cannot improve locality of hub vertices as much as other vertices and this negatively affects the performance as hubs dedicate a great percentage of edges (and execution time) to themselves.

We explained that graph relabeling has inherent limitations in improving locality as it changes the order of vertices and deploying reordering algorithms cannot improve locality for vertices with great degrees as cache cannot store the data of all their neighbours. In other words, while reordering algorithms consider the structure of graphs to renumber the vertices, the processing of vertices, still, ignores the structural differences of vertex classes and applies the same structure-oblivious traversal for all vertices.

This shows that **a unique traversal cannot satisfy contrasting memory demands of different subgraphs**, even in a simple graph traversal such as SpMV. We need to differentiate between these subgraphs to prevent inefficient processing of subgraphs by applying a traversal to the whole graph while it is only suitable for a particular subgraph. In other words, we need to consider the implications of the **graph structure** on graph traversals in order to satisfy the memory demands with the lowest overhead as a result of deploying subgraph-optimized graph traversals.

To that end, in this chapter, we introduce the **Uniform Memory Demands** strategy that acknowledges the diverse behaviours and demands of different vertex classes and subgraphs instead of trying to find a general solution for the whole graph.

The Uniform Memory Demands strategy operates by **categorizing memory be-**

haviours of different subgraphs into groups, where, in each group, memory accesses produce the same behaviour. Then, distinct data structures and algorithms are designed for each vertex class/subgraph aiming to satisfy the uniform memory demands in their vertex class/subgraph with the lowest overhead.

As the Uniform Memory Demands strategy concentrates on optimizing the *performance* of graph algorithms, we consider the execution of graph algorithms in the computing environment[1]. As a result, it is necessary to analyze **the mutual implications of (i) the computer architecture, (ii) the graph structure, and (iii) different graph algorithms as solutions for a graph problem**. This analysis helps us to identify or to design the best graph algorithm for each subgraph that satisfies its memory behaviours with the lowest overhead.

After identifying/designing the algorithms that best match the memory demands of subgraphs/vertex classes, we may need to apply some changes in the graph structure. For example, we may need to separate edges of each subgraph and to store them together in order to facilitate consecutive accesses to these edges. As a result, a restructuring step (as a pre-processing step) may be required. Section 4.2 details the steps of applying the Uniform Memory Demands strategy to design structure-aware graph algorithms.

It is also helpful to compare the Uniform Memory Demands strategy to strategies with similar concepts. The "Divide and Conquer" and "Dynamic Programming" strategies try to find a solution for a problem by dividing the problem into smaller problems that are easier to solve, or can be further divided, recursively. In contrast, the Uniform Memory Demands strategy (i) states that a general solution is not capable of satisfying contrasting memory demands of different subgraphs/vertex classes in real-world graphs and (ii) finds bespoke solutions with optimized performance for each subgraph.

From another perspective, in applying the Uniform Memory Demands strategy, it is not the problem to find an algorithm as a solution for a graph problem. But, **we are curious about the formation of different subgraphs in the structure of the graph and**

[1] As we detail in Section 8.2, this dependency of considering the execution of graph algorithms in computing environments restricts the domain of the work and the direct applicability of the results to the considered computing environments. However, it is the requirement of the work as memory behaviours are created only in an execution environment and based on the architecture of the processor. Therefore, this dependency is the **virtue** of the work and not a **limitation**. Moreover, while a study may consider a restricted domain, it may result in more general falsifiable insights and techniques that are not limited to that domain.

how these subgraphs express different memory demands in different graph traversals as solutions to a particular problem.

Although, as the Uniform Memory Demands strategy divides the input graph dataset based on memory access demands in order to provide better performance, we can say that the Uniform Memory Demands strategy applies the Divide and Conquer strategy on the structure of the input dataset to find the best algorithm for each subgraph.

In the following sections of this chapter we explain the Uniform Memory Demands strategy and its applications in more detail. In Section 4.2 we explain the steps of the Uniform Memory Demands strategy for designing structure-aware graph algorithms. In Section 4.3, we summarize the applications of the Uniform Memory Demands strategy in designing structure-aware graph algorithms in the following chapters of this thesis. We discuss further applications of the Uniform Memory Demands strategy in Section 4.4.

4.2 Analysis and Design Steps

In designing algorithms based on the Uniform Memory Demands strategy, we consider the following steps:

Step 1. Identifying Contrasting Demands & Behaviours

As the first step, we have to identify if it is observed/expected from different vertex classes/subgraphs of power-law graphs to present contrasting behaviours. We need to know the immediate results of these behaviours and how they affect performance.

In this step, we answer these questions: **What are the opposing behaviours of vertex classes or subgraphs?** What is the main bottleneck in the main algorithm (poor memory locality, work inefficiency, or load imbalance)? What features of the input datasets may change the performance of the main algorithm? Can we extract or imagine vertex classes or subgraphs with different behaviours?

Analysis techniques, such as the ones introduced in Chapter 3, that evaluate a graph for its smaller constituents (such as vertex classes, component, and subgraphs) are useful in this step.

Step 2. Considering Potential Solutions

In the second step, we consider different solutions for performing the main task. We need to understand the effects of each solution and algorithm and its requirements.

The questions in this step are: **What are the possible algorithms/traversals for performing the main algorithm?** What are the special features, advantages, and disadvantages of each algorithm? Do particular conditions exist which maximize the performance of an algorithm? What are the effects of these algorithms on different vertex classes and/or subgraphs?

Step 3. Matching & Adjusting

Now, we have to specify which algorithm facilitates the best performance for each vertex class/subgraph, and what changes are required to adjust the matched algorithms in order to provide the same results as the main task.

In this step, we need to find answers for these questions: **What traversal/algorithm is the best for each vertex class/subgraph?** What are the direct and subsidiary results of selecting a potential solution for a vertex class? How does it provide better performance? Does it affect other metrics (like load balance, memory locality, and work-efficiency)?

Step 4. Merging

In the last step, we have to specify if we need to perform other modifications to make it possible that different algorithms of different vertex classes/subgraphs work together.

Questions that we need to answer are: **How to merge the distinct algorithms that work on distinct vertex class/subgraph to deliver compatible results with the main task?** Is it necessary to modify results of any of the algorithms (that work on a subgraph)? Is it required to apply changes into the structure of the graph to allow algorithms to work independently? Can these distinct algorithms work concurrently? If not, does sequential processing of subgraphs have side effects on load-balance or performance?

4.3 Applications

In the rest of this thesis, we design the following three structure-aware graph algorithms by deploying the Uniform Memory Demands strategy and the design steps explained in Section 4.2:

- In Chapter 5, we consider different parallel counting sort algorithms applied to a set of key-value pairs as input data with a skewed degree distribution for the values. This condition happens for degree ordering of real-world graphs with power-law degree distribution.

  For parallel counting sort with shared variables between threads, atomic memory accesses are the main cause of inefficiency. On the other hand, if we use private variables, the size of memory and the overhead of merging are the causes of inefficiency. This has made parallel counting sort practically slower than comparison-based sorting algorithms.

  By applying the Uniform Memory Demands strategy, we explain that shared variables are suitable for a subset of vertices and the private variables are the best option for the rest of vertices. In this way, we introduce the **SAPCo Sort** algorithm that optimizes performance in degree ordering of power-law graphs by considering the structure of the input dataset in order to reduce memory space requirement while increasing work-efficiency.

- In Chapter 6, we consider the SpMV-based graph algorithms and we use the Uniform Memory Demands strategy to explain that the skewed degree distribution of graphs prevents processing of hub edges from benefiting from CPU's cache to have better locality.

  If we process the edges in the pull direction, the great degree of in-hubs does not allow reuse of fetched data from memory. On the other hand, processing in the push direction suffers from (i) the overhead of the atomic memory accesses or (ii) the memory requirement and performance overhead of buffer merging.

  By using the Uniform Memory Demands strategy, we design the **iHTL** algorithm as a structure-aware SpMV. iHTL processes the incoming edges to in-hubs in the push direction as these edges have a small number of destinations but numerous sources. So, it is better to dedicate cache to the destinations (whose small numbers allow them to be maintained in the cache) and benefit from reusing of destinations'

data as a result of the push-direction traversal. On the other hand, for incoming edges to non-hubs, iHTL uses the pull direction that facilitates reuse of source vertices of the edges.

- In Chapter 7, we analyze the memory accesses in the Triangle Counting (TC) algorithm when running on real-world graphs with power-law degree distribution.

  We use the Uniform Memory Demands strategy to show that the Forward algorithm, as a structure-oblivious TC algorithm, results in (i) depriving non-hub vertices from experiencing locality in accessing their neighbour lists, (ii) inefficient usage of cache capacity, and (iii) fruitless searches.

  Then, we introduce the **LOTUS** algorithm that optimizes locality by dividing the graph into three subgraphs and by performing the triangle counting in 3 steps. In each step, Lotus uses a compact data structure and a bespoke algorithm in order to optimize memory locality.

4.4 Discussion

In this thesis, we explain the applications of the Uniform Memory Demands strategy for designing graph algorithms with optimized performance in processing skewed-degree graphs. However, this strategy is also useful for other graph structures with other degree distributions as long as the degree distribution results in contrasting memory behaviours for different vertex classes or subgraphs.

The main ideas behind the Uniform Memory Demands strategy are to (i) recognize contrasting memory behaviours produced by different subsets of the input datasets and to (ii) categorize these demands in order to find the best solution for each category. So, the Uniform Memory Demands strategy works based on two assumptions: (a) existence of contrasting behaviours and (b) separation of these contrasting behaviours results in satisfying these demands with lower overhead by designing and deploying specific data structures and algorithms for each group of behaviours. The first assumption depends on the structure of the graph dataset and the second assumption is, mainly, a data and task management technique whose efficacy is identified in practice and the three following chapters of this thesis act as proofs-of-concept.

In general, the applicability of the Uniform Memory Demands strategy, similar to other strategies, is identified a posteriori, in practice, and on a per-case-basis consid-

eration. To examine if the Uniform Memory Demands strategy works for a particular problem, first, we need to see if contrasting memory behaviours can be a source of inefficiency. Then, we should try to identify subsets of the input dataset (such as subgraphs/vertex classes) that show contrasting demands. In the next step, we have to design/find algorithms that satisfy these contrasting demands with the lowest overhead. After this final step, we can say the Unified Memory Demands strategy has been successful for this problem.

The Uniform Memory Demands strategy can also be generalized to facilitate optimized usage of other resources. As an example, in communication-intensive applications in a distributed system, the structure of the input datasets can be used to categorize the contrasting communications demands in order to satisfy them with lower communication overheads. In processing dynamic graphs [58, 139], the structure of the dataset can be used to design bespoke updatable data structures and updating algorithms for different subgraphs. In using Non-Volatile RAM (NVRAM) for graph processing, we may find similar relationships between read-intensive and write-intensive behaviours to the structure of the graph.

Chapter 5

SAPCo Sort: Structure-Aware Parallel Counting Sort

5.1 Introduction

Degree ordering, or sorting vertices IDs by their degrees, is a basic tool in several graph algorithms and frameworks [6, 38, 55, 102, 108, 159].

The input data is an array containing $|V|$ elements, where, each element contains the vertex ID (as key) and the vertex degree (as value). The output is similar to the input, but vertices are sorted by their degrees in descending or ascending order. Without loss of generality, we use the ascending order of degrees. In this way, the problem is to sort $n = |V|$ key-value pairs.

Several sorting algorithms with optimized complexities and implementations have been introduced [9, 30, 42, 56, 63, 93, 128, 140, 148]; however, they are not well-adjusted for real-world graphs with skewed degree distribution. The parallel algorithms that work based on sample sort [63] and radix sort [140], move elements several times until they are accommodated in their final places. On the other hand, counting sort [140] makes advantage of writing elements directly in their final places and has a time complexity of $\mathcal{O}(n)$ (while comparison-based sorting algorithms have a complexity of $\mathcal{O}(nlogn)$); but its parallelization is restricted by the range of values.

In this chapter, we deploy the Uniform Memory Demands strategy to analyze the parallel counting sort algorithm and the behaviour of different vertex classes in this algorithm. Then, we design the **Structure-Aware Parallel Counting (SAPCo) Sort** algorithm. The evaluation of SAPCo in comparison to parallel counting sort shows that

SAPCo is 33–71 times faster. In comparison to state-of-the-art sample sort and radix sort algorithms, SAPCo is 1.7–4.0 times faster.

5.2 Counting Sort

5.2.1 Sequential Counting Sort

Algorithm 3 shows the sequential counting sort of the input array IN containing n key-value pairs:

Step 1. Reading the input array and identifying the greatest value and assigning it to max_val (Lines 1–4).

Step 2. The input array is read and a **counters** array of length max_val is used to count the number of times different unique values occur in the input array (Lines 5–7).

Step 3. To specify the insertion point of the first occurrence of unique values in the output array, the prefix sum of *counters* is calculated and stored in the **Insertion Points (IP)** array (Lines 8–10).
If value val appears $r = counters_{val}$ times in the input array, IP reserves space for all r repetitions of v as $IP_{val+1} = IP_{val} + counters_{val}$.

Step 4. The input array is read again and values are placed in the output array using IP: After reading an element with value val, it is written on an index of the output array that is identified by the insertion point, IP_{val}, and IP_{val} is incremented to be ready for the next val (Lines 11–13).

As the *counters* array is not needed after Step 2, its allocated memory is used for IP; however, we use different names to mention distinct usages and contents.

5.2.2 Parallel Counting Sort

To parallelize counting sort, we consider parallelisation of each step. In Step 1, we need to replicate the max_val variable over all threads and to reduce the replicates to identify the max_val. To parallelize Steps 2-4, we need to identify the memory access types as IP and *counters* arrays are modified by concurrent threads. As a result, we have two options:

Algorithm 3: Sequential counting sort

Input: $struct\{\textbf{key};\ \textbf{val};\}\ IN[n]$
Output: $struct\{\textbf{key};\ \textbf{val};\}\ OUT[n]$

```
   /* Step 1: Identifying the greatest value                          */
1  max_val = 0;
2  for ( v ∈ IN ) do
3  |  if v.val > max then
4  |  |  max_val = v.val
5  max_val++;

   /* Step 2: Counting number of occurrences of different values      */
6  counters[max_val] = {0};
7  for ( v ∈ IN ) do
8  |  counters_{v.val}++;

   /* Step 3: Specifying insertion points                             */
9  IP[max_val] = {0};
10 for ( val = 1; val < max_val; val++ ) do
11 |  IP_{val} = IP_{val-1} + counters_{val-1};

   /* Step 4: Writing the output                                      */
12 for ( v ∈ IN ) do
13 |  OUT_{IP_{v.val}} = v;
14 |  IP_{v.val}++;
```

1. **Shared IP**: We divide the input array into partitions and threads read partitions of the input array and **atomically** increment the shared *counters* (Step 2), then, IP is calculated by parallel prefix sum (Step 3), and threads read the input array and use atomic memory accesses to get an insertion point from the shared IP (Step 4). To accelerate Step 2, per-thread *counters* can be used to avoid atomic memory accesses. In this case, the counters should be merged by the end of step.
2. **Private IP**: The input array is divided into partitions and per-partition *counters* arrays are allocated. Then, partitions are read by threads and their private *counters* are set (Step 2). A global *counters* array is accumulated by private *counters*, and the global IP is identified by parallel prefix sum. The global IP and the private *counters* of partitions are used to identify the private IP of each partition (Step 3). The input array is read again and private IP are used to identify the index required for writing to the output array (Step 4).

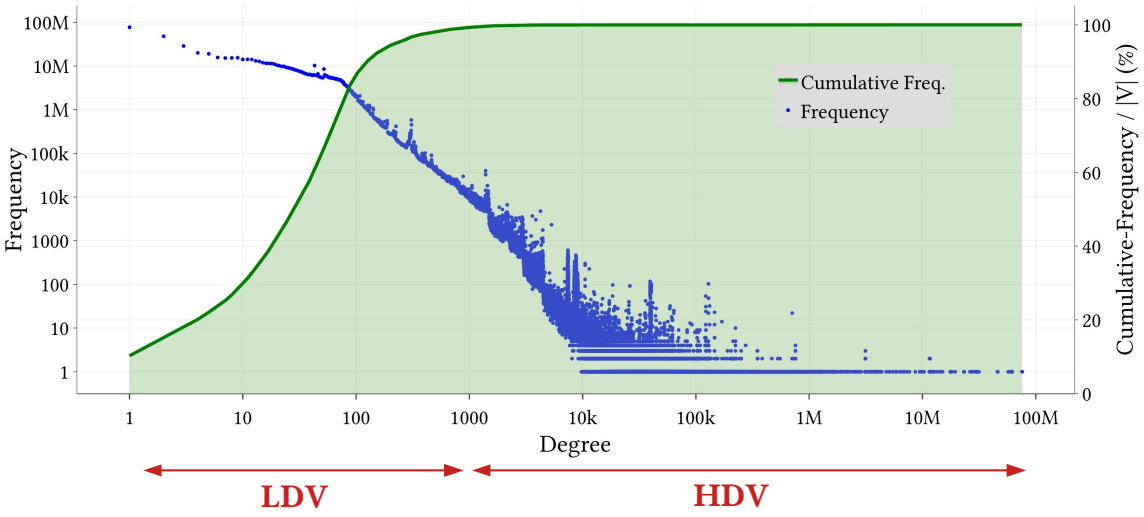


Figure 5.1: LDV and HDV in ClueWeb12

5.3 Algorithm Design

5.3.1 Step 1: Identifying Contrasting Demands & Behaviours

In the first parallelization approach, shared IP, we need $\mathcal{O}(n)$ atomic memory accesses during Step 4.

The applicability of the second approach, private IP, depends on the number of partitions (which is affected by the number of CPU cores and also affects the load balance) and the range of values, max_val. For p partitions, the memory space complexity is $\mathcal{O}(max\_val \cdot p)$. For a small max_val, p can be large enough to keep all processors busy; however, that is not the case for degree-ordering of real-world graphs where max_val may reach 95 million (Section 5.5). Moreover, Step 3 (merging private *counters* and calculating private IP) has a time complexity of $\mathcal{O}(max\_val \cdot p)$ that is increased by both max_val and p. In Step 2, also, we need to merge counters of t threads that results in $\mathcal{O}(max\_val \cdot t)$ memory accesses.

We see that each of these approaches suffers from particular limitations. The shared IP approach sacrifices performance in order to save memory space and the private IP provides better performance while requesting much greater memory space.

5.3.2 Step 2: Considering Potential Solutions

Figure 5.1 is the same as Figure 2.1 and shows the Low-Degree Vertices (**LDV**) and High-Degree Vertices (**HDV**). Since in power-law graphs, the number of LDV are expo-

nentially greater than HDV, in degree-ordering of these graphs, the input array has a very small number of HDV and a huge number of LDV.

As a result, when traversing the input array, the very small indices of *counters* and IP arrays (that correspond to LDV) are accessed frequently; but, the greater indices (that correspond to HDV) are rarely accessed.

5.3.3 Step 3: Matching & Adjusting

Since HDV are rare and have a wide range of values, it is more efficient to save memory space and time by allocating a shared memory array for HDV and using atomic memory accesses to protect it from concurrent accesses of threads processing different partitions.

In other words, shared IP is the best match for HDV. Here, we take advantage of lower memory space requirement by paying the cost of atomic memory accesses for HDV, but these are only a small percentage of the total memory accesses. The "Cumulative Frequency" plot in Figure 5.1 shows that less than 1% of vertices are HDV in the ClueWeb12 graph; therefore, less than 1% of memory accesses will be protected by atomic accesses.

In contrast, LDV are frequent and their corresponding indices that are accessed in the *counters* and IP arrays are confined to a short range. So, it is more efficient to deploy private IP for LDV that assigns per-partition private memory for IP arrays to accelerate their accesses that form almost all of the memory accesses (more than 99% in Figure 5.1). On the other hand, by limiting the range of values to LDV, we avoid high memory space consumption of private IP approach.

5.3.4 Step 4: Merging

We explained that the private IP is the best match for LDV and the shared IP is the best one for HDV. In order to deploy both these algorithms, we need to specify a threshold between LDV and HDV.

We use $tsld = min(1000,\ 0.5 * max\_val\ )$ as the threshold between HDV and LDV. We control the size of total memory space by using 1000 as the maximum value of this threshold that also acts as a suitable border between HDV and LDV in our graphs. The $0.5 * max\_val$ prevents assigning the total range of values for per-partition IP arrays when range of values is small, i.e., $max\_val < 2000$.

By using this threshold we limit atomic memory accesses to HDV to prevent sacrific-

ing performance. On the other hand, we assign private memory for LDV that form most of the memory accesses. In this way, we limit the replicated private memory space and overheads of aggregating *counters* and disseminating IP to only *tsld* (and not *max_val*) indices.

5.4 SAPCo Sort Algorithm

Algorithm 4 shows the SAPCo sort. In this algorithm, we call degree of vertices as their values.

Step 1. In Lines 1–9, we identify the maximum value (i.e., maximum degree) of the input and we assign a **private counters (pcounters)** array of size *tsld* for each partition. We also create a **global counters (gcounters)** array of size *max_val*.

Step 2. In Lines 10–15, threads process elements in each partition of the input array. For an element with value *val*, if $val < tsld$, $pcounters_val$ of the partition is incremented; otherwise, $gcounters_val$ is atomically incremented.

Step 3. In Lines 16–19, we merge *pcounters* of different partitions into the *gcounters*. Then, by applying parallel prefix sum on the *gcounters*, the **Global Insertion Points (GIP)** array is identified. By using GIP and *pcounters*, **Private Insertion Points (PIP)** arrays of LDV in different partitions are identified (Lines 21-25).

Step 4. In Lines 26–32, the final pass over partitions of the input array is performed by threads. When reading a value *val*, if *val* is a LDV, PIP_{val} of the partition identifies the insertion point (ip) in the output and PIP_{val} is incremented. If *val* is a HDV, the GIP_{val} identifies the insertion point in the output array and is atomically increased by one.

5.5 Evaluation

We use the SkyLakeX-2 machine (Appendix A) to evaluate SAPCo in comparison to (i) counting sort with shared IP, (ii) counting sort with private IP, (iii) IPS^{2}Ra radix sort[1] (commit 18795bb) [9], and (iv) IPS^{4}o sample sort[2] (commit d7a74ab) [9].

[1] https://github.com/ips4o/ips2ra
[2] https://github.com/ips4o/ips4o

Algorithm 4: SAPCo Sort

Input: $struct\{\textbf{key};\ \textbf{val};\}\ IN[n]$
Output: $struct\{\textbf{key};\ \textbf{val};\}\ OUT[n]$

```
   /* Step 1: Identifying the greatest value                          */
1  max_val = 0;
2  par_for ( v ∈ IN )
3  |  if v.val > max then
4  |  |  max_val = v.val
5  max_val++;
6  tsld = min(1000, 0.5 * max_val) ;          // threshold between LDV and HDV
7  pc = 64 * #threads ;                                // number of partitions
8  gcounters[max_val] = {0} ;                             // global counters
9  pcounters[pc][tsld] = {0} ;            // private (per partition) counters

   /* Step 2: Counting occurrence of different values                 */
10 par_for ( p = 0; p < pc; p++ )
11 |  for ( v ∈ IN_p ) do
12 |  |  if v.val < tsld then
13 |  |  |  pcounters^p_{v.val}++ ;                                    // LDV
14 |  |  else
15 |  |  |  atomic_inc(gcounters_{v.val}) ;                            // HDV

   /* Step 3: Specifying insertion points                             */
16 par_for ( val = 0; val < tsld; val++ )
17 |  for ( p = 0; p < pc; p++ ) do
18 |  |  gcounters_{val} += pcounters^p_{val} ;          // merging private counters
19 GIP = par_prefix_sum(gcounters) ;                                 // global IP
20 PIP = pcounters ;                                // private (per partition) IP
21 par_for ( val = 0; val < tsld; val++ )
22 |  for ( p = 0; p < pc; p++ ) do
23 |  |  tmp = pcounters^p_{val};
24 |  |  PIP^p_{val} = GIP_{val};
25 |  |  GIP_{val} += tmp;

   /* Step 4: Writing the output                                      */
26 par_for ( p = 0; p < pc; p++ )
27 |  for ( v ∈ IN_p ) do
28 |  |  if v.val < tsld then
29 |  |  |  ip = PIP^p_{v.val}++ ;                                     // LDV
30 |  |  else
31 |  |  |  ip = atomic_get_and_inc(GIP_{v.val}) ;                     // HDV
32 |  |  OUT_{ip} = v;
```

Dataset	$\|V\|$ (M)	Max. Degree	Performance (Milliseconds)				
			Cnt. Shared	Cnt. Private	IPS^{2}Ra	IPS^{4}o	SAPCo
GBRd	7.7	7	463	**5.0**	9.9	10.2	**5.0**
USRd	23.9	8	1,334	**10.9**	25.6	22.5	11.0
WWik	2.1	1.14 M	119	573	12.6	**4.3**	4.8
LJ	5.2	15.0 K	210	19.2	10.0	6.4	**5.5**
LJGrp	7.5	1.05 M	315	799	23.1	9.8	**8.9**
Twtr10	21.3	422 K	1,130	166	55.9	21.4	**18.7**
Twtr	28.5	278 K	1,324	213	73.1	28.6	**20.5**
TwtrMpi	41.7	770 K	1,687	422	103	40.8	**37.5**
SK	50.6	8.56 M	2,286	6,120	130	50.1	**33.8**
Frndstr	65.6	3,615	2,765	39.4	122	65.0	**35.7**
WbCc	89.1	2.33 M	4,226	1,369	228	82.5	**55.9**
UKDmn	105.2	975 K	2,280	629	266	100	**57.7**
UKDls	109.5	1.26 M	4,649	984	276	109	**56.7**
WebB	118.1	816 K	5,591	783	296	117	**54.4**
UU	133.6	6.37 M	5,511	3,478	335	134	**66.6**
GSH	988.5	58.8 M	31,541	24,175	2,948	936	**467**
ClWb9	1,685	6.44 M	86,988	5,336	4,203	1,725	**781**
WDC14	1,725	45.7 M	87,792	27,643	5,744	1,732	**679**
WDC12	3,564	95.0 M	151,382	–	11,021	3,344	**1,537**

Table 5.1: Performance of sorting algorithms: counting sort with Shared IP ("Cnt. Shared") and Private IP ("Cnt. Private"), IPS^{2}Ra (radix sort), IPS^{4}o (sample sort), and SAPCo - Failed attempts are shown by dash.

5.5.1 Performance Evaluation

Table 5.1 shows the numbers of vertices of graph ($|V|$) in millions (which specifies the number of elements in the input array, n). Column 3 of Table 5.1, "Max. Degree", shows the maximum in-degree of graphs (which specifies the value of *max_val* in Section 5.4).

Table 5.1 shows that **SAPCo is, on average, 1.7× faster than IPS^{4}o, 4.0× faster than IPS^{2}Ra**, 33.5× faster than counting sort with private IP, and 71.5× faster than counting sort with a shared IP.

Dataset	$\|V\|$	Max.	Memory Accesses			HW Instructions		
	(M)	Degree	IPS^{2}Ra	IPS^{4}o	SAPCo	IPS^{2}Ra	IPS^{4}o	SAPCo
GBRd	7.7	7	13.8	16.7	12.1	52.0	47.2	34.6
USRd	23.9	8	13.8	16.5	12.0	48.9	46.1	34.3
WWik	2.1	1.14 M	38.3	30.4	16.0	99.9	80.2	52.6
LJ	5.2	15.0 K	29.2	27.4	13.1	76.9	74.6	39.6
LJGrp	7.5	1.05 M	35.4	33.8	13.2	90.6	87.0	39.8
Twtr10	21.3	422 K	32.4	23.4	12.4	83.6	65.7	36.2
Twtr	28.5	278 K	34.6	29.3	12.3	87.7	76.7	35.6
TwtrMpi	41.7	770 K	30.9	28.3	12.3	81.9	75.2	35.7
SK	50.6	8.56 M	31.8	26.3	12.6	82.3	70.8	36.4
Frndstr	65.6	3,615	30.9	29.0	12.1	81.0	77.6	34.7
WbCc	89.1	2.33 M	32.6	25.5	12.2	83.6	69.5	35.2
UKDmn	105.2	975 K	37.4	28.9	12.1	92.9	76.9	34.6
UKDls	109.5	1.26 M	33.0	27.8	12.1	85.1	73.8	34.6
WebB	118.1	816 K	30.1	24.6	12.1	79.8	66.0	34.4
UU	133.6	6.37 M	35.3	31.7	12.2	89.3	81.1	34.8
GSH	988.5	58.8 M	37.1	32.0	12.2	94.7	82.4	34.5
ClWb9	1,685	6.44 M	28.7	25.1	12.1	78.6	66.6	34.4
WDC14	1,725	45.7 M	36.9	25.2	12.1	95.3	67.0	34.3
WDC12	3,564	95.0 M	43.4	30.7	12.1	106.5	79.3	34.3

Table 5.2: Comparison of memory accesses and hardware instructions. Values are divided by the number of elements ($|V|$) - "Memory Accesses" are load and store instructions

5.5.2 Hardware Instructions and Memory Accesses

We compare memory accesses and hardware instructions in Table 5.1 that shows **SAPCo, on average, performs 12.6 memory accesses per vertex while, IPS^{4}o requires 27.4 accesses**. Moreover, **SAPCo requires 37.1 hardware instructions per vertex, on average while, IPS^{4}o requires 72.8 instructions**. This shows that SAPCo facilitates better work-efficiency.

5.6 Conclusion and Further Applications

In this chapter, we used the Uniform Memory Demands strategy to distinguish different memory demands in parallel counting sort, and to introduce the SAPCo sort that

uses per thread private data structures for low-degree vertices while using shared data structures for high degree vertices.

The structure-aware design of SAPCo provides 33–71 times speedup in comparison to parallel counting sort. Moreover, SAPCo is 1.7–4.0 times faster than comparison-based sorting algorithm that makes SAPCo the first parallel counting sort that is practically faster than comparison-based sorting algorithms, however, in a special application domain (having skewed frequency in the input data).

The SAPCo Sort algorithm can be used in identifying k vertices with maximum degrees (in algorithms such as iHTL and Lotus which are introduced in Chapter 6 and Chapter 7, respectively). In this case, we do not need to count the LDV and we do not need to assign memory for the *pcounters* (i.e., we skip Lines 9, 13, 16–18, 21-25, and 29 in Algorithm 4). Similar changes can be applied to identify k vertices with minimum degrees.

Chapter 6

iHTL: Exploiting in-Hub Temporal Locality in SpMV

6.1 Introduction

In Chapter 3, we explained that SpMV-based graph processing (where all edges of the graph are traversed in the pull direction) suffers from poor locality in processing hubs. We also identified that locality-optimizing relabeling algorithms cannot improve locality of hubs.

In this chapter, we use the Uniform Memory Demands strategy to design the iHTL algorithm that identifies different memory demands in a power-law graph and deploys distinct traversal directions for each subgraph. The iHTL algorithm extracts subgraphs containing incoming edges to in-hubs and processes them in the push direction. The push direction traversal dedicates the cache to data of the destinations of the edges in these subgraphs that are exponentially smaller in count than their sources. For the remainder of the graph, iHTL uses the pull direction.

The evaluation of iHTL shows that iHTL is 1.5× - 2.4× faster than state-of-the-art implementations of the pull traversal. More importantly, iHTL is 1.3× - 1.5× faster than the pull traversal of state-of-the-art locality-optimizing reordering algorithms while iHTL's preprocessing is much faster than the preprocessing of the locality-optimizing relabeling algorithms.

Section 6.2 explains the design of iHTL and the detailed iHTL algorithm is presented in Section 6.3 that is followed by the evaluation of iHTL in Section 6.4. In Section 6.5, we compare the iHTL to the cache blocking technique and Section 6.6 contains the future

work and application of the iHTL technique on other graph algorithms.

6.2 Algorithm Design

6.2.1 Step 1: Identifying Contrasting Demands & Behaviours

In Section 2.4.1, we explained that SpMV is used in several graph algorithms and is performed in two directions: pull direction (traversing incoming edges to vertices, Algorithm 1) or push direction (traversing outgoing edges to vertices, Algorithm 2). The pull direction is faster as it does not require protecting data of vertices.

In Section 3.4.4, we explained that pull processing of in-hubs suffers from poor locality as a massive amount of vertex data is pulled into the cache which displaces much of the cache contents and intensely reduces the opportunity for future reuse. We also observed that locality optimizing relabeling algorithms does not improve locality of hubs. Moreover, efficient processing of hubs becomes more important as they dedicate a great percentage of total edges (therefore, a great percentage of processing time) to themselves.

6.2.2 Step 2: Considering Potential Solutions

As we explained in Section 3.2.2, in a pull traversal (Algorithm 1), each vertex reads data of its in-neighbours (Line 4) and writes its new data. Therefore most of the capacity of **cache is dedicated to support random read accesses to data of source vertices in pull traversal**.

In a push traversal (Algorithm 2), on the other hand, the new data of out-neighbours (Line 5) are randomly updated by data of source vertices and most of the capacity of **cache is dedicated to support random write accesses to destination vertices in push traversal**.

6.2.3 Step 3: Matching & Adjusting

The skewed degree distribution of graphs implies that for processing incoming edges to in-hubs, the number of destination vertices (in-hubs) is much less than the number of source vertices. Therefore, cache is efficiently used only if it is dedicated to the destination vertices. In other words, **push direction is suitable for traversing incoming edges to in-hubs**.

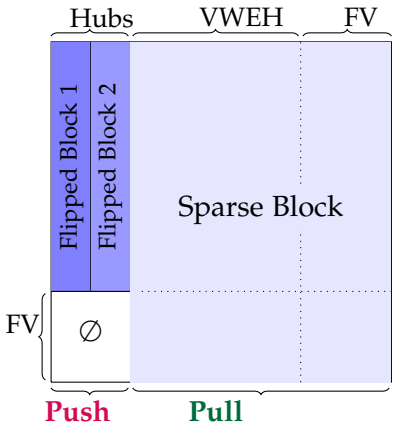


Figure 6.1: Adjacency matrix of an iHTL graph with two Flipped Blocks - Pull direction corresponds to a column-major traversal and push direction corresponds to a row-major traversal - The Zero Block (∅) does not contain any edges

On the other hand, when processing incoming edges to non in-hubs, we need to access data of a small number of in-neighbours. The pull direction facilitates better locality while it does not require race protection.

6.2.4 Step 4: Merging

As we want to apply different directions for different subgraphs, we need to separate subgraphs. As a result, we need a preprocessing step that separates incoming edges to in-hubs from other edges.

6.3 iHTL: In-Hub Temporal Locality

6.3.1 iHTL Graph

In order to facilitate push and pull traversals in iHTL, we distinguish blocks (i.e., subgraphs) within the graph adjacency matrix. Figure 6.1 shows the adjacency matrix of an iHTL graph. We use the convention that a pull traversal corresponds to a column-major traversal of the adjacency sub-matrix, while a push traversal corresponds to a row-major traversal.

The iHTL graph is comprised of:

- A number of **Flipped Blocks** that contain incoming edges to in-hubs,
- A **Sparse Block** that contains edges to non-hubs, and

- A **Zero Block** that contains no edges.

iHTL uses push traversal for processing incoming edges to in-hubs and it is necessary to ensure data of in-hubs are maintained in the cache. For a graph that has more in-hubs than the cache has capacity for, iHTL creates multiple flipped blocks.

Due to the skewed degree distribution of graphs, flipped blocks are very dense (contain few hubs, but many edges). We will see in Section 6.4.5 that flipped blocks in iHTL contain up to 70% of the edges.

The sparse block, on the other hand, contains edges to non-hubs and iHTL uses pull traversal which dedicates cache to the source vertices of edges and since there is no in-hub in the sparse block, reuse of cache contents is improved.

To create these blocks, iHTL categorizes vertices into:

- **In-hubs**,
- **VWEH**: Vertices With Edges to Hubs, and
- **FV**: Fringe Vertices, which have no edges to in-hubs.

In iHTL, all edges in the flipped blocks are either edges from VWEH to in-hubs, or from in-hubs to in-hubs. Fringe vertices do not link to in-hubs. As such, they do not appear in flipped blocks and a **zero block** (∅) appears in the adjacency matrix (Figure 6.1).

We separate out fringe vertices in order to (1) avoid loading their vertex data from main memory during processing of flipped blocks, and also to (2) shrink the size of topology data of flipped blocks.

6.3.2 Creating The iHTL Graph

The iHTL graph (Figure 6.1) is created in 3 steps:

(1) Creating The Relabeling Array: To enforce the new arrangement of vertices, the iHTL relabeling array is created such that all in-hubs have smaller labels than VWEH and all VWEH have smaller labels than FV. iHTL brings vertices of the same type (in-hubs, VWEH, and FV) close to each other by assigning consecutive IDs. However, it keeps the initial order between vertices of the same type in VWEH and FV. In this way, iHTL tries to have minimal impacts on the initial neighbourhood of the vertices, which is important to retain locality that exists in the graph.

Firstly, in-hubs are selected as a number of vertices with the highest in-degree and first IDs are dedicated to in-hubs. The number of in-hubs depends on the number of flipped blocks and is discussed in Section 6.3.3. Secondly, the VWEH is identified by

traversing CSC representation of the main graph for the selected in-hub vertices. The remaining vertices are FV.

It is worth mentioning that contrary to locality optimizing relabeling algorithms like GOrder and Rabbit-Order, the relabeling array in iHTL does not improve locality and is used to form the subgraphs that are required in iHTL adjacency matrix. Locality is improved in iHTL by increasing reuse in push traversal for incoming edges to in-hub vertices.

(2) Creating Flipped Blocks: Flipped blocks in iHTL contain in-edges of in-hubs. If a flipped block contains H in-hubs, then the i-th flipped block contains edges to the in-hubs with IDs in the range $HR_i = [(i-1)H, iH)$. Creating flipped blocks requires a pass over outgoing edges from $\{hubs \cup VWEH\}$ in the CSR representation of the main graph and selecting edges with in-hub destinations (that are identified by using the iHTL relabeling array).

(3) Creating The Sparse Block: The sparse block of iHTL contains edges to non-hubs that are processed in pull direction. It is formed by a pass over the CSC representation of the main graph for all in-edges to $\{VWEH \cup FV\}$ and relabeling source of edges using the iHTL relabeling array.

6.3.3 Number of in-Hubs and Flipped Blocks

The main benefit of iHTL is to traverse the flipped blocks such that random accesses are made to the few hubs that are maintained in the cache. To accomplish this, the number of hubs is dimensioned based on a combination of cache size and graph structure. Taking cache size into account is critical to catch the random accesses to the in-hubs on chip. However, graph data sets may require more hubs than contemporary processor cache sizes can handle. Because of this, **iHTL fixes the number of in-hubs in a flipped block based on cache size and constructs multiple flipped blocks as needed based on graph structure**.

The number of in-hubs in a flipped block is determined by the on-chip cache size. We identified that the level 2 cache is the best location for holding the vertex data of in-hubs (Section 6.4.6). As such, we specify the number of hubs per flipped block as H by dividing the level 2 cache size by the size of vertex data.

If graph structure mandates more hubs, we increase the number of flipped blocks. Therefore, **iHTL needs to balance the benefit of creating more flipped blocks with the drawbacks**. The benefit is improved locality, however, there are two drawbacks of

Algorithm 5: SpMV in iHTL

Input: $iHTL\_graph\ h$, $\mathcal{D}^{i-1}$, $\mathcal{D}^{i}$

/* Push traversal of the flipped blocks */

1 **par_for** $fb \in h.flipped\_blocks$

2 **par_for** $v \in \{h.hubs \cup h.VWEH\}$

3 **foreach** $hub \in fb.hubs_v$ **do**

4 $\textbf{buffer\_d}^{tid}_{hub} += \mathcal{D}^{i-1}_{v}$;

/* Aggregation of thread buffers */

5 **par_for** $hub \in h.hubs$

6 **foreach** $t \in threads$ **do**

7 $\mathcal{D}^{i}_{hub} += \textbf{buffer\_d}^{t}_{hub}$;

/* Pull traversal of the sparse block */

8 **par_for** $v \in \{h.VWEH \cup h.FV\}$

9 **foreach** $u \in N^{-}_{v}$ **do**

10 $\mathcal{D}^{i}_{v} += \mathcal{D}^{i-1}_{u}$;

increasing the number of flipped blocks: (1) While all members of $\{hubs \cup VWEH\}$ have edges to in-hubs in the first flipped block, this number diminishes in subsequent flipped blocks, reducing efficiency as some fetched vertex data will not be used during push traversal. (2) Flipped blocks, moreover, increase the size of the graph topology data, as each block requires its own metadata. Based on these observations, **iHTL allows a new flipped block to be formed if its hubs have edges from at least 50% of the $\{$hubs $\cup$ VWEH$\}$**.

If $HV_i = \{s \in \{hubs \cup VWEH\} | \exists (s,h) \in E \wedge h \in HR_i\}$, iHTL increases the number of flipped blocks ($\#fb$), while $|HV_{\#fb}| > 0.5 * |HV_1|$. In order to calculate $|HV_i|$, a pass over in-edges to H in-hub vertices in the i-th flipped-block is required to mark the HV members and one other pass is needed to count the number of marked vertices.

6.3.4 iHTL Processing

In parallel processing of a flipped block, concurrent threads will perform random updates to the vertex data of in-hubs. To avoid race conditions, we opt for the buffering technique 2.4.5 (where each thread operates on copies of the vertex data which are later merged) as it is more efficient in the setting of iHTL using the private and fast L2 cache for each thread. As we see in the evaluation, buffer merging in iHTL does not take more than 3% of iHTL execution time (each thread buffers $H * \#fb$ vertex data).

Algorithm 5 shows SpMV execution for iHTL. Flipped blocks (Lines 1-4) use push

traversal in iHLT. For each flipped block, the old data of a vertex v that has edges to hubs ($fb.hubs_v$) is read and the related index of the local buffer (**buffer_d**^{tid}) of thread (tid) is updated. Since threads write updates locally during processing flipped blocks, the parallel `for` loop in Line 1 does not require synchronization between threads and **different threads can process vertices of different flipped blocks**. However, each thread should process only one flipped block at a time.

After completion of processing flipped blocks, thread buffers are merged (Lines 5-7) to specify data of hubs. Finally, pull traversal is used for processing the sparse block (Lines 8-10).

6.4 Evaluation

We use the SkyLakeX-2 machine (Appendix A) to evaluate iHTL in comparison to (i) GraphGrind[1] (commit `5099761`) [153], (ii) GraphIt[2] (commit `c4781d8`, OpenMP) [175] , and (iii) Galois[3] (V5, commit `6ce5f0d`) [66, 123] are graph processing frameworks we use to evaluate iHTL.

We evaluate iHTL using the PageRank application which has been implemented in all graph processing frameworks and iteratively performs SpMV-type calculations: $PR^i_v = \frac{0.15}{n} + 0.85 \sum_{u \in N^-_v} \frac{PR^{i-1}_u}{|N^+_u|}$. The vertex data size is 8 bytes.

6.4.1 iHTL vs Pull and Push Implementations

Table 6.1 compares per iteration PageRank execution time for iHTL vs pull and push traversals in different graph processing frameworks (Galois does not include PageRank in push direction). We compare against several frameworks as each applies a different set of optimizations. GraphGrind performs an edge-balanced partitioning for a pull traversal. GraphIt includes the Cagra [174] locality optimizations (Section 6.5) which make it faster than Galois for some graphs. Table 6.1 demonstrates the effectiveness of the iHTL locality optimizations as it is faster than different implementations of pull traversal by 1.5× - 2.4×.

Table 6.1 also shows that iHTL preserves the initial locality of graphs well, even for graphs like "SK-Domain" with high initial locality.

[1] https://github.com/DIPSA-QUB/GraphGrind

[2] https://github.com/GraphIt-DSL/graphit

[3] https://github.com/IntelligentSoftwareSystems/Galois/

	Push		Pull			iHTL
	GGrind	GraphIt	GGrind	GraphIt	Galois	
LvJrnl	91	770	54	106	37	28
Twtr10	176	340	143	76	114	57
TwtrMpi	895	1,606	693	402	422	268
Frndstr	1,352	2,023	1,149	858	885	627
SK	828	2,547	289	187	176	112
WbCc	1,245	1,444	981	606	664	382
UKDls	1,606	1,346	535	312	281	231
UU	2,479	3,626	757	430	390	320
UKDmn	2,637	1,827	806	439	407	348
ClWb9	6,844	6,220	7,301	3,405	4,407	2,367
Avg. Speedup	**4.8×**	**9.5×**	**2.4×**	**1.7×**	**1.5×**	**1×**

Table 6.1: Performance of one iteration of SpMV PageRank (in milliseconds) in push direction, pull direction, and iHTL

6.4.2 Memory Accesses and Cache Misses

Table 6.2 compares the memory accesses (loads and stores instructions) and also the level 3 cache misses for pull traversal vs iHTL, captured using PAPI. iHTL incurs additional memory accesses due to: (1) increased volume of topology data (Section 6.4.3), (2) additional memory accesses in processing flipped blocks (when data of one vertex is read in different flipped blocks), (3) merging buffers and (4) resetting buffers. However, all these memory accesses are sequential and assisted by prefetching.

As such, the key distinction in cache misses that impacts performance occurs when processing in-hubs: **where the pull traversal performs random reads that result in L3 cache misses, iHTL performs random writes captured by the L2 cache**. This large difference in L3 cache misses is a key explainer for the performance of iHTL.

6.4.3 Memory Space Overhead

Table 6.3 compares the memory size of CSC representation of the graphs in comparison to their iHTL graphs. The topology data grows in iHTL compared to a standard compressed sparse columns representation. This results from replication of the index array for each block. However, topology data is read sequentially from main memory as the graph topology and is accelerated by prefetching. The size increase is therefore not a major problem.

Dataset	Memory Accesses		L3 Cache Misses		L2 Cache Misses	
	Pull	iHTL	Pull	iHTL	Pull	iHTL
LvJrnl	502	630	25	23	148	54
Twtr10	662	1,216	72	61	207	132
TwtrMpi	3,219	5,917	510	341	1,023	606
Frndstr	4,042	6,627	1,317	974	1,733	1,392
SK	4,243	5,702	194	169	316	235
WbCc	4,656	5,715	673	540	1,167	817
UKDls	8,630	10,803	346	349	480	477
UU	11,917	14,637	493	469	782	647
UKDmn	13,942	15,923	525	528	730	729
ClWb9	25,306	26,797	3,537	3,207	3,869	3,539

Table 6.2: Memory accesses (load and store instructions), L3 and L2 cache misses (in millions)

Dataset	**CSC** (GiB)	**iHTL** (GiB)	**iHTL Overhead** (%)
LvJrnl	.9	1.0	3
Twtr10	1.2	1.9	57
TwtrMpi	6.2	9.7	56
Frndstr	7.5	10.7	42
SK	8.2	8.5	4
WbCc	8.5	8.9	5
UKDls	16.5	17.0	3
UU	22.5	23.3	3
UKDmn	26.6	27.2	2
ClWb9	43.8	44.9	3

Table 6.3: Size of topology data (in GigaBytes)

6.4.4 iHTL vs Relabeling Algorithms

To have a better scale of locality optimization of iHTL, Table 6.4 compares PageRank execution time for iHTL and pull traversal of the datasets after relabeling by SlashBurn (SB), GOrder (GO), and Rabbit-Order (RO).

Relabeling algorithms rearrange the vertices to provide better reuse of vertex data, and as we explained in Section 3.4.4, they can provide better locality for non-hub vertices. However, a structure-agnostic pull traversal does not allow relabeling algorithms to improve locality of hubs. In contrast, iHTL targets locality of hubs, which capture a significant portion of the edges (Table 6.5) and results in out-performing the relabeling

	Iteration Time (ms)				Preprocessing Time (s)			
	SB Pull	GO Pull	RO Pull	iHTL	SB	GO	RO	iHTL
LvJrnl	44	45	48	28	4	362	6	0.9
Twtr10	63	101	84	57	9	712	15	0.9
TwtrMpi	345	306	399	268	68	5,697	66	4.9
Frndstr	841	682	652	627	78	4,894	139	5.8
SK	212	192	153	112	240	588	35	4
WbCc	601	492	410	382	112	6,587	72	3.5
UKDls	356		234	231	1,044		67	3.3
UU	537		346	320	1,736		80	3.8
UKDmn	492		399	348	1,022		69	5.5
ClWb9	3,147			2,367	416			16.9
Avg. Speedup	**1.5×**	**1.4×**	**1.3×**	**1×**	**>200×**	**>2000×**	**38×**	**1×**

Table 6.4: Left: Execution time (in milliseconds) of pull traversal after relabeling vs iHTL - Right: The preprocessing time of relabeling algorithms vs iHTL (in seconds)

algorithms. Figure 6.2 compares the last level cache miss rates of iHTL and relabeling algorithms.

The right side of Table 6.4 compares the preprocessing time of iHTL to relabeling algorithms. GOrder has a sequential implementation. SlashBurn and Rabbit-Order have parallel codes; however, the complexity of their algorithms makes them much slower than iHTL. iHTL has a simple preprocessing algorithm (Section 6.3.2) and does not need to investigate the neighbourhood of each vertex in detail. This gives iHTL a very short preprocessing time.

6.4.5 Execution Breakdown & Graph Statistics

Table 6.5 characterizes the iHTL graph and relative processing speed for flipped blocks. For social networks, flipped blocks contain 45% - 65% of the edges. The push traversal of flipped blocks makes good use of the sequentially fetched vertex data, as a high percentage of the vertices link to the hubs (column "VWEH"). As a result, iHTL spends just 22% - 40% of its time for processing flipped blocks of social networks. Web graphs contain only one flipped block that contains 40% of edges on average and is processed in just 25% of the processing time, on average.

The relatively high processing speed of flipped blocks compared to the whole graph is captured by the **flipped block speed** (column "FB speed"). It is calculated as the percentage of edges in the flipped blocks divided by the relative time spent in flipped

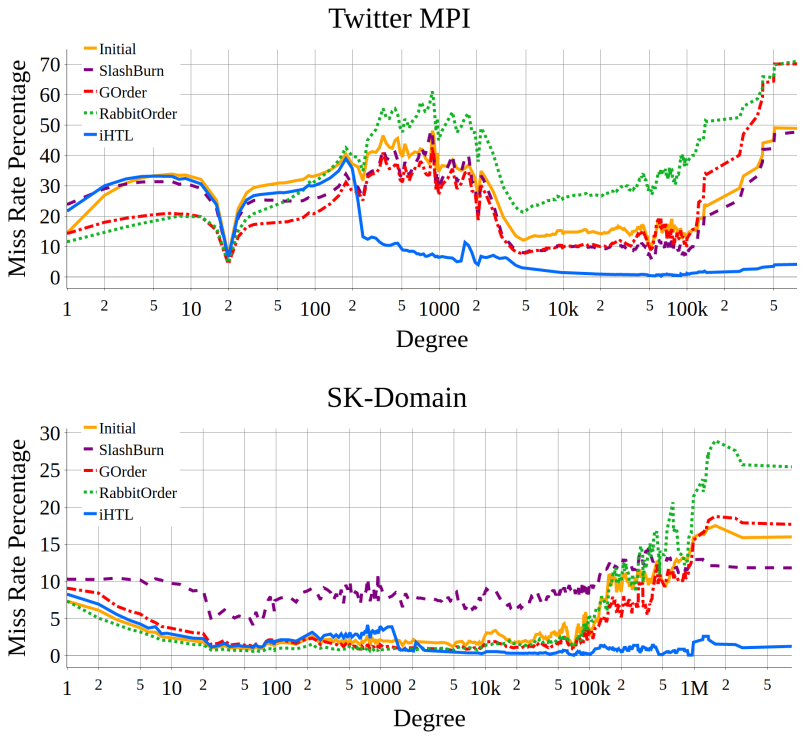


Figure 6.2: [Simulation] The last level cache miss rate of SpMV conditional on the degree of the traversed vertex

blocks. Values higher than 1 indicate that an edge in a flipped block is processed more efficiently than average across the graph. This is a consequence of containing the random memory accesses in the on-chip caches during processing of flipped blocks, which cannot be guaranteed for the sparse block.

Table 6.5 shows that buffer aggregation in iHTL takes less than 2.5% of the total processing. Each flipped block implies buffer merging overhead; however, by inspecting the graph structure (Section 6.3.3), iHTL incurs this overhead only when there is a corresponding gain in locality.

6.4.6 Buffer Size

Table 6.6 shows the impact of iHTL buffer size. As we explained in Section 6.3.3, the buffer size determines the number of hubs in each flipped block. So, we need to set the best size for buffer in order to provide the fastest random accesses to the buffers while processing flipped blocks.

Table 6.6 shows that aligning the buffers to L1 cache size is inefficient as its 32 KB

Dataset	Graph Topology Statistics				Exec. Breakdown		
	#FB	VWEH	Min. Hub Degree	FB Edges	FB Time	Buffer Merging	FB Speed
LvJrnl	1	47%	158	47%	32%	2.38%	1.48
Twtr10	2	28%	109	67%	37%	1.73%	1.81
TwtrMpi	8	87%	223	59%	41%	1.73%	1.46
Frndstr	16	60%	192	45%	22%	1.56%	2.00
SK	1	78%	1,389	68%	48%	0.52%	1.43
WbCc	1	56%	1,351	44%	13%	0.18%	3.32
UKDls	1	65%	4,844	49%	34%	0.28%	1.45
UU	1	71%	3,703	44%	32%	0.22%	1.39
UKDmn	1	67%	3,961	27%	21%	0.19%	1.26
ClWb9	1	9%	2,654	13%	4%	0.03%	2.94

Table 6.5: iHTL graph statistics and iHTL PageRank execution breakdown (FB: flipped blocks)

size is too small to accommodate data of many hubs. The L2 cache is private to each core, which implies unfettered access. Increasing the buffer size beyond the L2 size is detrimental for social networks; however, web graphs tolerate this well. Increasing buffer size beyond L2 size can be seen to be sub-optimal as it increases usage of the L3 cache, which is shared between the threads and is non-inclusive and non-exclusive (NINE) with L2. Hence L3 size (22 MB per 16 cores) provides only fractionally more space per core compared to the 1 MB L2 cache. Consequently, L2 cache is the best choice for accommodating data of in-hub vertices.

Dataset	L1-Size	L2-Size / 2	L2-Size	L2-Size $*$ 2
TwtrMpi	340	269	268	329
Frndstr	778	652	627	669
WbCc	424	384	382	376
UKDls	242	235	231	228
UU	337	326	320	318
UKDmn	355	347	348	343
ClWb9	2,424	2,356	2,367	2,371

Table 6.6: Execution time (in milliseconds) for different buffer sizes

6.5 Related Work

In this section, we compare iHTL to the cache blocking techniques that are used in SpMV-based graph processing and explain the limitation of these techniques for power-law graphs.

Starting from [80], blocking techniques have been widely used to optimize different metrics in distributed memory and shared memory computing systems. SpMV cache blocking technique [78,79] works based on bringing a data block to cache, performing all random accesses to these data (that results in cache hits), and continuing to the next block.

Vertical cache blocking (or partitioning edges by destination 2.4.6) is used in push traversals of some graph frameworks [153,159,173] in order to prevent race conditions between concurrent updates.

Cagra [174] applies horizontal cache blocking of the adjacency matrix in a pull traversal that limits the range of random memory accesses during processing of a block and cache misses are reduced. Per-thread buffers are used in Cagra to contain intermediate updates of data of all vertices.

LAV [169] reduces the overheads of Cagra by creating horizontal dense blocks only for those out-hubs that, in total, capture 80% of the out-edges. To avoid buffer merging and to reduce cache misses, LAV prevents concurrent processing of blocks that may introduce load imbalance.

iHTL provides an efficient buffering that is limited to in-hubs. Moreover, iHTL's flipped blocks are easily load-balanced and are processed concurrently (Section 6.3.4).

Since real-world graphs are not truly power-law graphs [31], it is not always possible to select the number of dense blocks using estimated degree distribution statistics. So, iHTL identifies the number of flipped blocks by assessing the relation between hubs independently of their degree (Section 6.3.3), and flipped blocks contain a wide range of 13% - 68% of the edges (Table 6.5).

On the other hand, **efficient horizontal blocking based on out-degrees is, fundamentally, impossible in some graphs**. As we explained in Section 3.5.1, in-hubs are almost symmetric in social networks (in-hubs are out-hubs), but web graphs do not have symmetric in-hubs. Therefore, the lack of very high out-degrees in the graph implies that horizontal blocking cannot create dense blocks, which is most prominent in web graphs. In this case, horizontal blocking is not able to improve locality and increases

the overhead of reading topology data. Similarly, if the graph does not have very high in-degree vertices, it is not possible to create vertical dense blocks.

As an example, `SK-Domain` with maximum in- and out-degrees of 8.6 million and 13 kilo, has in-hubs and no out-hub, and for creating horizontal dense blocks based on out-hubs, 36% of vertices are required to capture 80% of edges, but iHTL creates a single vertical flipped block that contains 68% of the edges by selecting 0.3% of the vertices as in-hubs (Section 6.4.5). In this way, **iHTL creates flipped blocks using the same type of hubs that experience low locality**: in order to optimize locality of a pull traversal, in-hubs do not experience locality and iHTL creates vertical flipped blocks based on in-hubs (that exist) and not based on out-hubs (that may not exist).

Moreover, iHTL maintains the relative order of vertices within the VWEH and FV categories, while other locality optimizing algorithms apply degree ordering throughout [169, 174]. This destroys locality expressed in the initial assignment of vertex labels.

6.6 Conclusion & Further Applications

This chapter introduced the iHTL algorithm that improves temporal locality by applying the Uniform Memory Demands strategy. iHTL deploys the push and pull traversals for distinct subgraphs and the evaluation shows that iHTL is faster than pull and push traversals in graph processing frameworks. Furthermore, iHTL outperforms state-of-the-art locality optimization relabeling algorithms.

We consider the following suggestions to accelerate iHTL:

- The size of iHTL topology data (Section 6.4.3) can be reduced by deploying light-weight graph compression techniques [25, 26] and vectorization [159, 169].
- iHTL reduces cache misses of hubs and high-degree vertices. Locality optimizing relabeling algorithms like Rabbit-Order improve spatial locality of low-degree vertices (Section 3.4.3). This suggests that locality of the sparse block can be improved by applying Rabbit-Order.

Similar to how iHTL optimizes memory locality in the pull traversal, the push traversal can be optimized. In the push direction, outgoing edges from out-hubs form a wide range of destinations that cannot be stored in the cache and ruin the performance of push traversal. For this subgraph it is necessary to deploy the pull direction.

In other words, we can extract subgraphs containing outgoing edges from out-hubs. Then, for each of these subgraphs, we process vertices in the pull direction. Since source

vertices (out-hubs) are exponentially fewer in count than the destination vertices, random memory accesses in the pull processing of these blocks (for reading data of out-hubs) are satisfied by cache. However, we have to note that this optimization is useful for graphs that have out-hubs (Section 3.5.1).

We introduced iHTL as a traversal algorithm for all edges of the graph; however, the dense iterations of graph traversals like Connected Components, Breadth First Search, and Single Source Shortest Path can also benefit from iHTL, as in these iterations all or most edges of the graph are processed.

Chapter 7

LOTUS: Locality Optimizing Triangle Counting

7.1 Introduction

Triangle Counting (TC) is one of the fundamental problems in graph processing and is used in several fields of science, humanities, and technology [15,16,35,43,57,61,120,127, 162,164]. Different algorithms and optimizations have been proposed in the literature. However, efficient TC is still a challenge for large and fast-growing graph datasets.

In this chapter, we study the effects of skewed degree distribution of the graphs on TC. We identify the main challenges in state-of-the-art TC algorithms and opportunities for improvement, in particular: (1) poor memory locality, especially during traversal of triangles containing non-hub vertices, (2) (lack of) compactness of the graph topology representation, and (3) opportunity to prune searches that cannot uncover triangles.

Then, we use the Uniform Memory Demands strategy to design the **LOTUS**, a structure-aware and locality-optimizing TC algorithm. Lotus improves TC by distinguishing 4 types of triangles (depending on whether they include 3, 2, 1 or 0 hubs) and splits TC into multiple phases, corresponding to the types of triangle. Each TC phase is designed as a stand-alone TC algorithm, using bespoke, compact data structures. The algorithms and data structures are designed such that random memory accesses, a key challenge for memory locality in graph processing, are targeted towards a small data structure in each TC phase to utilize hardware caches in the best possible way.

This chapter is structured as follows: Section 7.2 explains key background terminology and TC algorithms. Section 7.3 presents the performance analysis of TC for

power-law graphs. Section 7.4 explains the design of Lotus and Section 7.5 introduces the Lotus algorithm which is evaluated in Section 7.6. Section 7.7 discusses further related work and avenues for future work and further applications of Lotus are presented in Section 7.8.

7.2 Prerequisites

7.2.1 Terminology

In a TC algorithm, the target is to identify the number triangles of an undirected graph (each 3 vertices that are connected). For an undirected graph $G = (V, E)$ (Section 2.1), edge (v, u) is called the symmetric edge of edge (u, v), if $u < v$. N_v is the set of neighbours of vertex v, $N_v^< = \{u \in N_v | u < v\}$, and $N_v^> = \{u \in N_v | u > v\}$.

Vertices are divided into (1) **hub** and (2) **non-hub** vertices (In Section 7.5.1, we explain how Lotus separates hubs from non-hubs). Since the graph is undirected, hubs, in-hubs, and out-hubs are the same.

An edge can be in one of 3 forms: (1) **hub to hub** edge, (2) **hub to non-hub** edge, or (3) **non-hub to non-hub** edge.

A **hub edge** is an edge with at least one hub endpoint. A **non-hub edge** is an edge without any hub as its endpoints.

A triangle is called a **hub triangle** if at least one of its vertices is a hub vertex.

7.2.2 TC Algorithms

Three main TC algorithms are summarized as the following [138]:

- The *Node iterator* algorithm enumerates each pair of neighbours of a vertex and checks if they are connected,
- The *Edge iterator* algorithm searches for common neighbours of two endpoints of each edge, and
- The *Forward* algorithm sorts vertices by their degrees in descending order and identifies common neighbours between a vertex and each of its neighbours.

Algorithm 6 is an improved version of the Forward algorithm [55] that we use as the baseline algorithm. For each vertex v and each $u \in N_v^<$, $N_v^< \cap N_u^<$ specifies the number of triangles including u and v. By limiting neighbours to $N^<$, a triangle is counted only once and the execution time is reduced as only half of the edges are processed. The

intersection is performed using merge join [138], bitmap lookup [106], hashing [41,138], or binary search [62].

Algorithm 6: Forward algorithm

Input: $G(V,\ E)$
Output: Triangles
1 $G'(V', E') =$ **reorder_by_degree**(G);
2 $triangles = 0$;
3 **par_for** $v \in V'$
4 | **par_for** $u \in N_v^<$
5 | | $triangles\ +=\ |N_v^< \cap N_u^<|$
6 **return** $triangles$;

7.3 Analysis of The Forward Algorithm for Power-Law Graphs

In this section, we investigate the implications of real-world graphs with skewed degree distribution on performance of the Forward algorithm.

7.3.1 Low Locality in Processing Non-Hub Vertices

The Forward algorithm (Algorithm 6) uses degree ordering to accelerate TC. Degree ordering improves load balance and reduces the number of comparisons occurring in the intersection operation [7].

For each edge, only one of (u, v) or (v, u) needs to be present in the graph as the symmetric edges are redundant for TC. In other words, each edge is attached to only one of its endpoints.

Degree ordering decides that (v, u) is retained if $v < u$. By consequence, only vertices v with a smaller ID ($v < u$) are stored in the neighbour list of a vertex u. As degree ordering assigns lower IDs to hubs, the neighbour list of a hub only contains hubs, and the neighbour list of a non-hub contains both hub and non-hub neighbours.

This is illustrated in an example graph (Figure 7.1) where edges have the same color as the vertex they are assigned to by degree ordering. Edges between vertex 3 and hub vertices 0 and 1 are assigned to vertex 3 and vertex 3 also has an edge to vertex 2 which is a non-hub.

An immediate consequence of this organisation is that the neighbour lists of hubs (containing hub-to-hub edges) are very frequently accessed. Indeed, hubs have many neigh-

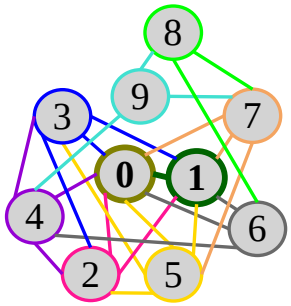


Figure 7.1: An example graph - hubs: 0, 1

bours, which by construction are mostly non-hubs. Each time a non-hub vertex v that is a neighbour of a hub u is processed, the neighbour list of the hub is accessed (Algorithm 6, Line 5). Frequent accesses to the hubs' neighbour lists prompt the processor cache to **maintain hub neighbour lists in cache**. Column 2 of Table 7.1 shows that these neighbour lists, consisting of hub-to-hub edges, include 18.1% of the edges of the graph.

The flip-side of this is that **there is a low opportunity for the cache to retain neighbour lists of non-hubs**. While each non-hub neighbour list is accessed less frequently, together they constitute 81.9% of the edges of power-law graphs (Table 7.1, Columns 3 and 5).

Dataset	Hub Edges (%)			Non-hub to Non-hub Edges (%)	Hub Triangles (%)	Relative Density of Hubs Sub-graph	Fruitless Searches (%)
	Hub to Hub	Hub to Non-hub	Total				
LJGrp	4.7	76.9	81.5	18.5	99.9	467	78.1
Twtr10	43.5	29.8	73.3	26.7	99.6	4,347	64.0
Twtr	26.3	60.3	86.6	13.4	99.7	2,627	72.2
TwtrMpi	19.1	53.5	72.7	27.4	99.4	1,911	67.8
Frndstr	6.0	25.3	31.3	68.7	47.3	600	36.9
SK	4.9	75.6	80.5	19.5	97.0	490	56.4
WbCc	37.0	35.9	72.8	27.2	99.6	3,695	47.1
UKDls	14.2	63.9	78.2	21.8	98.8	1,423	39.9
UU	12.5	61.9	74.4	25.6	96.2	1,252	31.7
UKDmn	12.8	64.9	77.7	22.3	96.6	1,279	39.1
Average	18.1	54.8	72.9	27.1	93.4	1,809	53.3

Table 7.1: Topological characteristics of hubs (1% of vertices with maximum degrees selected as hubs)

7.3.2 Lack of Compactness of Graph Topology

In graph algorithms like SpMV, BFS, SSSP, and CC, the main memory access challenge consists of random memory accesses to vertex data, which typically consists of 1–64 bits per vertex. Random memory accesses thus target a data set of size proportional to the number of vertices.

In contrast, the data accessed per vertex consists of the neighbour lists, i.e., the graph topology, in TC. Thus, **random memory accesses in TC target a much larger data set of size proportional to the number of edges**. This shows why achieving memory locality in TC is both more challenging and more important.

Driven by this comparison, we question whether it is possible to represent neighbour lists more compactly, without incurring overheads. The key to this is again in the hubs: the number of hubs is very few, but the majority of the IDs in neighbour lists of any vertex refer to hubs. In the power-law graphs used in this study, **1% of vertices are connected to 72.9% of edges** (Table 7.1, Column 4).

Drawing on the principles of coding and compression theory [77], **it is wasteful to represent highly frequently occurring IDs using the same bitwidth as rarely occurring IDs**. The penalty for doing so is inefficient cache utilization. While we reference coding theory to explain the problem, **it is important to design techniques that do not incur runtime overhead** to read graph topology data, as this is the main operation in TC.

7.3.3 Fruitless Searches

Out of the many neighbour list intersections performed, a relatively low number of triangles is found. Conservatively, all combinations need to be investigated to ensure all triangles are counted. However, there is some knowledge we can use to identify fruitless searches.

Assume there is a set of vertices $S \subset V$ where we know ahead of time that $N_v \cap S = \{\}$. Let $F = N_u \cap S$, then we know ahead of time that there exists no triangle (f, u, v), where $f \in F$ as f cannot be a neighbour of v. This follows from $N_v \cap N_u = N_v \cap (N_u \setminus F)$.

The most important S, that fits in well with the power-law graphs, is the set of hubs. As an example, in Figure 7.1 vertex 8 processes its neighbour 6 and loads its edges {0, 1, 4}. As 8 is not connected to a hub (0 or 1), it can be inferred that no triangle exists including vertices 8, 6 and any of the hubs.

In other words, **accessing hub edges cannot result in a triangle during process-**

ing non-hub vertices that have no edges to hubs ($N_v \cap Hubs = \{\}$). However, hub edges are frequently accessed in processing these non-hub vertices. We measured what fraction of accessed edges point to hubs when processing these vertices (Table 7.1, Column 8). On average, **53.3% of memory accesses are performed to hub edges that can be avoided**. This data was collected using merge join intersection. Deploying binary search intersection [62] also reduces these memory accesses (Section 7.7.3).

This shows that **if we *know* that v is not connected to a hub, it is possible to prune the search** and the power-law structure of graphs is helpful to prevent accessing a large fraction of edges in processing non-hub vertices.

7.3.4 Highly Dense Hubs Sub-graph

We have already observed that hubs are few and incident to 72.9% of the edges. Interestingly, as each vertex in a triangle must have two incident edges, these statistics become **even more skewed** when considering hubs. Column 6 of Table 7.1 ("Hub Triangles") shows the percentage of triangles containing at least one hub. It shows that, on average, **93.4% of the triangles contain at least one hub**.

This observation is an immediate result of the tight connections between hubs. We define the relative density (RD) of a sub-graph $S = (V', E')$ where $V' \subset V$ and $E' = E \cap (V' \times V')$, as $RD_S = \frac{|E'|/(|V'|^2)}{|E|/(|V|^2)}$. Column 7 of Table 7.1 reports the RD_H, where H is the set of hubs and shows that **hub-to-hub edges create a dense sub-graph that is, on average, 1809 times more dense than the full graph**.

These statistics demonstrate that (i) hubs should be central to the design of a TC algorithm; (ii) the high density of the hubs sub-graph invites a highly compact data structure to store this sub-graph.

7.4 Algorithm Design

7.4.1 Step 1: Identifying Contrasting Demands & Behaviours

In Section 7.3, we explained that mixing hub edges and non-hub edges in the topology of the graph, results in

- dedicating cache to hub-to-hub edges and depriving accesses to 80% of edges from acceleration by cache,
- inefficient usage of cache capacity, and

- fruitless searches.

7.4.2 Steps 2 and 3: Considering Potential Solutions, Matching, and Adjusting

In order to tackle these inefficiencies, we need to separate memory accesses between hub and non-hub edges. Unlike SpMV that allows separating hub edges by separating hub vertices from non-hub vertices, in TC, hub edges are accessed even in processing non-hub vertices. Therefore, we need to divide the total execution into steps based on how hub and non-hub are accessed. To that end, we distinguish 4 types of triangles:

- **HHH**: triangles between 3 hub vertices,
- **HHN**: triangles between 2 hub and 1 non-hub vertices,
- **HNN**: triangles between 1 hub and 2 non-hub vertices, and
- **NNN**: triangles between 3 non-hub vertices.

These 4 types of triangles form 3 different accesses to hub and non-hub edges during TC:

Counting HHH and HHN Triangles To count triangles with at least two hub vertices, the key question is if two hubs are connected? To check if two hubs are neighbours, we exploit two features of power-law graphs: (1) hubs have a dense connection to each other (Section 7.3.4), and (2) there is a small number of hubs (Section 7.3.2).

So, we represent the adjacency information of hubs in a dense bit array, with 1 bit per pair of hubs. As there are few hubs, the bit array can be retained in cache. Using this bit array, it is enough to iterate over all distinct pairs of hub neighbours of each vertex and to identify triangles if two hubs in a pair are connected.

Counting HNN Triangles HNN triangles have two non-hub vertices that are connected to a hub. Hence, the most frequently occurring edges, and thus the ones most frequently queried, are hub edges. As such, we organize the search around iterating non-hubs and their non-hub neighbours, and then querying if they have a common hub neighbour.

To perform HNN TC efficiently, we store the hub neighbours separately from the non-hub neighbours. Moreover, based on the observation that hubs are few but frequently mentioned (Section 7.3.2), we store the hub neighbours using fewer bits per ID.

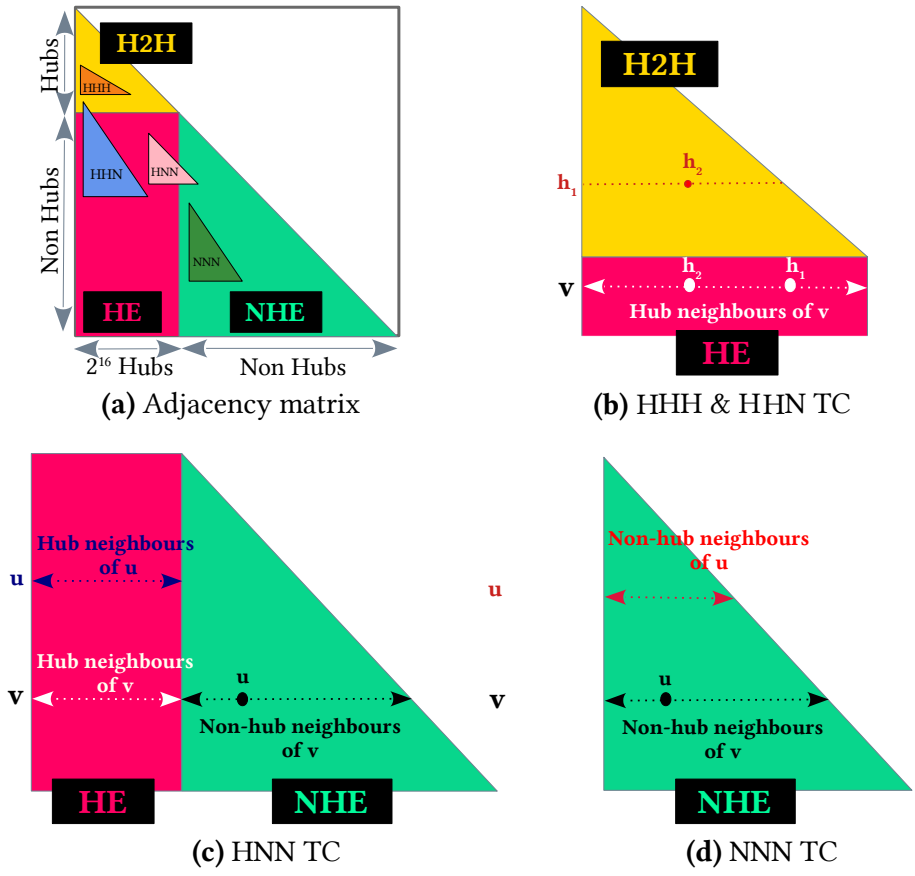
Counting NNN Triangles In this step, we identify common non-hub neighbours be-

tween a non-hub vertex and its non-hub neighbours. This avoids loading hub edges (Section 7.3.3) and only processes non-hub edges in this step as we separately store hub and non-hub edges.

7.4.3 Step 4: Merging

The distinct TC that was explained in the previous section, requires to separate hub edges from non-hub edges and to store the subgraphs separately.

7.5 LOTUS Algorithm



(a) Adjacency matrix **(b)** HHH & HHN TC

(c) HNN TC **(d)** NNN TC

Figure 7.2: Lotus adjacency matrix and TC steps

7.5.1 Lotus Graph Structure

In order to achieve the targets explained in Section 7.4.2, Lotus creates a special graph structure that consists of:

- **Number of Hubs**: Lotus selects the 64K (2^{16}) vertices with the highest degrees as hubs.
- **Hub to Hub (H2H) Bit Array**: Lotus represents hub-to-hub edges using this bit array. Since each hub has edges only to hubs with lower IDs, H2H is a triangular array (instead of a 2D square array). In this way, for hub vertices $h1$ and $h2$ where $h1 > h2 \geq 0$, the bit with index $h1(h1-1)/2 + h2$ specifies if $h1$ has an edge to $h2$.
- **Hub Edges (HE) Sub-graph**: This sub-graph represents all hub edges of the graph. It is stored in CSX format. As Lotus selects 64K hubs, each edge (neighbour ID) in HE sub-graph is represented in 16 bits. For vertex v, HE represents all hub neighbours h of v where $h < v$.
- **Non-Hub Edges (NHE) Sub-graph**: This sub-graph represents edges from each vertex v to its non-hub vertices u, where $u < v$. This sub-graph is also in CSX format, but unlike HE, NHE assigns 32 bits memory space per edge.

Figure 7.2a shows the adjacency matrix of Lotus. The 4 types of triangles are illustrated to demonstrate in which range their endpoints sit. Note that hub-to-hub edges are recorded twice: once in the HE sub-graph and once in the H2H, which overlaps HE in the figure.

7.5.2 Lotus Preprocessing

Lotus creates its graph structure in a preprocessing step before counting triangles. Algorithm 7 shows how Lotus creates its graph.

Creating Relabeling Array Lotus assigns the first consecutive IDs to hub vertices, therefore it is necessary to relabel vertices. Line 1 creates the relabeling array. The **create_relabeling_array**() function selects the hub vertices with highest degrees and assigns the first IDs to them.

In addition to hub vertices, there are a number of high-degree vertices. If they are assigned large IDs, the number of comparisons when processing NNN triangles is increased (Section 7.3). So, Lotus assigns the first consecutive IDs to 10% of vertices with the highest degrees instead of only 64K ones.

The remaining IDs are assigned to non-hub vertices in the same order as the main graph. In this way, Lotus prevents destroying the initial locality of graphs.

create_relabeling_array() returns an array that is indexed by the original ID of a vertex and the value at that index specifies the new ID of that vertex.

Algorithm 7: Lotus Preprocessing

```
   Input: G(V, E)
   Output: LotusGraph
1  RA = create_relabeling_array(G);
2  hubs_count = (1 ≪ 16);
3  TBitArray H2H(hubs_count);
4  Graph < ushort > HE;
5  Graph < uint > NHE;
6  par_for v_old ∈ V
7      v_new = RA[v_old];
8      Array < ushort > he;
9      Array < uint > nhe;
10     for u_old ∈ N_{v_old} do
11         if u_old == v_old then                          // self-edge?
12             continue;
13         u_new = RA[u_old];
14         if u_new > v_new then                       // symmetric edge?
15             continue;
16         if u_new < hubs_count then                   // hub neighbour?
17             he.push(u_new);
18             if v_new < hubs_count then          // hub neighbour of a hub?
19                 H2H.set(v_new, u_new);
20         else                                       // non-hub neighbour
21             nhe.push(u_new);
22     HE.setEdges(v_new, he);
23     NHE.setEdges(v_new, nhe);
24 return (hubs_count, H2H, HE, NHE);
```

Creating Bit Array and Sub-graphs Line 3 initializes the H2H triangular bit array storing hub-to-hub edges by allocating memory of $hubs\_count * (hubs\_count - 1)/2$ bits size and setting all bits to zero.

Lines 4 and 5 initialize sub-graphs for **HE** and **NHE** where the size of each edge is 16 and 32 bits, respectively.

Lines 6-23 process each vertex in the graph. Lines 8-9 initialize the **he** and **nhe** arrays to contain hub and non-hub neighbours of a vertex, respectively.

Each neighbour of a vertex is considered in Lines 11-21 and self-edges and symmetric edges are ignored (Lines 11-15). Similar to the baseline algorithm (Section 7.2.2), Lotus does not process symmetric edges and limits neighbours of a vertex to the ones that have lower IDs. This restricts the neighbour list of vertex v_{new} to $N^{<}_{v_{new}}$.

The neighbour is assigned to **he** (Line 17), if it is a hub neighbour. In this case, the H2H bit array is set if the vertex and its neighbour are both hubs (Line 19). If the neighbour is a non-hub vertex, it is added to **nhe** (Line 21).

After processing all edges of a vertex, Lines 22-23 call $setEdges()$ method that sorts the neighbour lists **he** and **nhe** and assigns them to the relevant vertex (v_{new}) of **HE** and **NHE** sub-graphs, respectively.

In Lines 5 and 9, 32-bit vertex ID is sufficient for public data sets as they have fewer than 2^{32} vertices. However, for datasets with greater number of vertices, 64-bit IDs can be used without losing the benefits of Lotus.

7.5.3 Counting Triangles in Lotus

Algorithm 8 shows how Lotus counts triangles:

Algorithm 8: Counting Triangles in Lotus

Input: $LotusGraph\ (hubs\_count,\ H2H,\ HE,\ NHE)$
Output: Triangles

```
1  triangles = 0;
   /* Counting HHH and HHN triangles                                  */
2  par_for v ∈ HE.V
3  |  par_for h1 ∈ HE.N_v
4  |  |  for h2 ∈ {h ∈ HE.N_v | h < h1} do
5  |  |  |  if H2H.isSet(h1, h2) then
6  |  |  |  |  triangles++;
   /* Counting HNN triangles                                          */
7  par_for v ∈ NHE.V
8  |  par_for u ∈ NHE.N_v
9  |  |  triangles += |HE.N_v ∩ HE.N_u|;
   /* Counting NNN triangles                                          */
10 par_for v ∈ NHE.V
11 |  par_for u ∈ NHE.N_v
12 |  |  triangles += |NHE.N_v ∩ NHE.N_u|;
13 return triangles;
```

HHH and HHN Lotus creates all distinct pairs between hub neighbours of a vertex (Lines 3-4) and if two hubs of a pair are connected (Line 5), a triangle has been found. Note that the bit array is laid in "$h1$-major" format, ensuring that bits for sub-

sequent $h2$ values are placed in consecutive locations. Moreover, as $h1$ changes in the outer loop on Line 3, the calculation $h1(h1-1)/2$ is reused as $h2$ changes in the inner loop in Line 4.

Figure 7.2b shows counting HHH and HHN triangles for vertex v with hub neighbours h_2 and h_1. The existence of triangle (h_2, h_1, v) is validated by checking if h_2 has an edge to h_1 in the H2H sub-graph.

HNN Lotus finds common hub neighbours between each non-hub vertex and its non-hub neighbours. Line 7 iterates over all vertices. For each non-hub vertex v, its non-hub neighbours such as u are considered (Line 8), and each common hub neighbour of u and v forms a triangle (Line 9).

In Figure 7.2c, for vertex v and its non-hub neighbours such as u (that are in NHE sub-graph), hub neighbours of u and v (that are in HE sub-graph) are matched.

NNN Lines 10–12 are similar to the Forward algorithm to find NNN triangles in the NHE. Lotus uses merge join for intersection as the neighbour lists of non-hub vertices are relatively short. This prevents overheads imposed by other solutions (Section 7.7.3).

In Figure 7.2d, for vertex v and for its non-hub neighbours such as u (that are in NHE), non-hub neighbours of u and v (that are in NHE) are matched.

7.5.4 How Does Lotus Improve Locality?

In counting HHH and HHN triangles, Lotus reads hub neighbours of a vertex in sequential accesses and iterates over all pairs of hub neighbours (Lines 3-4 of Algorithm 8). In other words, **Lotus accesses the neighbour list of a vertex only for processing that vertex**. The neighbour lists are streamed through cache. Sequentially streamed accesses are prefetched by hardware in timely fashion. Only the H2H bit array is used (Line 5) for random accesses to topology data. By concentrating random accesses on the H2H bit array, **the range of data accessed randomly is significantly reduced**.

This increases the frequency of cache hits. Table 7.5 shows that graph datasets in this study have edges with topology size of 0.42 - 12.30 Gigabytes in the CSX format, but the H2H size is less than 256 Megabytes. Moreover, **H2H stores edges in an addressable format** that facilitates efficient checking if two hubs are connected in constant time, and just a few instructions. Section 7.6.6 shows that 64 Megabytes cache space suffices to satisfy 90% of accesses to H2H.

In Algorithm 8, Lotus has two similar nested loops for counting HNN and NNN triangles in Lines 7-8 and 10-11. These loops iterate over the same domain (the neigh-

bour lists of NHE). Lotus keeps the body of these loops (intersections at Lines 9 and 12) separate (as opposed to fusing the loops). Two contradictory effects need to be traded-off:

- Random memory accesses are made to $HE.N_u$ (Line 9) and $NHE.N_u$ (Line 12). Reuse of this data before eviction from the cache is possible. If we were to fuse the loops in Lines 7–12, then reuse of this data would become less likely, as the total volume of randomly accessed data, and thus the working set size, will increase.
- The cost of traversing the NHE sub-graph itself (fetching $NHE.V$ and $NHE.N_v$) is low as this data is streamed in sequentially. The NHE topology is relatively small as it contains only 27% of edges on average (Table 7.1, Column 7).

Lotus improves locality by dividing TC into three steps and in each step dedicates cache to a smaller special data structure that is most frequently needed. Table 7.2 summarizes which data structure is accessed in random order. Section 7.6.2 shows that Lotus reduces last level cache misses by 2.1× and DTLB misses by 34.6×, on average.

TC Step	Random Accesses	Edge Size	Total Size of Edges
HHH & HHN	$H2H$	1 bit	256 Megabytes
HNN	$HE.E$	16 bits	$\|HE.E\| * 2$ Bytes
NNN	$NHE.E$	32 bits	$\|NHE.E\| * 4$ Bytes

Table 7.2: Random memory accesses in Lotus TC

7.5.5 Graph Partitioning and Load Balancing in Lotus

Edge Tiling [123] improves load balance by splitting the edge list of high-degree vertices into smaller parts and scheduling these on different concurrent threads.

In Line 3 of Algorithm 8, the amount of work each neighbour ($h1$) performs depends on its offset from the first neighbour. As a consequence, we cannot evenly divide the work between threads by assigning the same number of neighbours to each thread.

In order to parallelize the loop in Line 3 of Algorithm 8, Lotus introduces **Squared Edge Tiling** [1] that creates partitions with equal work complexity for neighbours of a vertex.

[1] While it is not the best name, to keep connection to known phrases, we used this name. We also added the "Squared" to differentiate from the main concept as we want to divide a task with a complexity of $O(|N_v|^2)$ and not $O(|N_v|)$.

For vertex v with $|N_v|$ neighbours, the total work is $|N_v| * (|N_v| - 1)/2$ and if the total work performed from the first neighbour until the i-th neighbour is f fraction of the total work, where $0 < f < 1$, then:

$$i * (i-1)/2 = f|N_v|(|N_v| - 1)/2 \ ,$$

or

$$i = (\ 1 + \sqrt{f(2|N_v| - 1)^2 + 1 - f}\ )/2 \ .$$

Since $|N_v| \gg f$,

$$\frac{i}{|N_v|} \approx \frac{1 + \sqrt{f(2|N_v| - 1)^2}}{2|N_v|} \approx \sqrt{f} \ ,$$

or

$$i \approx |N_v| * \sqrt{f} \ .$$

Using this formula, we can identify the boundaries to partition the loop by changing f. As an example, for partitioning total work for a vertex with 100 neighbours into 5 partitions, the partition borders will be 0, $100 * \sqrt{0.2} = 45$, $100 * \sqrt{0.4} = 63$, $100 * \sqrt{0.6} = 77$,$100 * \sqrt{0.8} = 89$, and 100.

While the number of triangles may vary per tile, the effort per tile is balanced. Lotus performs squared edge tiling during the preprocessing step. Values of $\sqrt{f}$ are fixed for different vertices as f indicates the fraction of work and for dividing work into p partitions, $f = \frac{k}{p}$, where $0 < k < p$. So, values of $\sqrt{f}$ are pre-calculated and reused in calculating the partition boundaries of different high-degree vertices.

Section 7.6.7 shows that squared edge tiling provides 2.7× speedup in processing HHH and HHN triangles.

7.6 Evaluation

We use the SkyLakeX and Epyc machines (Appendix A) to evaluate Lotus in comparison to:

1. BBTC[2] [167] (commit `88fe6bc`) that improves load balancing in TC through better partitioning,
2. Edge iterator in GraphGrind[3] [152] (commit `5099761`),

[2] https://github.com/GT-TDAlab/bbTC/

[3] https://github.com/DIPSA-QUB/GraphGrind

3. Forward algorithm implementation in GAP[4] [14] (commit `6ac1afd`).
4. TC of GBBS[5] [51] (commit `38964eb`) that improves [144] by parallelizing the intersection in the Forward algorithm.

All algorithms use degree ordering to accelerate TC and we report end-to-end execution time.

7.6.1 Comparison to Previous Works

Tables 7.3 compares Lotus execution time with other TC algorithms for graphs smaller than 10 billion edges. This table shows that the speedup obtained by Lotus on the Epyc architecture with 128 cores is less than on the other architectures. This is due to the total on-chip cache size. The Epyc system has two sockets with 512MB total L3 cache, which is 12 times larger than the L3 cache on the SkyLakeX machine. This large L3 cache captures a significantly higher fraction of memory accesses, and poses lesser challenges relating to memory locality. As a result, speedup obtained by Lotus is less, due to the larger cache size.

Table 7.4 shows the results of Lotus in comparison to GBBS on the Epyc machine and for graphs greater than 10 billion edges. This shows that Lotus delivers better speedups for larger graphs.

On average, Lotus is 19.3 times faster than BBTC, 5.5 times faster than GraphGrind, 3.8 times faster than GAP, and 2.2 times faster than GBBS.

7.6.2 Hardware Counters

In Section 7.5.4, we explained how Lotus improves locality. To evaluate the locality effects of Lotus, we compare the last level cache misses and DTLB misses of Lotus and Forward algorithms on the SkyLakeX machine in Figure 7.3a, and Figure 7.3b. Lotus reduces last level cache misses by up to 4.0× and on average by 2.1×. DTLB misses are also reduced by up to 56× and on average by 34.6×.

Besides improving locality, Lotus is also a more efficient algorithm throughout. Figure 7.4 compares hardware events for execution of Lotus and Forward algorithms. It shows that, on average, Lotus reduces memory accesses (load and store instructions) by 1.5×, hardware instructions by 1.7×, and branch mis-predictions by 2.4×.

[4] https://github.com/sbeamer/GAPBS
[5] https://github.com/ParAlg/gbbs/

Dataset	SkyLakeX					Epyc				
	BBTC	GGrnd	GAP	GBBS	Lotus	BBTC	GGrnd	GAP	GBBS	Lotus
LJGrp	4.1	4.7	6.4	2.5	1.0	2.4	2.5	6.6	0.5	0.8
Twtr10	62.4	74.2	32.7	32.8	6.7	31.5	21.6	45.0	9.0	4.1
Twtr	98.0	77.0	32.1	32.1	10.0	81.3	25.8	20.3	9.4	6.1
TwtrMpi	377.7	282.2	80.5	90.5	36.8	333.3	67.2	38.8	25.9	18.2
Frndstr	129.5	129.1	70.5	76.4	56.7	59.9	33.3	27.4	24.5	23.8
SK	246.3	56.5	28.8	19.5	7.3	246.5	19.5	21.0	3.3	2.9
WbCc	602.0	649.0	121.1	233.8	64.2	534.5	134.1	92.1	51.7	21.9
UKDls	-	383.3	67.7	80.0	32.7	-	58.6	89.8	38.6	12.2
UU	-	134.9	61.6	74.4	29.3	-	43.8	36.0	15.0	9.5
UKDmn	-	123.9	50.3	53.6	19.9	-	32.6	32.4	10.3	7.2
Lotus Avg. Speedup	11.3×	7.4×	3.0×	2.8×		22.1×	4.5×	5.3×	1.7×	

Table 7.3: End to end TC execution times in seconds - GGrnd: GraphGrind - Failed attempts are shown by dash - Avg. Speedup is the arithmetic mean over Lotus speedup for each dataset

Dataset	Epyc	
	GBBS	Lotus
MClst	1,415.2	784.5
ClWb12	81.7	29.9
WDC14	170.1	85.7
EU15	449.3	256.9
Lotus Avg. Speedup	2.1×	

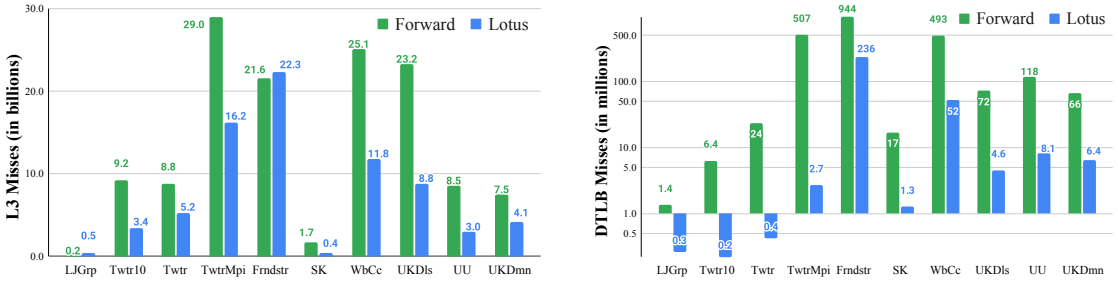
Table 7.4: End to end TC execution times in seconds

7.6.3 Execution Breakdown

Figure 7.5 displays the breakdown of Lotus execution time and shows time passed in (1) preprocessing, (2) counting HHH and HHN triangles, (3) counting HNN triangles, and (4) counting non-hub triangles.

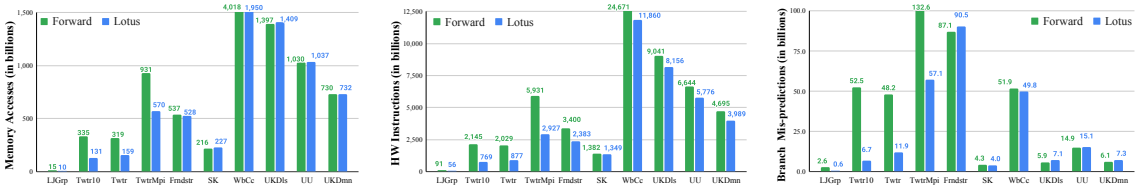
It shows that, on average, 19.4% of the total execution time is passed in preprocessing. Moreover, on average, 40.4% of the triangle counting time is passed in counting non-hub triangles.

Figure 7.6 compares the number of hub and non-hub triangles counted by Lotus. It shows that, on average, 68.9% of the triangles are counted as hub triangles in Lotus and 31.1% as non-hub triangles.



(a) Last level cache misses (b) DTLB misses (Log scale vertical axis)

Figure 7.3: Comparison of last level cache misses and DTLB misses [SkyLakeX]



(a) Memory accesses (load & store instructions) (b) Hardware instructions (c) Branch mis-predictions

Figure 7.4: Comparison of hardware events [SkyLakeX]

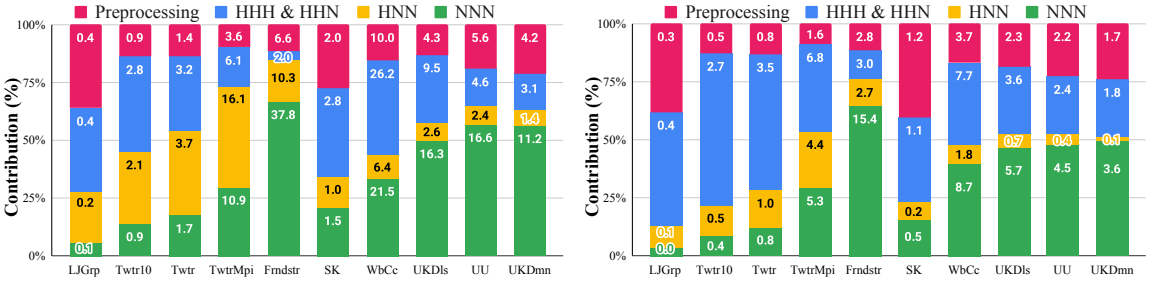
Figure 7.7 compares the number of edges in HE and NHE sub-graphs. It shows that, on average, Lotus processes 50.1% of edges as hub edges. The number of triangles and edges are different from Table 7.1 as 1% of vertices have been selected there as hubs.

7.6.4 Less Power-Law Graphs

Figures 7.7 and 7.6 show that less power-law graphs may not benefit from Lotus as other datasets. For example, the `Friendster` dataset has a relatively low skewness and the highest degree is 5K. However, Lotus selects a constant number of hubs (64K). By consequence, only 7.6% of the edges connect to these hubs and Lotus spends most of the TC time in counting non-hub triangles (Figure 7.5).

In general, less power-law graphs can be placed in two categories:

1. As we explained in Section 3.5.1, social networks with a great number of low-degree hubs where the tight connection between high-degree vertices allows improved performance by **recursively applying Lotus** and splitting the NHE sub-graph further in new H2H, HE and NHE components, similar to how iHTL ex-



(a) SkyLakeX

(b) Epyc

Figure 7.5: Lotus execution breakdown (numbers on each bar are in seconds)

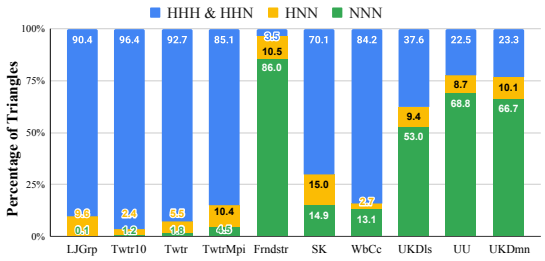


Figure 7.6: Contribution of triangles

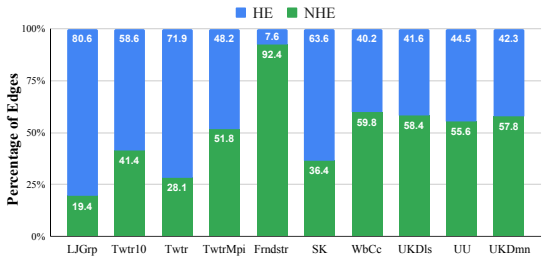


Figure 7.7: Percentage of edges in HE and NHE sub-graphs

tracts dense flipped blocks 6.

2. Graphs that have a very small number of very high-degree hubs, where the Forward algorithm is effective even without degree ordering. In processing low-degree vertices of these graphs, two types of memory accesses are performed:

   (i) Accesses to neighbour list of hub vertices that are easily maintained in the cache as hubs are rare but are accessed frequently (since they are neighbours to a great percentage of vertices), and

   (ii) Accesses to neighbour list of low-degree neighbours that a good spatial locality (which usually exists in graphs before degree reordering, especially in LWA graphs as a result of applying Layered Label Propagation [25]) makes the random memory accesses to be mostly hit in cache (as the spatial locality facilitates consecutive IDs to neighbours and necessitates consecutive processing of low-degree neighbours).

   For these graphs, it is necessary to check the degree distribution of the graph at the start of TC and to apply the Forward or edge-iterator algorithms if the graph

is not skewed enough. To that end, GAP uses the average degree of the graph and a sampling mechanism to compare the average and median degree of vertices.

7.6.5 Topology Data Size

Table 7.5 compares size of topology data in CSX format and Lotus. Since the Forward algorithm (Algorithm 6) uses only half of the edges, we have calculated the sizes of CSX edges and CSX without symmetric edges.

Dataset	CSX Edges (GB)	CSX (GB)	Lotus (GB)	Growth (%)
LJGrp	0.4	0.5	0.6	28.8
Twtr10	1.0	1.1	1.3	10.4
Twtr	1.8	2.0	1.8	-8.9
TwtrMpi	4.5	4.8	4.3	-10.8
Frndstr	6.7	7.2	7.7	6.7
SK	6.8	7.2	5.6	-21.6
WbCc	7.2	7.9	7.3	-6.8
UKDls	12.9	13.7	12.1	-11.9
UU	17.4	18.4	15.7	-14.5
UKDmn	12.3	13.1	11.5	-12.0

Table 7.5: Size of topology data (Gigabytes)

Lotus affects the size of topology data in 3 ways:

- The HE and NHE sub-graphs require an index array each, adding $8(|V|+1)$ Bytes.
- Adding the H2H bit array, of fixed size (256 Megabytes).
- Reducing the size of hub IDs, which saves 2 bytes per edge in the HE sub-graph.

For graph datasets like `SK-Domain` where Lotus collects a greater number of edges as hub edges, the topology size is reduced more as HE size is reduced.

Table 7.5 shows that, on average, **Lotus reduces size of topology data by 4.1%**. Independently of reducing size, only a subset of the topology data is accessed in each phase, resulting in smaller working sets.

7.6.6 H2H Bit Array

H2H is a dense triangular adjacency array that lists edges between a hub and its hub neighbours with lower IDs. The first column of Table 7.6 shows that the density of H2H (fraction of non-zero bits) is between 0.2% and 15.3%.

We also measured how many 64-byte aligned blocks of H2H contain 512 zero bits (Table 7.6, column 3). In web graphs, 75–95% of H2H blocks contain no edges. Edges are, thus, tightly packed in cache blocks, which implies that hubs in web graphs are mostly connected to a number of hubs as we know web graphs have a small number of hubs (Section 6.4.5). In contrast, social networks exhibit a different behavior where 5–62% of the blocks are zero that shows edges are thus more dispersed throughout H2H.

Dataset	H2H Density (%)	H2H Zero Cachelines (%)
LJGrp	0.20	62.51
Twtr10	2.83	5.72
Twtr	2.05	8.60
TwtrMpi	2.73	9.89
Frndstr	0.29	36.94
SK	1.04	91.74
WbCc	15.26	74.60
UKDls	2.56	93.31
UU	0.17	91.45
UKDmn	0.15	95.15

Table 7.6: Lotus H2H bit array characteristics

To have a better understanding of how H2H is placed in cache, we measure how many accesses to H2H are satisfied by selecting the most frequently accessed cache-lines. To this end, we sort cachelines based on how frequently they are accessed and we calculate the partial sum of their accesses.

Figure 7.8 shows that by storing one million cachelines of H2H in cache, more than 90% of accesses to H2H are satisfied. In other words, 64 Megabytes of cache space suffices to satisfy 90% of accesses to H2H.

This shows that 90% of (h_1, h_2) pairs produced in Line 5 of Algorithm 8 access only 25% of H2H cachelines. In other word, accesses to the H2H sub-graph benefit from a high level of locality.

While using a hash table can be seen as an option for implementing H2H, Figure 7.8 shows that the high level of locality in memory accesses to H2H makes it suboptimal to use a hash table for H2H. A hashing mechanism imposes more instruction count per memory access, a higher memory footprint, and a higher preprocessing time.

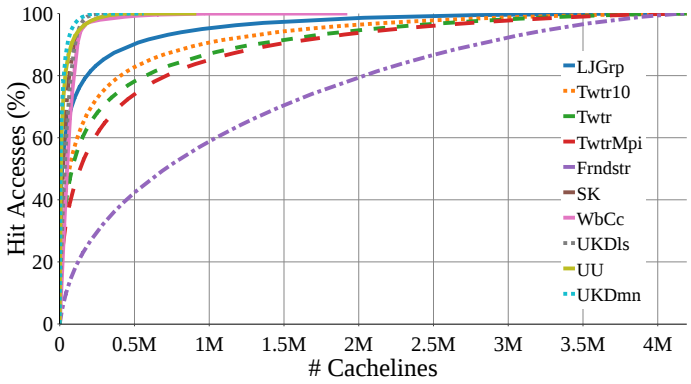


Figure 7.8: Percentage of accumulative memory accesses to most frequently accessed cachelines of H2H (M: Million)

7.6.7 Squared Edge Tiling

In Section 7.5.5, we introduced the squared edge tiling partitioning policy to provide better load balance in processing HHH and HHN triangles in Lotus algorithm. Lotus applies squared edge tiling for vertices with degree greater than 512 and divides the total work of each vertex between $p = 2 * \#threads$ partitions.

Table 7.7 shows the average idle time of threads in the first step of Lotus for two partitioning policies: edge balanced partitioning [153, 173] and squared edge tiling, when running on the SkyLakeX machine. Edge balanced divides edges into $256 * \#threads$ partitions. On average, squared edge tiling provides 2.7× speedup in processing HHH and HHN triangles.

Dataset	Edge Balanced (%)	Squared Edge Tiling (%)
Twtr10	32.1	1.0
TwtrMpi	32.6	0.7
SK	13.6	3.1
WbCc	83.3	1.3
UKDls	41.8	3.3

Table 7.7: Average idle time in percent of total execution time [SkyLakeX]

7.7 Further Related Works

7.7.1 TC History

Itai and Rodeh [81, 82] use rooted spanning tree for TC. *AYZ* algorithm [3] provides better computation complexity ($\mathbf{O}(|E|^{1.41})$) in counting triangles of sparse graphs. It uses matrix multiplication for triangles formed by high-degree vertices and for triangles made by at least one low-degree vertex, AYZ algorithm acts like the node iterator algorithm (Section 7.2.2) and finds the directed paths of length 2 and checks if their endpoints are connected by an edge.

In addition to the 3 algorithms explained in Section 7.2.2, Schank and Wagner [138] present 3 improvements:

- *Node-iterator-core* algorithm prioritizes vertices with smaller degree and removes the vertex after processing,
- *Edge-iterator-hashed* algorithm uses a hash container to identify the common neighbours of the endpoints of each node, and
- *Forward-hashed* algorithm uses a hash container for finding common neighbours.

Latapy [106] presents the *new-vertex-listing* algorithm to improve the node iterator algorithm for high-degree vertices. For each vertex, it iterates over all its neighbours and finds the common neighbours using a bitmap. Based on this, Latapy presents the *new-listing* as an improvement to the *AYZ* algorithm.

Lotus makes several benefits from these algorithms:

- Similar to AYZ and new-listing, Lotus differentiates between hub and non-hub vertices, however, Lotus counts a triangle as hub triangle if it has at least one hub vertex, as the main target of Lotus is to prevent accessing hub edges when it is not required.
- Lotus uses a bitmap array like the new-vertex-listing algorithm does. However, Lotus does not use it for presenting edges of only a vertex, but for all edges between hubs.
- Lotus has also similarities with the node-iterator-core algorithm as Lotus (1) counts triangles of hubs, (2) removes hubs and their edges from the graph (as they are not present in the NHE sub-graph), and (3) counts triangles between non-hub vertices in the NHE sub-graph.

7.7.2 Approximate and Streaming TC

Approximate and streaming TC has also been studied in the literature such as [15, 34, 84, 144, 157]. The Lotus algorithm can be used to accelerate counting hub triangles of a streaming graph and also to improve its precision.

We know hubs create a large percentage of total triangles (Sections 7.3.4 and 7.6.3) and therefore in a streaming context, Lotus stores the H2H bit array in the memory and accelerates processing of hub edges that are streamed in.

7.7.3 Improvements to TC and Forward Algorithm

Using hash maps for accelerating neighbour matching has been studied in some works such as [106, 144]. In this context, using binary search has been proposed in [62] and [70] deploys branch-free binary search [89, 93]. [76] decides between merge-based search and binary search by considering degree of vertices.

[55] improves TC by removing vertices with degree 1 (that cannot shape a triangle) from the graph and by ordering vertices of the same degree based on their connection to hub vertices. [68] reduces branch misses by using radix binning. Fast (but with more memory complexity) common neighbour counting through iterating over all wedges is studied in [4]. TC has been one of the problems pursued by the Graph Challenge and [135] surveys a number of TC studies.

7.7.4 Distributed and GPU-based TC

Distributed TC has been considered in studies such as [7, 8, 168], and GPU-based TC in [21, 50, 62, 70, 76, 167]. Patric [7] presents multiple types of partitioning for distributed TC and also a dynamic load balancing mechanism [8]. [167] studies block-based partitioning in TC. An evaluation of set intersection techniques has been studied in [17].

7.8 Conclusion and Further Applications

This chapter studied behaviours of real-world graphs with skewed degree distribution in triangle counting and explains that the large fraction of edges connected to hubs suffer from low reuse.

We introduced the *LOTUS* algorithm based on common features of power-law graphs. Lotus processes hub edges separately from non-hub edges, which allows Lotus to count

triangles in 3 steps. In each step, Lotus optimizes locality by concentrating random memory accesses on a data structure that contains more specific data in a much smaller size.

The evaluation shows that Lotus is 2.2–5.5× faster than previous works.

We propose the following extensions as future work:

- TC is the simplest form of the k-clique counting problem. We anticipate that the skewed statistics on triangles containing hubs will become even more skewed for larger cliques. So, Lotus can be applied for counting larger cliques and identifying the maximum clique algorithms of a power-law graph.
- Lotus improves locality in counting HNN triangles by reducing the size of topology data and avoiding interleaving hub and non-hub edges; however, locality of HNN may be further improved by applying blocking strategies [79] to limit domain of random accesses.
- Creating multiple HE sub-graphs may improve performance further, especially in graphs with many high-degree vertices (Section 7.6.4). It is an open question whether recognizing a higher number of distinct vertex types (two kinds of hubs and non-hubs) creates further opportunities to prune fruitless searches during HNN and NNN search.

Chapter 8

Conclusion and Future Directions

8.1 Summary

In this thesis, the main goal is to exploit the implications of the structure of real-world graph datasets with skewed-degree distribution to accelerate shared-memory graph algorithms.

In Chapter 3, we introduced structural metrics and techniques, *Cache Miss Rate Degree Distribution*, *Effective Cache Size*, *Push Locality and Pull Locality*, *N2N AID Degree Distribution*, and *Degree Range Decomposition*, to investigate the features of power-law graphs and the functionality of graph reordering algorithms for these graphs. We explained that graph relabeling suffers from inherent limitations and cannot improve locality of all vertices. We also explained that improving locality by deploying relabeling algorithms reduces the Effective Cache Size.

In Chapter 4, we introduced the **Uniform Memory Demands** strategy that states different behaviours and properties of vertex classes/subgraphs may be exploited in order to provide better performance. Instead of considering the graph in its general form, the Uniform Memory Demands strategy concentrates on investigating the similar patterns in different subgraphs in order to design data structures and algorithms that satisfy these demands with the lowest overhead that consequently optimizes performance.

We used the Uniform Memory Demands strategy to design three algorithms:

- **SAPCo Sort**: In Chapter 5, we introduced the SAPCo Sort algorithm that assigns different data structures and algorithms for different vertex classes to satisfy the contrasting memory demands of the parallel counting sort algorithm. This makes SAPCo the first parallel counting sort that is practically faster than comparison-

based sorting algorithms. SAPCo outperforms the radix sort and sample sort by 4.0 and 1.7 times speedup, respectively.

SAPCo Sort shows that we need different types of memory accesses and their race protection mechanisms; and, it is the dataset that specifies which one performs better: while private memory allocation is helpful for highly-referenced memory addresses, we skip the cost of aggregation and scattering (that grows with number of processors/partitions) by deploying atomic memory accesses. In this way, the Uniform Memory Demands strategy helps SAPCo to improve performance by enhancing work-efficiency.

- **iHTL**: In Chapter 6, we introduced the iHTL algorithm that optimizes temporal locality in processing in-hubs in a SpMV graph traversal. In processing incoming edges to in-hubs of power-law graphs, we have a small number of destinations (in-hubs); while, the source vertices are numerous. To deploy the Uniform Memory Demands strategy, we dedicate cache to the destinations instead of sources by traversing incoming edges to in-hubs in the push direction that results in 1.5–2.4 times speedup.

  Unlike SAPCo, iHTL does not benefit from improved work-efficiency as we need to merge buffers and to reset them in each iteration. However, by improving locality through concentrating memory accesses to cache, iHTL hides the cost of excess work. The excess work is in an order of $\#hubs \cdot \#threads$, but iHTL improves locality for incoming edges of hubs that are exponentially greater in count.

- **LOTUS**: In Chapter 7, we analyzed the memory accesses in Triangle Counting of power-law graphs to identify contrasting behaviours of different subgraphs. We used the Uniform Memory Demands strategy to introduce the LOTUS algorithm that divides the execution time into three steps; in each step, random memory accesses are concentrated on a small dataset that is easier to keep in cache. In this way, Lotus presents 2.2–5.5 times speedup in comparison to previous works.

  Lotus, provides better performance by improving locality and by dedicating cache to different data structures during the execution instead of inefficiently keeping a small percentage of data in cache for the whole duration of execution. Lotus also introduces the Squared Edge Tiling technique that improves load balance.

8.2 Limitations & Dependencies

We used the Uniform Memory Demands to design algorithms with better performance; however, these algorithms have their limitations:

- **Memory Overhead**. As we explained in Section 5.4, Section 6.4, and Section 7.6, SAPCo, iHTL, and Lotus may require more memory than their original algorithms. Moreover, Lotus and iHTL require a preprocessing step to create a new graph topology. While memory size is not a main concern, it can be a limiting factor, especially, in streaming graph processing.
- **Preprocessing Cost**. The iHTL and Lotus algorithms require a preprocessing step to form subgraphs. While this cost is compensated in the processing step, it is useful to find a graph topology data structure that supports different structure-aware algorithms. The first consequence of that topology data structure is removing the preprocessing time; but, more important insights (about the relation between graph algorithms and a graph topology that accelerates different graph algorithms by acknowledging the contrasting memory demands and behaviours) may be achieved as a result of that study.
- **Being Specific to An Algorithm and Processing Environment**. One advantage of relabeling algorithms is that they are independent of graph algorithms. When a graph dataset is relabelled by a reordering algorithm, we hope the relabeled version is processed faster by different graph algorithms. In contrast, the Uniform Memory Demands strategy optimizes the algorithms and not datasets. As a result, the optimization is specific to the algorithm and/or to the processing environment (shared memory, distributed memory, or GPU-based). Modifications on these two factors may impose adjustments.
- **Performance Speedup Dependency on The Architecture**. Optimizing algorithms based on the performance of different memory access types results in inherent limitation to the features of the processing environments. As an example, if we have a machine where atomic memory accesses are fast in comparison to regular memory accesses, then buffer merging in iHTL (Section 6.3.4) may require adjustment. As another example, the iHTL and Lotus algorithms facilitate speedups for a small cache that is much faster than memory accesses. Consequently, the speedup may reduce if most of random accessed data can be maintained in cache (Section 7.6.1)

or if memory access time of cache hits is similar to that of cache misses.

This shows that we need to consider the effects of the architecture on the performance as the structure-aware algorithms are experimentally designed based on the current features and efficiency of the processing environments. Changes in these factors may necessitate adjustments in the algorithms in order to deliver the best performance.

8.3 Suggestions for Future Work

We have explained the direct usage of techniques introduced in each chapter in its last section. In this section, we consider the future directions from a more general perspective:

- **Studying Other Algorithms & Other Processing Environments**. In this thesis, we showed that the structure of graph datasets is an effective factor to accelerate shared-memory graph algorithms. However, contrasting demands and behaviours in graph datasets may be used to accelerate graph processing in other environments based on the main performance bottleneck of those environments.

  In Section 2.4.3, we reviewed some partitioning algorithms that exploit the structure of the power-law graphs; however, more investigation is required. In distributed-memory graph processing, communication overhead and load-balance heavily affect the performance. By investigating the effects of the structure of power-law graphs on these factors, we achieve new insights for designing structure-aware algorithms for distributed-memory environments.

  By deploying GPUs, we are equipped with a high number of processors that is more challenging to load-balance. On the other hand, the caching system is different from CPU-based graph processing and the GPU memory is much smaller than CPU memory. So, it is more important to reduce GPU-CPU data transfer and to use the GPU cache efficiently by studying the effects of power-law graphs and designing structure-aware algorithms that improve memory locality.

  Some techniques introduced in this thesis may directly be applicable in distributed and/or GPU-based computing. As an example, iHTL and Lotus provide finer granularity by dividing the total work into multiple steps; this provides better load-balance in distributed and GPU environments.

- **Investigating Other Features of The Power-Law Graphs**. The Uniform Memory Demands strategy works, mainly, based on the skewed degree distribution of power-law graphs. However, these graphs have other unique features that can be exploited to accelerate graph algorithms.

  As an example, power-law graphs have a special connectivity as the giant component in these graphs contains a very large percentage of vertices and edges. This feature has been used in the Thrifty Label Propagation algorithm [97] to introduce a Connected Components algorithm that traverses only 1.4% of edges as a result of not processing vertices connected to the giant component. Similarly, the MASTIFF algorithm [99] exploits the connectivity of power-law graphs to introduce a Minimum Spanning Forest algorithm that skips processing vertices in the giant component and by allowing other vertices to attach themselves to the giant component.

- **Improving The Requirements of Graph Processing Studies**. In this thesis, we have used publicly available real-world power-law graph datasets that include web graphs and social networks. However, this study can be extended by investigating other graph datasets to purify the insights and to find more realistic analysis.

  On the other hand, most of real-world datasets are old (e.g., the latest Twitter graph is from 2010), and synthetic datasets do not represent the real-world graphs (in terms of structural properties of the graph such as degree-distribution skewedness and degree decomposition) well. So, a study on differences of synthetic and real-world graphs will result in better synthetic datasets, especially by deploying new metrics and novel features and differences between real-world graph types that were introduced in Section 3.5.

  We need more graph-specific simulation techniques and software to be able to investigate different aspects of execution of a graph algorithm. The large overhead of current simulation techniques (that is usually performed on a single machine) prevents simulating large graphs and complicated graph algorithms through cycle-accurate micro-architectural simulation.

- **Structure-Aware Approximation**. The results of this thesis open new perspectives on approximate and transprecise graph processing [16, 160]: Do different vertex classes and/or subgraphs require different precision and/or convergence criteria? Is it possible to use the structure of graphs to improve accuracy or performance?

- **Structure-Aware Automatic Code Optimization**. The Uniform Memory Demands strategy can be used to improve automatic code optimization: How can we use this strategy to automatically generate optimized code by (i) identifying the contrasting memory behaviours of different parts of the datasets (vertices, edges, or sub-graphs), and by (ii) deploying suitable mechanisms to satisfy the memory demands of each part without sacrificing performance?

- **Auxiliary Data Structure**. The preprocessing step in the iHTL and Lotus algorithms includes a graph relabeling process that changes the IDs of vertices and their neighbours. This shows that we need to create a more-structured data from the amorphous data structure of the raw graph. In other words, we use an auxiliary data structure to accelerate the graph processing.

  Now, the question is how to create an optimal auxiliary data structure with the lowest memory/time overhead while preserving the Uniform Memory Demands strategy? Is it necessary to embed data from the whole graph into this data structure? How to create this auxiliary data structure without relabeling the whole graph? Can we create an auxiliary data structure that accelerates a family of graph algorithms?

Appendix A

Experimental Setup

A.1 Machines

For the experiments in this thesis, we use 2 machines, listed in Table A.1. The machines use CentOS 7.

	SkyLakeX	**SkyLakeX-2**	**Epyc**
CPU Model	Intel Xeon Gold 6130	Intel Xeon Gold 6126	AMD Epyc 7702
CPU Frequency	3.7 / 2.10 GHz	3.7 / 2.6 GHz	3.35 / 2.0 GHz
Sockets	2	2	2
NUMA Nodes	2	2	8
Total CPU Cores	32	24	128
Hyperthreading	No	No	No
Total Threads	32	24	128
L1 Cache	32 KB / 1 core	32 KB / 1 core	32 KB / 1 core
L2 Cache	1 MB / 1 core	1 MB / 1 core	512 KB / 1 core
L3 Cache	22 MB / 16 cores	20 MB / 12 cores	16 MB / 4 cores
Total L3 Cache	44 MB	40 MB	512 MB
Total Memory	768 GB	1,584 GB	2,048 GB

Table A.1: Machines

Dataset	Name	Type	Source	$\|\mathbf{V}\|$ (M)	$\|\mathbf{E}\|$ (B)	$\|\mathbf{E}_{Sym}\|$ (B)
WWiki	War Wikipedia	KG	KN	2	0.04	0.05
LJ	LiveJournal Links	SN	KN	5	0.10	0.10
LJGrp	LiveJournal	SN	KN	7	0.22	0.22
Twtr10	Twitter 2010	SN	NR	21	0.26	0.53
Twtr	Twitter	SN	NR	28	0.53	0.96
WebB	WebBase-2001	WG	LWA	114	1.02	1.71
TwtrMpi	Twitter-MPI	SN	NR	41	1.47	2.41
Frndstr	Friendster	SN	NR	65	1.81	3.61
SK	SK-Domain	WG	LWA	50	1.95	3.64
WbCc	Web-CC12	WG	NR	89	2.04	3.87
UKDls	UK-Delis	WG	LWA	110	3.94	6.92
UU	UK-Union	WG	LWA	133	5.51	9.36
UKDmn	UK-Domain	WG	KN	105	6.60	6.60
ClWb9	ClueWeb09	WG	NR	1.7K	7.9	15.6
ClWb12	ClueWeb12	WG	LWA	978	42.6	74.7
UK14	UK-2014	WG	LWA	787	47.6	84.9
WDC14	WDC 2014	WG	WDC	1,724	64.4	123.8
EU15	EU Domains	WG	LWA	1,071	91.8	161.1

Table A.2: Datasets

A.2 Datasets

Table A.2 shows the datasets and their sources: "*Konect*"[1] (KN) [23, 103, 121], "*NetworkRepository*"[2] (NR) [26, 36, 39, 131, 147], "*Laboratory for Web Algorithmics*"[3] (LWA) [23–26, 104], and "*Web Data Commons*"[4] (WDC) [107, 118, 119]. Graph types are Knowledge Graph (KG), Social Network (SN), and Web Graph (WG).

In Table A.2, the numbers of vertices ($|\mathbf{V}|$) are in millions, and the numbers of edges of the graph ($|\mathbf{E}|$) are in billions, counted after removing zero-degree vertices. The column $|\mathbf{E}_{Sym}|$ shows the numbers of edges of the symmetric graphs in billions. We use the CSX representation for the undirected graphs undirected graphs (Section 2.1), and in this representation, each edge of the graph appears two times (once in the neighbour

[1] http://konect.cc
[2] http://networkrepository.com
[3] http://law.di.unimi.it
[4] http://webdatacommons.org/hyperlinkgraph/

list of each endpoint).

As graph datasets have fewer than 2^{32} vertices, 4 Bytes IDs are assigned to vertices and each element of the edges array has a size of 4 Bytes. The graphs in this study have up to hundreds of billions edges and we use 8 Bytes for indexing the edges array. So, each element of the offsets array has a size of 8 Bytes.

A.3 Implementation and Source Code

We have implemented our algorithms in the `C` language using the `OpenMP` API [46] and `pthread`. We also use `libnuma`, and `papi` [156] libraries. We use the interleaved NUMA memory policy and the `gcc-9.2` used as compiler with `-O3` flag.

The source code and further discussions are available online on https://blogs.qub.ac.uk/GraphProcessing/LaganLighter.

References

[1] Emily E Ackerman, John F Alcorn, Takeshi Hase, and Jason E Shoemaker. A dual controllability analysis of influenza virus-host protein-protein interaction networks for antiviral drug target discovery. *BMC bioinformatics*, 20(1):1–13, 2019. doi:10.1186/s12859-019-2917-z.

[2] Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. Improved external memory bfs implementations. In *Proceedings of the Meeting on Algorithm Engineering and Expermiments*, page 3–12, USA, 2007. Society for Industrial and Applied Mathematics. URL: https://dl.acm.org/doi/10.5555/2791188.2791189.

[3] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997. doi:10.1007/BF02523189.

[4] Xiaojing An, Kasimir Gabert, James Fox, Oded Green, and David A. Bader. Skip the intersection: Quickly counting common neighbors on shared-memory systems. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, USA, 2019. IEEE. doi:10.1109/HPEC.2019.8916307.

[5] Renzo Angles. A comparison of current graph database models. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, pages 171–177, 2012. doi:10.1109/ICDEW.2012.31.

[6] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31, USA, 2016. IEEE. doi:10.1109/IPDPS.2016.110.

[7] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*, CIKM '13, page 529–538, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2505515.2505545.

[8] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1839–1847, USA, 2015. IEEE. doi:10.1109/BigData.2015.7363957.

[9] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Trans. Parallel Comput.*, 9(1), jan 2022. doi:10.1145/3505286.

[10] Vignesh Balaji and Brandon Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 203–214, 2018. doi:10.1109/IISWC.2018.8573478.

[11] Reet Barik, Marco Minutoli, Mahantesh Halappanavar, Nathan R. Tallent, and Ananth Kalyanaraman. Vertex reordering for real-world graphs and applications: An empirical evaluation. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 240–251, 2020. doi:10.1109/IISWC50251.2020.00031.

[12] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–386, 2019. doi:10.1109/HPCA.2019.00051.

[13] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press. doi:10.1109/SC.2012.50.

[14] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619:1–16, 2015. arXiv:1508.03619.

[15] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, page 16–24, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1401890.1401898.

[16] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient algorithms for large-scale local triangle counting. *ACM Trans. Knowl. Discov. Data*, 4(3), October 2010. doi:10.1145/1839490.1839494.

[17] Christos Bellas and Anastasios Gounaris. Exploiting gpus for fast intersection of large sets. *Information Systems*, page 101992, 2022. doi:https://doi.org/10.1016/j.is.2022.101992.

[18] Jonathan W. Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–14, 2007. doi:10.1109/IPDPS.2007.370685.

[19] Kristof Beyls and Erik H. D'Hollander. Reuse distance as a metric for cache behavior. In *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, pages 617–662, 2001. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.8798.

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011. doi:10.1145/2024716.2024718.

[21] Mauro Bisson and Massimiliano Fatica. Static graph challenge on gpu. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, USA, 2017. IEEE. doi:10.1109/HPEC.2017.8091034.

[22] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, oct 2008. doi:10.1088/1742-5468/2008/10/p10008.

[23] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Softw. Pract. Exper.*, 34(8):711–726, July 2004. doi:10.1002/spe.587.

[24] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. Bubing: Massive crawling for the masses. *ACM Trans. Web*, 12(2), June 2018. doi:10.1145/3160017.

[25] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 587–596, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1963405.1963488.

[26] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, page 595–602, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/988672.988752.

[27] Stephen P. Borgatti, Ajay Mehra, Daniel J. Brass, and Giuseppe Labianca. Network analysis in the social sciences. *Science*, 323(5916):892–895, 2009. doi:10.1126/science.1165821.

[28] Gergana Bounova and Olivier De Weck. Overview of metrics and their correlation patterns for multiple-metric topology analysis on heterogeneous graph ensembles. *Physical Review E*, 85(1):016117, 2012. doi:10.1103/PhysRevE.85.016117.

[29] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998. Proceedings of the Seventh International World Wide Web Conference. doi:10.1016/S0169-7552(98)00110-X.

[30] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. Engineering a cache-oblivious sorting algorithm. *ACM J. Exp. Algorithmics*, 12, jun 2008. doi:10.1145/1227161.1227164.

[31] Anna D Broido and Aaron Clauset. Scale-free networks are rare. *Nature communications*, 10(1):1–10, 2019. doi:10.1038/s41467-019-08746-5.

[32] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2063384.2063471.

[33] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, June 1997. doi:10.1145/268806.268810.

[34] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, page 253–262, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1142351.1142388.

[35] Ronald S Burt. Structural holes and good ideas. *American journal of sociology*, 110(2):349–399, 2004. doi:10.1086/421787.

[36] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *ICWSM*, volume 14, Washington DC, USA, May 2010. URL: https://ojs.aaai.org/index.php/ICWSM/article/view/14033.

[37] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 444–453, New York, NY, USA, 1999. ACM. doi:10.1145/305138.305231.

[38] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2741948.2741970.

[39] Charles L Clarke, Nick Craswell, and Ian Soboroff. Overview of the trec 2009 web track. Technical report, DTIC Document, 2009. URL: https://apps.dtic.mil/sti/citations/ADA517817.

[40] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3):45–47, March 2004. doi:10.1145/971617.971643.

[41] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11:29–42, 2009. doi:10.1109/MCSE.2009.120.

[42] Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. *ACM Trans. Parallel Comput.*, 3(4), mar 2017. doi:10.1145/3040221.

[43] James S. Coleman. Social capital in the creation of human capital. *American Journal of Sociology*, 94:S95–S120, 1988. URL: https://www.jstor.org/stable/2780243.

[44] Guojing Cong and Tong Wen. Locality behavior of parallel and sequential algorithms for irregular graph problems. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS '07, page 391–397, USA, 2007. ACTA Press. URL: https://dl.acm.org/doi/10.5555/1647539.1647611.

[45] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172, New York, NY, USA, 1969. Association for Computing Machinery. doi:10.1145/800195.805928.

[46] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. doi:10.1109/99.660313.

[47] Antonio del Sol, Hirotomo Fujihashi, and Paul O'Meara. Topology of small-world networks of protein–protein complex structures. *Bioinformatics*, 21(8):1311–1315, 01 2005. doi:10.1093/bioinformatics/bti167.

[48] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. *Dynamic Graph Algorithms*, volume 1, chapter 9. CRC Press, 2009. doi:10.1201/9781584888239-c9.

[49] Peter J. Denning and Craig H. Martell. *Great Principles of Computing*. The MIT Press, 2015. URL: https://mitpress.mit.edu/9780262527125/great-principles-of-computing/.

[50] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Multi-threaded sparse matrix-matrix multiplication for many-core and GPU architectures. *CoRR*, abs/1801.03065:24, 2018. arXiv:1801.03065.

[51] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Trans. Parallel Comput.*, 8(1), April 2021. doi:10.1145/3434393.

[52] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 229–241, 1999. doi:10.1145/301631.301670.

[53] Chen Ding and Yutao Zhong. Reuse distance analysis, 2001. URL: http://hdl.handle.net/1802/415.

[54] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5):245–257, May 2003. doi:10.1145/781131.781159.

[55] Evan Donato, Ming Ouyang, and Cristian Peguero-Isalguez. Triangle counting with a multi-core computer. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, USA, 2018. IEEE. doi:10.1109/HPEC.2018.8547540.

[56] Bin Dong, Surendra Byna, and Kesheng Wu. SDS-Sort: Scalable dynamic skew-aware parallel sorting. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, page 57–68, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2907294.2907300.

[57] Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the National Academy of Sciences of the United States of America*, 99:5825–5829, 2002.

[58] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, 2012. doi:10.1109/HPEC.2012.6408680.

[59] Priyank Faldu, Jeff Diamond, and Boris Grot. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–13. IEEE, 2019. doi:10.1109/IISWC47752.2019.9041948.

[60] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. On identifying strongly connected components in parallel. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, page 505–511, Berlin, Heidelberg, 2000. Springer-Verlag. doi:10.1007/3-540-45591-4_68.

[61] Brooke Foucault Welles, Anne Van Devender, and Noshir Contractor. *Is a Friend a Friend? Investigating the Structure of Friendship Networks in Virtual Worlds*, page 4027–4032. Association for Computing Machinery, New York, NY, USA, 2010. doi:10.1145/1753846.1754097.

[62] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A. Bader. Fast and adaptive list intersections on the gpu. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, USA, 2018. IEEE. doi:10.1109/HPEC.2018.8547759.

[63] W. D. Frazer and A. C. McKellar. SampleSort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, jul 1970. doi:10.1145/321592.321600.

[64] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. High-performance graph algorithms from parallel sparse matrices. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 260–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-75755-9_32.

[65] John R Gilbert, Steve Reinhardt, and Viral B Shah. A unified framework for numerical and combinatorial computing. *Computing in Science & Engineering*, 10(2):20–25, 2008. doi:10.1109/MCSE.2008.45.

[66] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proc. VLDB Endow.*, 13(8):1304–1318, April 2020. doi:10.14778/3389133.3389145.

[67] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez.

[68] Oded Green, James Fox, Alex Watkins, Alok Tripathy, Kasimir Gabert, Euna Kim, Xiaojing An, Kumar Aatish, and David A. Bader. Logarithmic radix binning and vectorized triangle counting. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, USA, 2018. IEEE. doi:10.1109/HPEC.2018.8547581.

[69] Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 423–437, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1094811.1094844.

[70] Chuangyi Gui, Long Zheng, Pengcheng Yao, Xiaofei Liao, and Hai Jin. Fast triangle counting on gpu. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, USA, 2019. IEEE. doi:10.1109/HPEC.2019.8916216.

[71] Hwansoo Han and Chau-Wen Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, 2006. doi:10.1109/TPDS.2006.88.

[72] F. Harary and G. Gupta. Dynamic graph models. *Math. Comput. Model.*, 25(7):79–87, apr 1997. doi:10.1016/S0895-7177(97)00050-2.

[73] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 27–38, New York, NY, USA, 2014. ACM. doi:10.1145/2628071.2628091.

[74] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 3–12, New York, NY, USA, 2011. ACM. doi:10.1145/1941553.1941557.

[75] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 78–88, 2011. doi:10.1109/PACT.2011.14.

[76] Yang Hu, Hang Liu, and H. Howie Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182, USA, 2018. IEEE. doi:10.1109/SC.2018.00017.

[77] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi:10.1109/JRPROC.1952.273898.

[78] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications*, 18(1):135–158, 2004. arXiv:10.1177/1094342004041296.

[79] Eun-Jin Im and Katherine A Yelick. Optimizing sparse matrix vector multiplication on smp. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1999*, USA, 1999. SIAM. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.8485.

[80] Francois Irigoin and Rémi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 319–329, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/73560.73588.

[81] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, page 1–10, New York, NY, USA, 1977. Association for Computing Machinery. doi:10.1145/800105.803390.

[82] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978. doi:10.1137/0207033.

[83] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 60–71, New York, NY, USA, 2010. ACM. doi:https://doi.org/10.1145/1815961.1815971.

[84] Madhav Jha, C. Seshadhri, and Ali Pinar. A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. *ACM Trans. Knowl. Discov. Data*, 9(3), February 2015. doi:10.1145/2700395.

[85] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In Rajiv Gupta, editor, *Compiler Construction*, pages 264–282, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-11970-5_15.

[86] U Kang, Duen Horng Chau, and Christos Faloutsos. Mining large graphs: Algorithms, inference, and discoveries. In *2011 IEEE 27th International Conference on Data Engineering*, pages 243–254, 2011. doi:10.1109/ICDE.2011.5767883.

[87] U. Kang, Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. Hadi: Mining radii of large graphs. *ACM Trans. Knowl. Discov. Data*, 5(2), feb 2011. doi:10.1145/1921632.1921634.

[88] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *2009 Ninth IEEE International Conference on Data Mining*, pages 229–238, 2009. doi:10.1109/ICDM.2009.14.

[89] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching. *ACM J. Exp. Algorithmics*, 22, May 2017. doi:10.1145/3053370.

[90] Ian P. King. An automatic reordering scheme for simultaneous equations derived from network systems. *International Journal for Numerical Methods in Engineering*, 2(4):523–533, 1970. doi:10.1002/nme.1620020406.

[91] Philip S. Kitcher et al. Eliminativism and falsification. Britannica Encyclopaedia, https://www.britannica.com/topic/philosophy-of-science/Eliminativism-and-falsification, 1998. [Online; accessed September 2022].

[92] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, sep 1999. doi:10.1145/324133.324140.

[93] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. URL: https://dl.acm.org/doi/10.5555/280635.

[94] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. Exploiting in-hub temporal locality in SpMV-based graph processing. In *50th International Conference on Parallel Processing*, ICPP 2021, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3472456.3472462.

[95] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. How do graph relabeling algorithms improve memory locality? In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 84–86, USA, 2021. IEEE Computer Society. doi:10.1109/ISPASS51385.2021.00023.

[96] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. Locality analysis of graph reordering algorithms. In *2021 IEEE International Symposium on Workload Characterization (IISWC'21)*, pages 101–112, USA, 2021. IEEE Computer Society. doi:10.1109/IISWC53511.2021.00020.

[97] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. Thrifty Label Propagation: Fast connected components for skewed-degree graphs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 226–237, USA, 2021. IEEE Computer Society. doi:10.1109/Cluster48925.2021.00042.

[98] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. LOTUS: Locality optimizing triangle counting. In *27th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 2022)*, page 219–233, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503221.3508402.

[99] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. MASTIFF: Structure-aware minimum spanning tree/forest. In *Proceedings of the 36th ACM International Conference on Supercomputing*, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3524059.3532365.

[100] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. SAPCo Sort: Optimizing degree-ordering for power-law graphs. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 2022. doi:10.1109/ISPASS55109.2022.00015.

[101] Valdis E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2001. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.2612.

[102] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014. doi:10.1137/130930352.

[103] Jérôme Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, page 1343–1350, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2487788.2488173.

[104] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1772690.1772751.

[105] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola.

[106] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1):458–473, 2008. doi:0.1016/j.tcs.2008.07.017.

[107] Oliver Lehmberg, Robert Meusel, and Christian Bizer. Graph structure in the web: Aggregated by pay-level domain. In *Proceedings of the 2014 ACM Conference on Web Science*, WebSci '14, page 119–128, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2615569.2615674.

[108] Yongsub Lim, U Kang, and Christos Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3077–3089, Dec 2014. doi:10.1109/TKDE.2014.2320716.

[109] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, page 52–55, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1837274.1837289.

[110] Damien Magoni. Nem: a software for network topology analysis and modeling. In *Proceedings. 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pages 364–371, 2002. doi:10.1109/MASCOT.2002.1167097.

[111] Priya Mahadevan, Dmitri Krioukov, Kevin Fall, and Amin Vahdat. Systematic topology analysis and generation using degree correlations. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, page 135–146, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1159913.1159930.

[112] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010. doi:10.1145/1807167.1807184.

[113] Dániel Marx. Graph colouring problems and their applications in scheduling. *Periodica Polytechnica Electrical Engineering (Archives)*, 48(1-2):11–16, 2004. URL: https://www.cs.bme.hu/~dmarx/papers/marx-pp.pdf.

[114] Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, may 2014. doi:10.1145/2627692.2627694.

[115] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *Intl. Journal of Parallel Programming*, pages 217–247, 2001. doi:10.1023/A:1011119519789.

[116] Mohd Khairul Anam Che Mentri. Descartes and popper on the foundations of knowledge. Master's thesis, University of Malaya, 2011. URL: http://studentsrepo.um.edu.my/3917/.

[117] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, February 2012. doi:10.1145/2370036.2145832.

[118] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph structure in the web — revisited: A trick of the heavy tail. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, page 427–432, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2567948.2576928.

[119] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the web – analyzed on different aggregation levels. *The Journal of Web Science*, 1(1):33–47, 2015. doi:10.1561/106.00000003.

[120] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002. doi:10.1126/science.298.5594.824.

[121] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, page 29–42, New York, NY, USA, 2007. ACM. doi:10.1145/1298306.1298311.

[122] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2522738.

[123] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2522739.

[124] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010. doi:10.1109/SC.2010.34.

[125] Juan C. Pichel, David E. Singh, and Jesús Carretero. Reordering algorithms for increasing locality on multicore processors. In *2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 123–130, 2008. doi:10.1109/HPCC.2008.96.

[126] Karl Popper. *The logic of scientific discovery*. Hutchinson, 1959.

[127] Alejandro Portes. Social capital: Its origins and applications in modern sociology. *Annual review of sociology*, 24(1):1–24, 1998. doi:10.1146/annurev.soc.24.1.1.

[128] David M W Powers. Parallelized quicksort with optimal speedup. Technical report, Universitt Kaiserslautern, 2007. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.7281.

[129] Murali K. Pusala, Mohsen Amini Salehi, Jayasimha R. Katukuri, Ying Xie, and Vijay Raghavan. *Massive Data Analysis: Tasks, Tools, Applications, and Challenges*, pages 11–40. Springer India, New Delhi, 2016. doi:10.1007/978-81-322-3628-3_2.

[130] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. Scalable big graph processing in mapreduce. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 827–838, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2588555.2593661.

[131] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 4292–4293, USA, 2015. AAAI Press. URL: https://dl.acm.org/doi/10.5555/2888116.2888372.

[132] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2522740.

[133] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.3853.

[134] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003. doi:10.1137/1.9780898718003.

[135] Siddharth Samsi, Jeremy Kepner, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Albert Reuther, Steven Smith, William Song, Diane Staheli, and Paul Monticciolo. Graphchallenge.org triangle counting performance. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, USA, 2020. IEEE. doi:10.1109/HPEC43674.2020.9286166.

[136] Peter Sanders and Rudolf Fleischer. Asymptotic complexity from experiments? a case study for randomized algorithms. In Stefan Näher and Dorothea Wagner, editors, *Algorithm Engineering*, pages 135–146, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-44691-5_12.

[137] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 979–990, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2588555.2610518.

[138] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In Sotiris E. Nikoletseas, editor, *Experimental and Efficient Algorithms*, pages 606–609, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11427186_54.

[139] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*, page 319–333, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-319-43659-3_24.

[140] Harold H. Seward. Internal sorting by floating digital sort. Master's thesis, Massachusetts Institute of Technology, 1954. URL: http://bitsavers.org/pdf/mit/whirlwind/R-series/R-232_Information_Sorting_in_the_Application_of_Electronic_Digital_Computers_to_Business_Operations_May54.pdf.

[141] Muhammad Akram Shaikh, Jiaxin Wang, Zehong Yang, and Yixu Song. Graph structural mining in terrorist networks. In Reda Alhajj, Hong Gao, Jianzhong Li, Xue Li, and Osmar R. Zaïane, editors, *Advanced Data Mining and Applications*, pages 570–577, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-73871-8_54.

[142] Hiroaki Shiokawa, Tomokatsu Takahashi, and Hiroyuki Kitagawa. Scalescan: Scalable density-based graph clustering. In Sven Hartmann, Hui Ma, Abdelkader Hameurlain, Günther Pernul, and Roland R. Wagner, editors, *Database and Expert Systems Applications*, pages 18–34, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-98809-2_2.

[143] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013. doi:10.1145/2517327.2442530.

[144] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160, USA, 2015. IEEE. doi:10.1109/ICDE.2015.7113280.

[145] Scott W. Sloan. An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering*, 23(2):239–251, 1986. doi:10.1002/nme.1620230208.

[146] George M. Slota, Cameron Root, Karen Devine, Kamesh Madduri, and Sivasankaran Rajamanickam. Scalable, multi-constraint, complex-objective graph partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 31(12):2789–2801, 2020. doi:10.1109/TPDS.2020.3002150.

[147] Friendster social network. Friendster: The online gaming social network, 2011. URL: https://archive.org/details/friendster-dataset-201107.

[148] Andrew Sohn and Yuetsu Kodama. Load balanced parallel radix sort. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, page 305–312, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/277830.277903.

[149] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. GraphR: Accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543, 2018. doi:10.1109/HPCA.2018.00052.

[150] Michelle Mills Strout and Paul D. Hovland. Metrics and models for reordering transformations. In *Proceedings of the 2004 Workshop on Memory System Performance*, MSP '04, page 23–34, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1065895.1065899.

[151] Bor-Yiing Su, Tasneem G. Brutch, and Kurt Keutzer. Parallel bfs graph traversal on images using structured grid. In *2010 IEEE International Conference on Image Processing*, pages 4489–4492, 2010. doi:10.1109/ICIP.2010.5652307.

[152] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Accelerating graph analytics by utilising the memory locality of graph partitioning. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 181–190, USA, 2017. ACM. doi:10.1109/ICPP.2017.27.

[153] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3079079.3079097.

[154] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. VEBO: A vertex- and edge-balanced ordering heuristic to load balance parallel graph processing. *CoRR*, abs/1806.06576:1–13, 2018. arXiv:1806.06576.

[155] Michael Sutton, Tal Ben-Nun, and Amnon Barak. Optimizing parallel graph connectivity computation via subgraph sampling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 12–21. IEEE, 2018. doi:10.1109/IPDPS.2018.00012.

[156] Daniel Terpstra, Heike Jagode, Haihang You, and Jack J. Dongarra. Collecting performance data with PAPI-C. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 157–173. Springer, 2009. doi:10.1007/978-3-642-11261-4_11.

[157] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, page 837–846, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1557019.1557111.

[158] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, aug 1990. doi:10.1145/79173.79181.

[159] Hans Vandierendonck. Graptor: Efficient pull and push style vectorized graph processing. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. ACM. doi:10.1145/3392717.3392753.

[160] Hans Vandierendonck. Software-defined floating-point number formats and their application to graph processing. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3524059.3532360.

[161] Youwei Wang, Weihui Dai, and Yufei Yuan. Website browsing aid: A navigation graph-based recommendation system. *Decis. Support Syst.*, 45(3):387–400, June 2008. doi:10.1016/j.dss.2007.05.006.

[162] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684):440–442, 1998. doi:10.1038/30918.

[163] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1813–1828, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2882903.2915220.

[164] Howard T Welser, Eric Gleave, Danyel Fisher, and Marc Smith. Visualizing the signatures of social roles in online discussion groups. *Journal of social structure*, 8(2):1–32, 2007. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.352.3618.

[165] Stefan Wuchty. Controllability in protein interaction networks. *Proceedings of the National Academy of Sciences*, 111(19):7156–7160, 2014. doi:10.1073/pnas.1311231111.

[166] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. Scan: A structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 824–833, New York, NY, USA, 2007. ACM. doi:10.1145/1281192.1281280.

[167] Abdurrahman Yasar, Sivasankaran Rajamanickam, Jonathan W. Berry, and Ümit V. Çatalyürek. A block-based triangle counting algorithm on heterogeneous environments. *CoRR*, abs/2009.12457:1–13, 2020. arXiv:2009.12457.

[168] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V. Çatalyürek. Fast triangle counting using cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, USA, 2018. IEEE. doi:10.1109/HPEC.2018.8547563.

[169] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. Speeding up spmv for power-law graph analytics by enhancing locality & vectorization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. URL: https://dl.acm.org/doi/10.5555/3433701.3433815.

[170] Haiyuan Yu, Philip M Kim, Emmett Sprecher, Valery Trifonov, and Mark Gerstein. The importance of bottlenecks in protein networks: correlation with gene essentiality and expression dynamics. *PLoS computational biology*, 3(4):e59, April 2007. URL: https://europepmc.org/articles/PMC1853125, doi:10.1371/journal.pcbi.0030059.

[171] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. Imp: Indirect memory prefetcher. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 178–190, 2015. doi:10.1145/2830772.2830807.

[172] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 27–38, 2017. doi:10.1109/CGO.2017.7863726.

[173] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. *SIGPLAN Not.*, 50(8):183–193, January 2015. doi:10.1145/2858788.2688507.

[174] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302, USA, 2017. IEEE. doi:10.1109/BigData.2017.8257937.

[175] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276491.

[176] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, -, 2002. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.8280.

[177] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 301–316, Berkeley, CA, USA, 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu.

[178] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 375–386, USA, 2015. USENIX Association. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/zhu.

[179] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. GraphQ: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 712–725, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3352460.3358256.