Locality Analysis of Graph Reordering Algorithms

Mohsen Koohi Esfahani
0000-0002-7465-8003

Peter Kilpatrick
0000-0003-0818-8979

Hans Vandierendonck
0000-0001-5868-9259

{mkoohiesfahani01, p.kilpatrick, h.vandierendonck}@qub.ac.uk

Queen's University Belfast, Northern Ireland, UK
https://blogs.qub.ac.uk/GraphProcessing

***Abstract*—A major challenge in processing real-world graphs stems from poor locality of memory accesses and vertex reordering algorithms (RAs) have been proposed to improve locality by changing the order of memory accesses.**

While state-of-the-art RAs like SlashBurn, GOrder, and Rabbit-Order effectively speed up graph algorithms, their capabilities and disadvantages are not fully understood, mainly, for three reasons: (1) the large size of datasets, (2) the lack of suitable measurement tools, and (3) disparate characteristics of graphs. The paucity of analysis has also inhibited the design of more efficient RAs.

This paper unlocks this black box by introducing a number of tools, including: (1) a cache simulation technique for processing large graphs, (2) the Neighbour to Neighbour Average ID Distance (N2N AID) as a spatial locality metric, (3) the degree distributions of simulated cache miss rate and AID to investigate how locality of different vertices is affected by RAs, and (4) "effective cache size" to measure how much of cache capacity is used to support random accesses.

We introduce (1) asymmetricity degree distribution, (2) degree range decomposition, and (3) push and pull locality to present a structural analysis of different types of real-world graphs by explaining their contrasting behaviours in confronting RAs.

Finally, we propose a number of improvements for RAs using the analysis provided in this paper.

***Index Terms*—Graph processing, Graph reordering, Memory locality, High performance computing, Graph algorithms**

I. Introduction

Among data-intensive problems, graph processing is particularly challenging due to high memory bandwidth requirements. Many real-world graphs, such as those derived from social networks and the world-wide web, show a heavy-tailed or **power-law degree distribution**, i.e., a very small fraction of the vertices are connected to a disproportionately large fraction of the edges. The large size of these graphs results in seemingly-random memory accesses in graph traversals that cannot be completely satisfied by the processor caches and necessitates improving locality of accesses.

Locality is defined as *"the tendency for programs to cluster references to subsets of address space for extended periods"* [1], and graph relabeling (also called "reordering" or "renumbering") algorithms try to increase the cache hit rate by changing the order in which vertices are processed, and thus the order in which random memory accesses are made.

A **relabeling algorithm (RA)** improves locality of a graph traversal by assigning new IDs to vertices in a way that increases the clustering of memory accesses into a range that can be mostly satisfied by cache contents. However, identifying the optimal order that delivers the best locality is a NP-complete problem [2] and heuristics are employed in RAs [2]–[19] based on assumptions about graph structure or execution environment.

Some studies investigate the impact of RAs on graph analytics [20]–[23] 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** [24]–[26] summarize the characteristics of graphs independently of execution properties like processing order and vertex ID assignment. As such, they are great for analyzing the graph, but do not reflect on execution efficiency. **Reuse distance** curves [27]–[30] 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 analyzing 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**.

This paper studies three state-of-the-art RAs: SlashBurn [10], GOrder [2], and Rabbit-Order [11]. 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 misses and DTLB misses) and degree distribution of introduced metrics to compare contrasting effects of RAs on different graphs. To explain these effects, we present a structural analysis of graph datasets.

The **contributions** of this paper are thus:

978-1-6654-4173-5/21/©2021 IEEE
https://doi.org/10.1109/IISWC53511.2021.00020

- Introducing locality types in a parallel graph traversal,
- Introducing the Neighbour to Neighbour Average ID Distance (N2N AID) as a spatial locality metric,
- Introducing the degree distributions of simulated cache miss rates and AID to study impacts of RAs on vertex classes,
- Introducing degree range decomposition and the degree distribution of asymmetricity to provide structural analysis of different graph types, and
- A characterization of how locality manifests itself differently in a push traversal vs a pull traversal.

Section II describes the background materials and terminology. Section III explains the methodology and the datasets. Section IV presents the RAs studied in this paper and introduces the locality types. We introduce the graph-specific cache simulation technique and the AID metric in Section V. Section VI analyzes the RAs by extending and improving our previous study [31]. Section VII demonstrates a structural analysis of datasets. Finally, we present improvements to RAs and future work in Section VIII.

II. Background

A. Graph Representation

Graph $G = (V, E)$ has a set of vertices V, and a set of directed edges E. The adjacency matrix is a binary matrix representing 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.

We use the Compressed Sparse Columns (CSC), and Compressed Sparse Rows (CSR) formats [32] for representing **topology data** of graphs. 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 and specifies the index of the first edge of that vertex in the edges array. The edges array specifies the ID of source and destination of edges in CSC and CSR, respectively. Each element of the offsets array has a size of 8 Bytes and each element of the edges array has a size of 4 Bytes.

In addition to topology data, the **data of vertices** is stored in an array of $|V|$ elements and is indexed by a vertex ID.

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

B. Graph Traversal Model

There are many variations to graph traversal patterns depending on the graph analytics algorithm evaluated. In order to present a single, widely applicable analysis, we focus on a **Sparse Matrix-Vector (SpMV) multiplication** graph traversal model (Algorithm 1) that traverses all edges of the graph. SpMV underpins several graph analytics like Hyperlink Induced Topic Search [33], Belief Propagation [34], Graph Neural Networks [35], Recurrent Neural Networks [36], PageRank [37], and Community Detection [38], and is the target structure of RAs.

Algorithm 1: SpMV graph traversal

```
Input: G(V, E), D^i
Output: D^{i+1}
1 for v ∈ V do
2 |   sum = 0;
3 |   for u ∈ v.neighbours do
4 |   |   sum += D^i[u];
5 |   end
6 |   D^{i+1}[v] = sum;
7 end
```

SpMV traverses all edges of the graph which allows it to reveal the maximum improvement provided by RAs. Other graph analytics such as Breadth-First Search (BFS), Connected Components (CC), and Single Source Shortest Path (SSSP) selectively traverse edges as their execution is organized around a frontier (worklist). For instance, the frontier in BFS and SSSP is dependent on the start vertex. So, the memory access pattern (i.e., locality) are unpredictable as they depend on the specifics of the worklist content. Nonetheless, these algorithms have **dense** phases where all or the majority of the edges are processed (similar to SpMV). As the dense phases dominate the execution time, SpMV is also a suitable representative of these graph analytics.

C. SpMV

Algorithm 1 demonstrates the **SpMV in pull direction** that we use to investigate RAs in this paper. 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 vertex data ($\mathbf{D}^{i+1}$) is calculated using the vertex data ($\mathbf{D}^{i}$) of neighbours.

D. Random and Sequential Memory Accesses

As a result of the CSC graph representation, memory accesses for reading neighbours of a vertex (Line 3 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.

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. So, **accessing data of a vertex such as u is repeated** in processing each of its out-neighbours. SpMV makes $|E|$ accesses to $|V|$ elements of the data array ($|E| \gg |V|$). On average, each vertex data is read $\frac{|E|}{|V|}$ times; however, accesses to the data of a vertex are dispersed among $|E|$ accesses. In this way, these accesses are called **random**.

TABLE I: Datasets

Dataset	Name	Source	|V| (M)	|E| (B)	Type
WebB	WebBase-2001	LWA	115	1.0	WG
TwtrMpi	Twitter MPI	NR	41	1.5	SN
Frndstr	Friendster	NR	65	1.8	SN
SK	SK-Domain	LWA	50	2.0	WG
WbCc	Web-CC12	NR	89	2.0	WG
UKDls	UK-Delis	LWA	110	4.0	WG
UU	UK-Union	LWA	133	5.5	WG
UKDmn	UK-Domain	KN	105	6.6	WG
ClWb9	ClueWeb09	NR	1,700	7.9	WG

E. Graph Reordering

CPU caches try to keep the recently accessed data in cache to accelerate execution by preventing expensive memory accesses; however, the skewed degree distribution of large real-world graphs results in a huge number of random accesses that cannot be fully satisfied by cache (with limited capacity). So, 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) the number of edges each vertex has. RAs keep the second factor unchangeable and concentrate on the first factor: **rearrange the relative order between vertices to consequently change the order of edges, i.e., the order of random accesses**.

A RA permutes vertex IDs and receives a graph as its input and creates a relabeling array of size $|V|$ which is indexed by the old ID of a vertex to specify the new ID. Then, the CSC/CSR representations are rebuilt using the relabeling array.

F. Traversal Directions

Parallel traversal of edges of a graph is usually performed in push or pull direction.

In **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 **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**. Push direction has an additional cost for protecting the data of vertices from concurrent updates made by parallel processors. We focus this study on the pull direction which is faster.

Based on the adjacency matrix definition (Section II-A), 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. So, a **pull traversal uses the CSC** format and a **push traversal uses the CSR** format.

In this paper we study the pull traversal of SpMV, except in Section VII-B where we compare push and pull traversals.

III. Methodology

A. Datasets

Table I shows the datasets and sources: "*Konect*" (KN) [39]–[41], "*NetworkRepository*" (NR) [42]–[46], and "*Laboratory for Web Algorithmics*" (LWA) [16], [41], [43], [47], [48].

TABLE II: [Real execution] Preprocessing overheads

Dataset	Pre-processing Time (s)			Memory Footprint (GB)		
	SB	GO	RO	SB	GO	RO
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		

Numbers of edges are in billions and numbers of vertices are in millions, counted after removing zero degree vertices because of their destructive effect [49]. Graph types are WebGraph (WG) or Social Network (SN).

B. Environment

We use a 2-socket machine with 768 GB main memory. Each socket has an Intel® Xeon® Gold 6130 with 16 cores, 32KB L1 cache, 1MB L2 cache, and 22MB L3 shared cache. The machine uses CentOS 7 and does not use hyper-threading.

To have a correct evaluation of RAs, it is necessary to reduce the execution overheads of the processing framework because those overheads could swallow up the improvements provided by RAs. A study [50] shows that native hand-optimized implementations of graph analytics are faster than graph processing frameworks by a large performance gap. We used an optimized implementation of SpMV using `pthread`, `libnuma`, and `papi` [51] libraries. It uses the interleaved NUMA memory policy and applies work-stealing [52] for parallel processing of graph partitions created by edge-balanced partitioning [49]. The master-worker model is used for managing parallel threads and `futex` syscall for thread synchronization. The compiler is `gcc-9.2` with `-O3` flag. We use 8 Bytes vertex data.

Compared to other graph processing frameworks such as GraphGrind [49] (commit `5099761`), GraphIt [53] (commit `c4781d8`, OpenMP), and Galois [54](V5, commit `6ce5f0d`), for SpMV PageRank our implementation is faster for all datasets and, on average, it is faster by 1.2× - 2.1×.

IV. Relabeling Algorithms & Locality types

This section briefly explains the RAs studied in this paper and then introduces locality types. Table II shows the preprocessing time (in Seconds) and memory footprint (in GigaBytes) of the RAs.

A. SlashBurn

SlashBurn (SB) [10] considers the hubs as the main connector between vertices and exploits this feature to detect communities of vertices by removing hubs and finding the connected components (communities). This process continues in the next iteration for the giant connected component (**GCC**) - the community with the largest number of edges. SB assigns consecutive IDs to hubs of the main graph and the giant

communities starting from 0 (based on their degree) and vertices in a community also receive contiguous IDs.

We selected SB as it targets specifically real-world graphs; moreover, it is a representative of degree-ordering RAs. The original implementation of SlashBurn is in MATLAB® and we implemented a parallel version of SB in the C language that uses "basic hub-ordering" and $k = 0.02|V|$.

B. Rabbit-Order

Rabbit-Order (RO) [11] 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: $\Delta Q_{u,v} = 2(\frac{w_{uv}}{2|V|} - \frac{deg_u deg_v}{(2|V|)^2})$, where w_{uv} is the weight of edge between u and v and deg_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 weight 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.

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

C. GOrder

GOrder (GO) [2] 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 edges between u and v. 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 with its default window size (5) that is a single-threaded implementation for graphs with $|E| < 2^{31}$.

D. Locality Types

Considering random memory accesses in Line 4 of Algorithm 1, the following patterns of reuse of vertex data are identified. Each memory access is performed by cache queries and a TLB lookup in VIPT (Virtually Indexed, Physically Tagged) caches. We skip explanation of TLB index reuse for each locality type as it is similar to cache line reuse.

- **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 consecutive 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 (parallelization) and scheduling 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 clusters, and types II and III by assigning consecutive IDs to hubs.

V. Metrics and Tools

In order to measure spatial locality (type I), we introduce N2N AID degree distribution in Section V-A. Section V-B introduces cache miss rate degree distribution that measures temporal and spatio-temporal locality (types II and III).

A. 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 RAs succeed 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:

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

When a 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 VI-C (Figure 3). AID degree distribution is computed in $\mathcal{O}(|E|)$ time and $\mathcal{O}(max - degree)$ space complexities.

It is useful to compare N2N AID to "average gap profile" [23] 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 a 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 a 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). Consequently, changes in AID are generally sufficient to affect cache or TLB miss rates.

B. Cache Miss Rate Degree Distribution

To collect detailed information about RAs, we collect cache miss rate based on degree of vertices that 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 [55]) is time-consuming for large graphs.

Since graph analytics are memory intensive (in SpMV, there is just an add computation in Algorithm 1, Line 4), 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 [56] and equipped it with an accurate implementation of the dueling BRRIP and SRRIP [57] 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. 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 1 shows the degree distribution of cache miss rate for RAs. We will interpret these results in Section VI.

For datasets used in this study the average simulation time 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 1 are valid.

VI. Locality Analysis of RAs

A. Locality Analysis of 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"* [10]. The main idea is to iteratively remove hubs of power-law graphs; however, the practicality of this method depends on whether power-law graphs are destroyed recursively.

To assess this theory we depict the degree distribution of GCC for different iterations of SB in Figure 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 perceived as different communities. As a result, neighbours are assigned widely distinct IDs that reduces locality types I and III.

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 VI-F).

B. Locality Analysis of GOrder

GO tries to optimize locality by maximizing 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 IV-C). 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 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 occur in accessing data of HDV**. Table III 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**.

As we explained in Section VI-A, **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

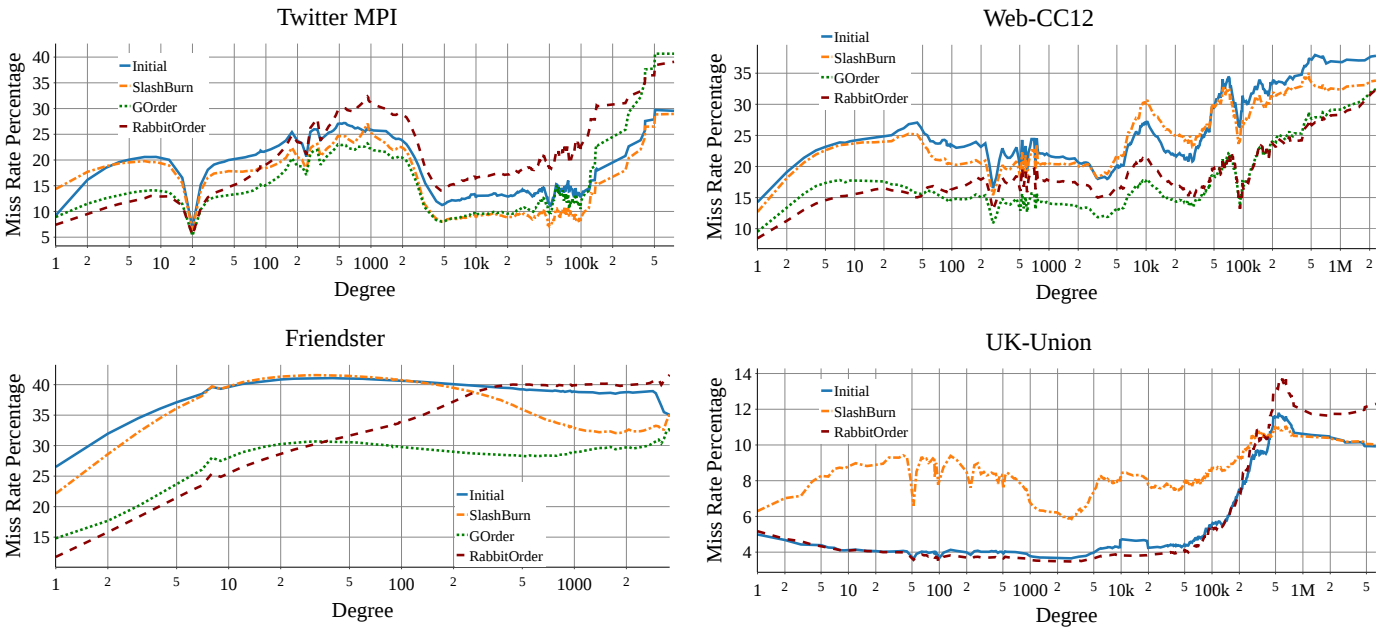

Fig. 1: [Simulation] Cache miss rate degree distribution

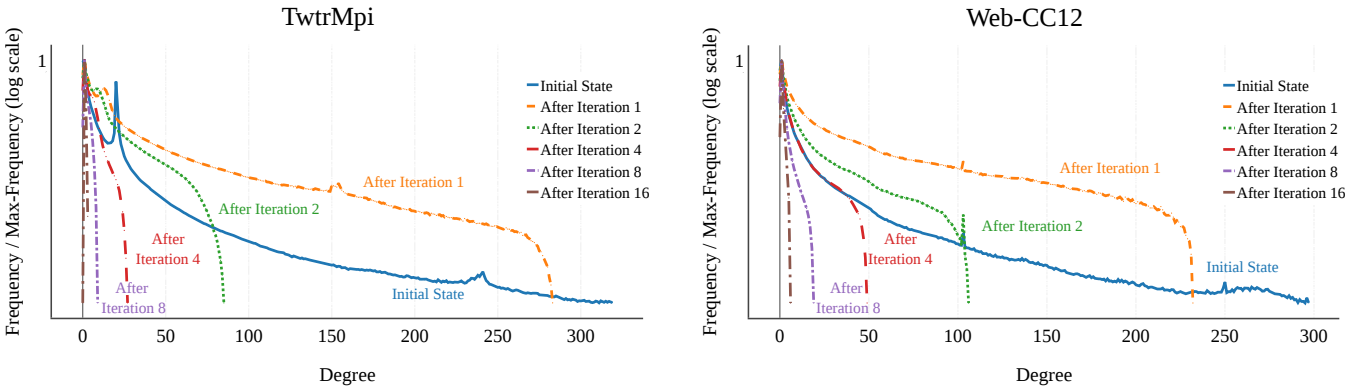

Fig. 2: [Real execution] Degree distribution of initial graph and GCC after SB iterations

TABLE III: [Simulation] Total number of misses (in millions) for accessing data of vertices with $degree > \text{“}Min.\ Degree''$

Dataset	Min. Degree	Initial	SB	GO	RO
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

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.

C. Locality Analysis of Rabbit-Order

RO builds communities bottom-up and starts from low degree vertices and merges neighbouring vertices while trying to maximize the gain function (SectionIV-B). 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.

We use degree distribution of AID (Section V-A) to assess changes made by RO in spatial locality. As Figures 1 and 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**.

D. Observation on Hubs

Figure 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 exist in cache. For other ones, memory accesses are required. While RAs change the order of edges of hubs, they cannot change the topology of graph. So, **locality of hubs is not improved by RAs as much as other vertices**.

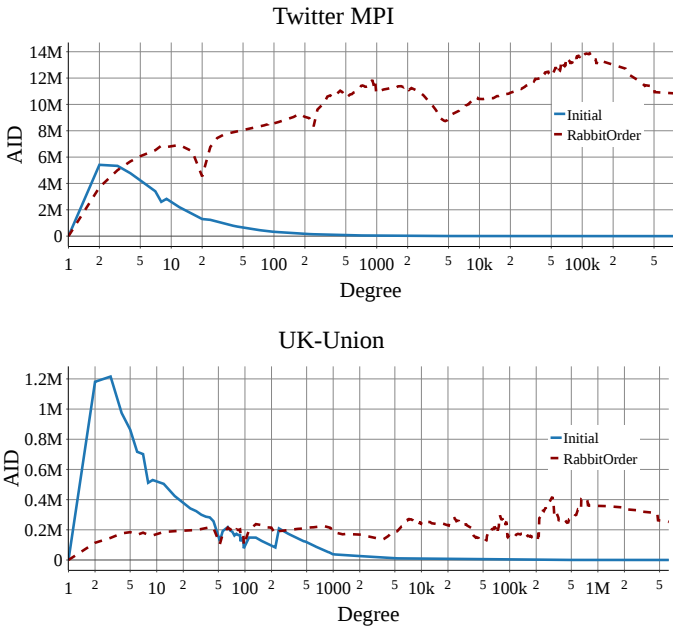

Fig. 3: [Calculation] AID degree distribution

Locality of hubs is important as they dedicate a large fraction of edges, i.e., a large fraction of traversal time to themselves and this observation demonstrates that **hubs of real-world graphs suffer from a structural problem in relation to locality that cannot be solved by RAs**.

Moreover, the data fetched from memory for processing an in-hub may be flushed before the end of processing that in-hub, as the data of subsequent neighbours of that in-hub (that again are missed in cache), should be read from memory and be placed in cache. In this way, **reuse of cache contents is reduced as a side effect of processing in-hubs**.

E. Real Execution Performance Metrics

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

Last level (L3) cache misses shows the number of memory accesses that are not satisfied by caches and sent to main memory. The number of L3 misses is the main locality metric. Table IV shows that **SB usually destroys locality** and increases the execution time. GO reduces L3 misses and execution time of **social networks**. RO improves locality of **web graphs** and reduces their performance.

DTLB misses specify 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 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 IV 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 evenly change the locality of consecutive vertices (as partitions that are assigned to or stolen by threads), as Table IV shows, **improving locality of a graph dataset by a RA may increase the idle time**.

*F. How Much of Cache Capacity Is **Effectively** Used?*

We introduce the term **Effective Cache Size (ECS)** as "the percentage of cache capacity dedicated to caching randomly accessed data". In SpMV (Algorithm 1), this measures the proportion of cache used to cache $\mathbf{D}^i$. It is important 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 is beneficial to 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 V shows the results: **RAs do not utilize all capacity of the cache to satisfy random memory accesses**.

Moreover, SB usually has the greatest ECS while it makes the most cache misses (Figure 1 and Table IV). 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 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, only a fraction of cache capacity is dedicated to those frequently accessed vertices and ECS is reduced. Comparison of L3 misses in Table IV and ECS in Table V also shows that **the RA with the best locality for a dataset usually has the lowest ECS**.

This observation has an important repercussion for hardware design: the full cache capacity remains unused in the current state of the art. So, improving locality will mean **caches are even more over-sized**. Moreover, **we need algorithms capable of deploying all capacity of the cache**.

Increasing ECS in SB results in filling cache with a great number of vertex data and **locality types IV and V are improved in processing numerous neighbours of hubs**. So, as Figure 1 shows, **the miss rate of hubs is reduced by SlashBurn**.

TABLE IV: [Real execution] SpMV execution results (Bl: Baseline without relabeling)

Dataset	Time (ms)				Idle (%)				L3 Misses (M)				DTLB Misses (K)			
	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 V: [Simulation] Average effective cache size (%)

Dataset	Initial	SB	GO	RO
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

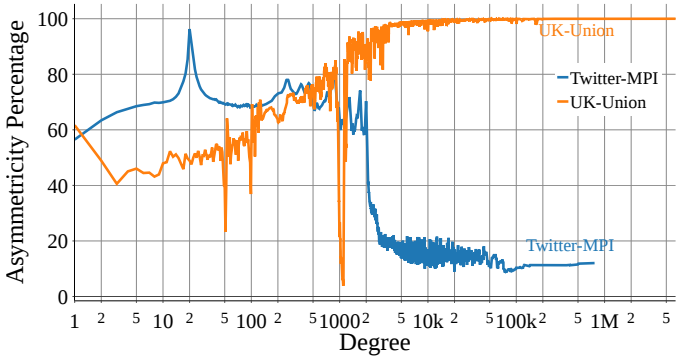

Fig. 4: [Calculation] Asymmetricity degree distribution

VII. Locality Analysis of Datasets

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

A. Web Graphs vs. Social Networks

Table IV shows that the RA that performs well for social networks is GO and for web graphs, it is RO. Section VI-B explains that GO improves locality of HDV and Section VI-C 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 4 compares the degree distribution of asymmetricity of `TwtrMpi` (as a social network) to `UK-Union` (as 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 5 represents the **degree range decomposition** as the correlation between the degrees of neighbouring vertices: all edges to vertices in a degree class are binned based on the degree class of their source vertex. E.g., 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 III also shows that RO has the most reloads, but GO manages hubs based on their temporal reuse (Section IV-C). 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 1 and 3).

B. Push Locality vs. Pull Locality

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

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

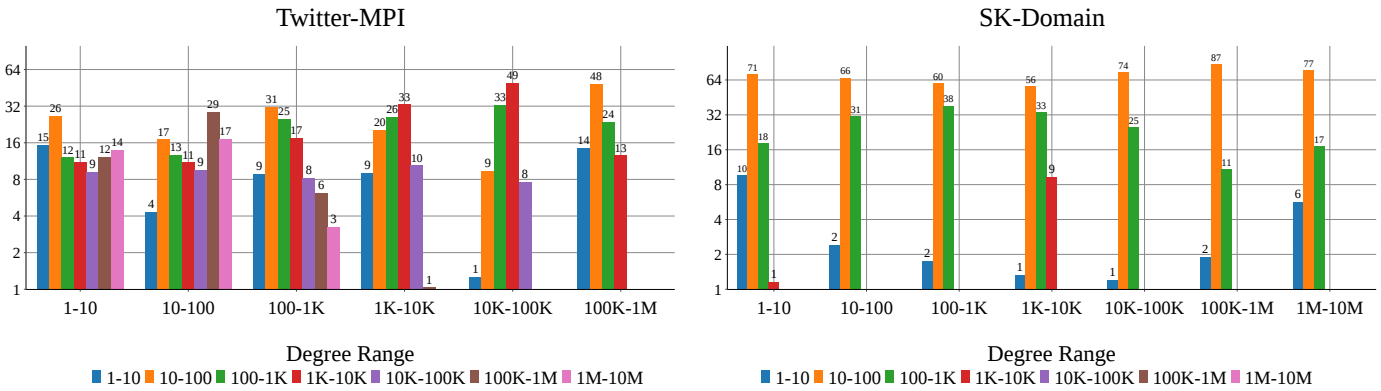

Fig. 5: [Calculation] Degree range decomposition of neighbours of vertices (in percent)

TABLE VI: [Real execution] CSC vs. CSR read traversals

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

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 the first step to understand the impacts of different real-world graphs on locality of push and pull traversals. Table VI 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**.

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 VII-A: (1) real-world graphs may have both in-hubs and out-hubs or only one of them, and (2) in-hubs are not always out-hubs. 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 that are processed by keeping H hubs with maximum degrees in the cache. This shows **what fraction of total edges (as an indicator of total processing) is covered (completed) by these H hubs**. Figure 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 6 shows that pull traversal of `Twitter MPI` can process 44% of edges by keeping $100K$ out-hubs in the cache, but push traversal can process about 23%. For `SK-Domain` it is vice versa, and pull traversal can process only 4% of the edges, while push traversal can process 64% of 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**.

VIII. Optimizing Locality and RAs

A. Optimizing Locality and Memory Accesses

Section VI-D showed that RAs are incapable of improving locality of hub vertices. To counter this, iHTL [58] presents a SpMV traversal to optimize locality of in-hubs in real-world graphs. iHTL creates dense flipped blocks (sub-graphs) that contain edges to in-hubs and processes them in push direction, while processing the sparse block in the pull direction.

In contrast to RAs that are not able to effectively utilize cache (Section VI-F), iHTL specifies the number of in-hubs by considering the cache size. In this way, cache capacity utilization is optimized in processing flipped blocks.

Section II-E showed two general methods for improving locality of random accesses: (1) changing the order of vertices, and (2) rearrangement of edges. The former is used by RAs and the latter is deployed by iHTL by creating a number of sub-graphs to exploit locality in processing in-hub vertices.

Thrifty Label Propagation [59] presents a structure-aware Connected Components by reducing memory accesses in processing hubs of power-law graphs.

B. Improving RAs

1) SlashBurn: In Section VI-A we explained that the GCC of SB does not have a power-law degree distribution after some initial iterations and contains a network of LDV. So, the next iterations destroy the neighbourhood of LDV. To counter this, we propose a variation on SB (called SlashBurn++), that continues the iterations while GCC-max-degree $\geq \sqrt{|V|}$. **SlashBurn++ reduces preprocessing time, traversal time, and L3 misses** (Table VII).

2) Rabbit-Order: Degree distribution of cache miss rate (Figures 1) can be used to identify an **efficacy degree range (EDR)** that for vertices in this range, the RA improves locality. We can **skip relabeling vertices that are not in EDR to reduce the preprocessing time and memory space**: during

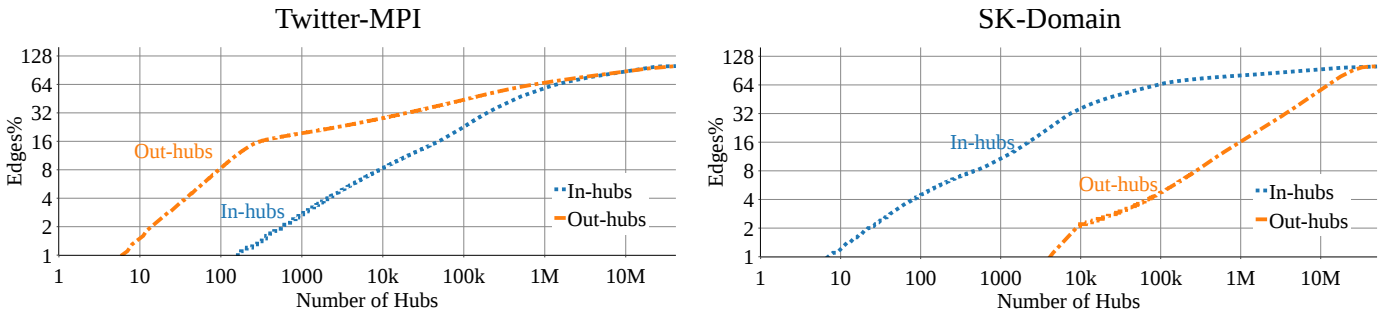

Fig. 6: [Calculation] Comparison of percentage of edges covered by in-hubs in CSR traversal vs. out-hubs in CSC traversal

TABLE VII: [Real execution] Effect of stopping SlashBurn after initial iterations

Dataset	**Preprocessing** (s)		**Traversal** (ms)		**L3 Misses** (M)	
	SB	SB++	SB	SB++	SB	SB++
TwtrMpi	46	21	339	328	14.2	13.6
Frndstr	75	43	761	700	39.2	36.0
WbCc	81	39	414	334	19.3	14.6

relabeling we pass only edges of those vertices to the RA that their degree is within the EDR. For other vertices, we let the labels be determined in the same manner as zero degree vertices. By applying this technique to RO and for two datasets, we experienced reduction in preprocessing time without affecting the traversal time. For `Frndstr` the preprocessing time reduced from 139 to 103 seconds, and for `TwtrMpi` from 66 to 12 seconds.

C. Further Suggestions as Future Work

Section VI-B showed that GO improves locality of HDV, while RO improves locality of LDV (Section VI-C). Therefore **a new RA can merge Rabbit-Order and GOrder techniques to improve locality of both of LDV and HDV**. Such an RA may start from LDV like RO to build initial clusters and then switch to a method like GO to relabel HDV.

We observed that GO cannot improve locality of LDV because of its fixed size of sliding window. It can be improved by **dynamically changing size of sliding window based on the contents of the window**. Moreover, **the vertex selection policy of GOrder can be improved**, for example by selecting the vertex with the highest percentage of neighbours that can be processed by traversing prior vertices in the sliding window.

In Section VI-A, we saw ECS is affected by RAs. However, RAs are cache-oblivious algorithms [60], [61] and do not take the cache size into account. **RAs can be improved by considering caching parameters of the execution machine(s)**:

- SB can specify the number of hubs and therefore the number of its iterations based on the cache size,
- GO can use cache size to identify its window size, and
- RO also can use cache size as an indicator of the maximum number of vertices in a community which prevents increasing size of communities indefinitely (Section VII-A).

IX. Related Work

A. Locality Optimizing Algorithms

Space filling curves improve locality without relabeling the graph. These techniques have first been investigated for dense linear algebra [62]. More recently they have been applied to graph processing [63], [64]. They are most easily applied in a coordinate list representation of the graph.

In [16], graph relabeling is used to provide better locality for neighbour vertices and therefore to provide better graph compression.

B. Evaluation of RAs

The impacts of RAs on different graph analytics have been studied in [20]–[23]; however, these studies do not reveal details of RAs and how they affect locality of graphs. This paper is the first one that investigates the functionality of RAs based on different vertex classes.

X. Conclusion

This paper introduces a number of techniques to efficiently analyze graph reordering algorithms (RA) and their effects on real-world graphs. We classified **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. Moreover, 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 their spatial locality is improved by **clustering low-degree vertices** in Rabbit-Order. **Effective cache size** introduced as a metric of cache capacity utilization and we see it is reduced as locality is improved by RAs.

We also studied differences in locality of push and pull traversals as consequences of the structure of datasets and 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.

Finally, we presented some immediate improvements to RAs based on our study and also expressed further suggestions that need more fundamental research.

Online Web Page

Further discussions relating to this paper are available online on https://blogs.qub.ac.uk/GraphProcessing/Locality-Analysis-of-Graph-Reordering-Algorithms/.

Acknowledgement

We are grateful to the shepherd, Prof. David Kaeli, and the other IISWC'21 reviewers of this paper for several suggestions to improve the presentation of this work.

We thank Jordan McComb for the SkyLakeX cache simulation system from his Master thesis.

This work is partially supported by the High Performance Computing center of Queen's University Belfast and the Kelvin supercomputer (EPSRC grant EP/T022175/1) and by DiPET (CHIST-ERA project CHIST-ERA-18-SDCDN-002, EPSRC grant EP/T022345/1).

First author is supported by a scholarship of the Queen's University Belfast and the Department for the Economy, Northern Ireland.

References

[1] P. J. Denning and C. H. Martell, *Great Principles of Computing*. The MIT Press, 2015.

[2] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. NewYork, NY, USA: ACM, 2016, pp. 1813–1828.

[3] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. New York, NY, USA: Association for Computing Machinery, 1969, p. 157–172.

[4] I. P. King, "An automatic reordering scheme for simultaneous equations derived from network systems," *International Journal for Numerical Methods in Engineering*, vol. 2, no. 4, pp. 523–533, 1970.

[5] S. Sloan, "An algorithm for profile and wavefront reduction of sparse matrices," *International Journal for Numerical Methods in Engineering*, vol. 23, pp. 239–251, 1986.

[6] S. W. Sloan, "A fortran program for profile and wavefront reduction," *International Journal for Numerical Methods in Engineering*, vol. 28, no. 11, pp. 2651–2679, 1989.

[7] M. M. Strout and P. D. Hovland, "Metrics and models for reordering transformations," ser. MSP '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 23–34.

[8] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 606–618, 2006.

[9] J. C. Pichel, D. E. Singh, and J. Carretero, "Reordering algorithms for increasing locality on multicore processors," in *2008 10th IEEE International Conference on High Performance Computing and Communications*, 2008, pp. 123–130.

[10] Y. Lim, U. Kang, and C. Faloutsos, "Slashburn: Graph compression and mining beyond caveman communities," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, Dec 2014.

[11] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2016, pp. 22–31.

[12] X. Xu, N. Yuruk, Z. Feng, and T. 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*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 824–833.

[13] H. Shiokawa, T. Takahashi, and H. Kitagawa, "Scalescan: Scalable density-based graph clustering," in *Database and Expert Systems Applications*, S. Hartmann, H. Ma, A. Hameurlain, G. Pernul, and R. R. Wagner, Eds. Cham: Springer, 2018.

[14] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, oct 2008.

[15] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "VEBO: A vertex- and edge-balanced ordering heuristic to load balance parallel graph processing," *CoRR*, vol. abs/1806.06576, 2018.

[16] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011.

[17] T. Messi Nguélé, M. Tchuente, and J.-F. Méhaut, "Using Complex-Network properties For Efficient Graph Analysis," in *International Conference on Parallel Computing, ParCo 2017*, ser. Parallel Computing is Everywhere, vol. 32. Bologne, Italy: IOS Press Ebooks, Sep. 2017.

[18] T. Messi Nguélé and J.-F. Méhaut, "Applying Data Structure Succinctness to Graph Numbering For Efficient Graph Analysis," Aug. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03317712

[19] B. Huang, Z. Liu, and K. Wu, "Structure preserved graph reordering for fast graph processing without the pain," in *2020 IEEE International Conferences on HPCC/SmartCity/DSS*, 2020, pp. 44–51.

[20] G. Cong and T. Wen, "Locality behavior of parallel and sequential algorithms for irregular graph problems," 2007.

[21] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 1–13.

[22] V. Balaji and B. 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)*, Sep. 2018, pp. 203–214.

[23] R. Barik, M. Minutoli, M. Halappanavar, N. R. Tallent, and A. Kalyanaraman, "Vertex reordering for real-world graphs and applications: An empirical evaluation," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 240–251.

[24] G. Bounova and O. De Weck, "Overview of metrics and their correlation patterns for multiple-metric topology analysis on heterogeneous graph ensembles," *Physical Review E*, vol. 85, no. 1, p. 016117, 2012.

[25] P. Mahadevan, D. Krioukov, K. Fall, and A. Vahdat, "Systematic topology analysis and generation using degree correlations," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, p. 135–146, Aug. 2006.

[26] D. Magoni, "Nem: A software for network topology analysis and modeling," in *In "Proceedings of the 10th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems," IEEE Computer Society*, 2002.

[27] C. Ding and Y. Zhong, "Reuse distance analysis," 2001.

[28] K. Beyls and E. 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*, 2001, pp. 617–662.

[29] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?" in *Compiler Construction*, R. Gupta, Ed. Springer, 2010.

[30] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Predicting Whole-Program Locality through Reuse Distance Analysis*. New York, NY, USA: ACM, 2003.

[31] M. Koohi Esfahani, P. Kilpatrick, and H. Vandierendonck, "How do graph relabeling algorithms improve memory locality?" in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 2021, pp. 84–86.

[32] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations - version 2," 1994.

[33] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, pp. 604–632, Sep. 1999.

[34] U. Kang, D. H. Chau, and C. Faloutsos, "Mining large graphs: Algorithms, inference, and discoveries," in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 243–254.

[35] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li,

A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," *CoRR*, vol. abs/1909.01315, 2019.
[36] S. Narang, G. F. Diamos, S. Sengupta, and E. Elsen, "Exploring sparsity in recurrent neural networks," *CoRR*, vol. abs/1704.05119, 2017.
[37] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 30, no. 1-7, Apr. 1998.
[38] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," Tech. Rep., 2002.
[39] J. Kunegis, "KONECT – The Koblenz Network Collection," in *Proc. Int. Conf. on World Wide Web Companion*, 2013, pp. 1343–1350.
[40] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*. New York, NY, USA: ACM, 2007, p. 29–42.
[41] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.
[42] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.
[43] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004.
[44] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in twitter: The million follower fallacy," in *ICWSM*, Washington DC, USA, May 2010.
[45] F. social network, "Friendster: The online gaming social network," archive.org/details/friendster-dataset-201107.
[46] C. L. Clarke, N. Craswell, and I. Soboroff, "Overview of the trec 2009 web track," DTIC Document, Tech. Rep., 2009.
[47] P. Boldi, A. Marino, M. Santini, and S. Vigna, "BUbiNG: Massive crawling for the masses," in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2014, pp. 227–228.
[48] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, p. 591–600.
[49] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "GraphGrind: Addressing load imbalance of graph partitioning," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: ACM, 2017, pp. 16:1–16:10.
[50] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. 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*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014.
[51] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.
[52] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, Sep. 1999.
[53] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018.
[54] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *Proc. VLDB Endow.*, vol. 13, no. 8, p. 1304–1318, Apr. 2020.
[55] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011.
[56] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, p. 13–25, Jun. 1997.
[57] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 60–71, Jun. 2010.
[58] M. Koohi Esfahani, P. Kilpatrick, and H. Vandierendonck, "Exploiting in-Hub Temporal Locality in SpMV-based Graph Processing," in *Proceedings of The 50th International Conference on Parallel Processing*, ser. ICPP '21. Lemont, IL, USA: ACM, 2021.
[59] ——, "Thrifty label propagation: Fast connected components for skewed-degree graphs," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 2021.
[60] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms (extended abstract)," in *In Proc. 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1999, pp. 285–397.
[61] G. S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh, "Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths," in *Algorithm Theory - SWAT 2004*, T. Hagerup and J. Katajainen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.
[62] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS '99. New York, NY, USA: ACM, 1999, pp. 444–453.
[63] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. of the ACM Symp. on Operating Systems Principles*, 2013, pp. 439–455.
[64] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Accelerating graph analytics by utilising the memory locality of graph partitioning," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017.