Exploiting in-Hub Temporal Locality in SpMV-based Graph Processing

Mohsen Koohi Esfahani
mkoohiesfahani01@qub.ac.uk
Queen's University Belfast
United Kingdom

Peter Kilpatrick
p.kilpatrick@qub.ac.uk
Queen's University Belfast
United Kingdom

Hans Vandierendonck
h.vandierendonck@qub.ac.uk
Queen's University Belfast
United Kingdom

Abstract

The skewed degree distribution of real-world graphs is the main source of poor locality in traversing all edges of the graph, known as Sparse Matrix-Vector (SpMV) Multiplication. Conventional graph traversal methods, such as push and pull, traverse all vertices in the same manner, and we show applying a uniform traversal direction for all edges leads to sub-optimal memory locality, hence poor efficiency. This paper argues that different vertices in power-law graphs have different locality characteristics and the traversal method should be adapted to these characteristics.

To solve this problem, we propose to inspect the number of destination and source vertices in selecting a cache-compatible traversal direction for each type of vertex. We introduce *in-Hub Temporal Locality (iHTL)*, a structure-aware SpMV that combines push and pull in one graph traversal, but for different vertex types. iHTL exploits temporal locality by traversing incoming edges to in-hubs in push direction, while processing other edges in pull direction.

The evaluation shows iHTL is 1.5× - 2.4× faster than pull and 4.8× - 9.5× faster than push in state-of-the-art graph processing frameworks such as GraphGrind, GraphIt and Galois. More importantly, iHTL is 1.3× - 1.5× faster than pull traversal of state-of-the-art locality optimizing reordering algorithms such as SlashBurn, GOrder, and Rabbit-Order.

CCS Concepts: • **Computing methodologies → Shared memory algorithms**; **Massively parallel algorithms**; *Vector / streaming algorithms.*

Keywords: Graph traversal, Memory locality, High performance computing, Graph algorithms, Sparse Matrix-Vector Multiplication

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472462

ACM Reference Format:
Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. Exploiting in-Hub Temporal Locality in SpMV-based Graph Processing . In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472462

1 Introduction

Among data-intensive problems, graph processing is particularly challenging due to high memory bandwidth requirements and irregular memory access patterns that make it hard to optimize locality of memory accesses, especially in traversing all edges of the graph (or Sparse Matrix-Vector (SpMV) multiplication) where caches cannot contain the data of all vertices. Graph traversal is performed in two major patterns or directions: (1) In pull direction, a vertex pulls data of its in-neighbours and results in random read memory accesses and sequential write memory accesses. (2) In push direction, a vertex updates data of its out-neighbours and read accesses are sequential but write accesses are random.

The push traversal requires protecting data of out-neighbours from concurrent updates of parallel threads which is performed through (1) atomic instructions, (2) buffering [29], or (3) partitioning edges by destination [35]. Pull traversal is faster than push by traversing edges based on their unique destinations and therefore write memory accesses do not require protection. Pull traversal underpins several analytics like Hyperlink Induced Topic Search [20], Belief Propagation [19], Graph Neural Networks [40], Recurrent Neural Networks [27], PageRank [11], and Community Detection [47].

The structure of many real-world graphs poses challenges to efficient pull traversal. Graphs derived from social networks, the internet, and world-wide web show a skewed or heavy-tailed degree distribution, often following a power-law distribution: a very small fraction of vertices, which are known as hubs, are connected to a disproportionately large fraction of edges. The impact of hubs on temporal locality becomes problematic when **a massive amount of vertex data is pulled into the cache by pull processing of an in-hub (a vertex with large in-degree) which displaces much of the cache contents and intensely reduces the opportunity for future reuse**.

Locality optimizing graph relabeling algorithms [2, 9, 24, 36, 41, 42] use different techniques like graph clustering, community detection, and cache simulation to improve memory

locality by rearranging the order in which vertices are numbered. However, we found that graph reordering algorithms are successful in improving locality of non-hub vertices, but not so for the hubs. Crucially, hubs have so many neighbours that exploiting reuse of the neighbours of one hub to the neighbours of the next vertex is unlikely.

Our solution for this problem is based on the property of skewed graphs that a very small fraction of the vertices (the hubs) connect to a disproportionately high number of edges: **when traversing the in-edges of the in-hubs, it follows that the set of possible destinations (in-hubs) is very small, yet the set of possible sources is very large** (due to the large in-degree of the in-hubs). Our insight is that, for such a sub-graph, a cache-compatible traversal direction is the push direction and not the intended pull: **while the cache cannot satisfy random accesses to numerous source vertices of the edges to in-hubs in the pull traversal, the push traversal results in random memory accesses to a small set of destination vertices (in-hubs) that can be satisfied by cache**. Importantly, these destination-based random accesses can be captured in full in the on-chip caches if we control the number of hubs correctly. Consequently, we can traverse an important fraction of the edges with no random accesses to main memory.

This paper develops this insight and presents the **in-Hub Temporal Locality *(iHTL)*** that optimizes SpMV-based graph analytics by using push direction for processing incoming edges to in-hubs and pull for incoming edges to non-hubs.

The contributions of this paper are thus:

- We analyze the challenges imposed by hubs during graph traversal and demonstrate that locality optimizing graph relabeling algorithms fail to address locality issues of hubs.
- We introduce the iHTL algorithm that applies a bespoke mix of push and pull traversal in order to maximize cache reuse. We demonstrate how to efficiently prepare the iHTL graph in a light-weight preprocessing step by designing an algorithm to determine hubs based on graph structure.
- We evaluate iHTL on 10 real-world graphs with up to 7.9 billion edges demonstrating that iHTL significantly improves locality and outperforms the pull traversal in state-of-the-art graph processing systems such as GraphGrind [35], GraphIt [46] and Galois [16] by 1.5 × −2.4×. Moreover, the evaluation of iHTL in comparison to state-of-the-art relabeling algorithms such as SlashBurn [24], GOrder [41] and Rabbit-Order [2] shows iHTL is faster than pull traversal of the relabeled graphs by 1.3×−1.5× while reducing the preprocessing time by 780×.

Section 2 explains key background material and motivates the iHTL approach. Section 3 presents iHTL. The evaluation of iHTL is discussed in Section 4 and Section 5 studies related work. Section 6 presents future work.

Algorithm 1: SpMV in pull direction

Input: $G(V,\ E)$, $\mathcal{D}^{i-1}$, $\mathcal{D}^{i}$

1 **for** $v \in V$ **do**
2    $sum = 0;$
3   **for** $u \in N_v^-$ **do**
4      $sum \mathrel{+}= \mathcal{D}^{i-1}[u];$
5    $\mathcal{D}^{i}[v] = sum;$

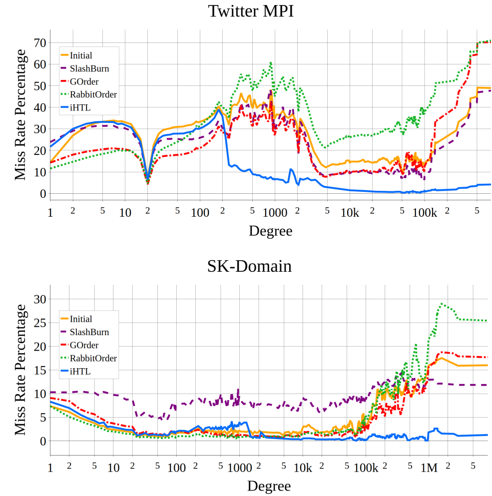


Figure 1. The last level cache miss rate of SpMV conditional on the degree of the traversed vertex

2 Background and Motivation

2.1 Terminology

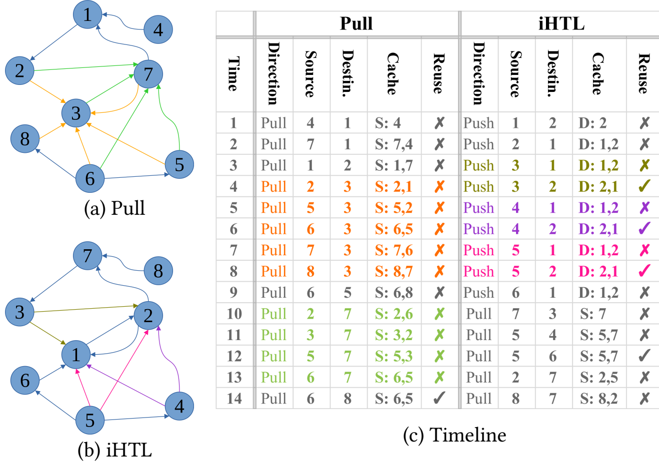
Graph $G = (V, E)$ has a set of vertices V, and a set of directed edges E. The adjacency matrix is a binary matrix representing the graph: the element at row i and column j is 1 if E contains an edge from vertex i to j, and 0 otherwise. Graphs are represented in Compressed Sparse Rows and Columns (CSR, and CSC) [30]. N_v^-, and N_v^+ are the set of in-neighbours and out-neighbours of vertex v, respectively.

The *hub*, *in-hub*, and *out-hub* vertices are vertices with the highest degree, in-degree, and out-degree, respectively. We do not present a formulaic definition of hubs. Instead, we reserve the term *hub* only for those vertices identified by iHTL as meriting an opposite traversal direction.

2.2 High Cache Miss Rate of in-Hubs

We first demonstrate that cache misses incurred during pull traversal are concentrated in the highest-degree vertices. We use SpMV multiplication (Algorithm 1) that iteratively calculates the new data of a vertex as summation of previous data of its in-neighbours. Using the notation of $\mathcal{D}_v^i$ for data of vertex v in iteration i, SpMV calculates: $\mathcal{D}_v^i = \sum_{u \in N_v^-} \mathcal{D}_u^{i-1}$.

Figure 1 depicts the miss rate conditional on the degree of a vertex for a social network graph (Twitter MPI) and a web graph (SK-Domain). **It shows that the initial graphs incur substantial miss rates for hubs that are the destination of the majority of the edges in power-law graphs** [21]



(a) Pull

(b) iHTL

	Pull					iHTL				
Time	Direction	Source	Destin.	Cache	Reuse	Direction	Source	Destin.	Cache	Reuse
1	Pull	4	1	S: 4	✗	Push	1	2	D: 2	✗
2	Pull	7	1	S: 7,4	✗	Push	2	1	D: 1,2	✗
3	Pull	1	2	S: 1,7	✗	Push	3	1	D: 1,2	✗
4	Pull	2	3	S: 2,1	✗	Push	3	2	D: 2,1	✓
5	Pull	5	3	S: 5,2	✗	Push	4	1	D: 1,2	✗
6	Pull	6	3	S: 6,5	✗	Push	4	2	D: 2,1	✓
7	Pull	7	3	S: 7,6	✗	Push	5	1	D: 1,2	✗
8	Pull	8	3	S: 8,7	✗	Push	5	2	D: 2,1	✓
9	Pull	6	5	S: 6,8	✗	Push	6	1	D: 1,2	✗
10	Pull	2	7	S: 2,6	✗	Pull	7	3	S: 7	✗
11	Pull	3	7	S: 3,2	✗	Pull	5	4	S: 5,7	✗
12	Pull	5	7	S: 5,3	✗	Pull	5	6	S: 5,7	✓
13	Pull	6	7	S: 6,5	✗	Pull	2	7	S: 2,5	✗
14	Pull	6	8	S: 6,5	✓	Pull	8	7	S: 8,2	✗

(c) Timeline

Figure 2. (a) Example graph with (in-hubs: #3, #7) - **(b)** iHTL graph (in-hubs: #1, #2) - **(c)** Timeline of pull vs iHTL (Effective cache size: 2 - cache contains data of source (S) vertices in pull direction, or destination (D) vertices in push direction.)

and consequently a significant part of execution time is consumed for processing hub vertices.

Figure 1 also illustrates the result of optimizing locality using relabeling algorithms: SlashBurn, GOrder, and Rabbit-Order. Relabeling algorithms change the order of vertices that affects hubs' locality by changing (1) the cache contents immediately before starting the processing of hubs, and (2) the spatial locality of hubs' neighbours. However, locality optimizing relabeling algorithms cannot change the structure of the graph and the number of hubs' neighbours.

Figure 1 shows that the locality optimizing relabeling algorithms can improve cache miss rates of non-hub vertices. However, **hubs experience a high cache miss rate even after reordering** [21]. In contrast, iHTL significantly reduces the cache miss rates corresponding to the hubs, which translates to substantial performance gain.

2.3 Inefficient Cache Utilization in Pull Traversal

To explain this high miss rate of hubs, Figure 2.(a) presents an example graph, and Figure 2.(c) shows its pull traversal execution timeline. Vertices 3 and 7 are in-hubs and the effective cache size is 2. For a LRU cache with one vertex data per cache line, the notation $[x, y]$ is used to show the data of vertices x and y are in the cache. Before processing vertex 3, vertices 1 and 2 have been processed and $\Rightarrow [1, 7]$.

For pull traversal of the first in-hub (vertex 3), the data of vertices 2, 5, 6, 7, and 8 should be read. Starting from vertex 2, its data is not in the cache and is fetched from the memory, and $\Rightarrow [2, 1]$. Then the data of vertex 5 is required that is not in the cache and is read from the memory and $\Rightarrow [5, 2]$. In the same way, the data of vertices 6, 7, and 8 also is read from the main memory and no reuse happens for processing 5 in-edges of vertex 3.

Algorithm 2: SpMV in push direction

Input: $G(V, E)$, $\mathcal{D}^{i-1}$, $\mathcal{D}^{i}$
1 **for** $v \in V$ **do**
2   **for** $u \in N_v^+$ **do**
3         $\mathcal{D}^i[u]$ += $\mathcal{D}^{i-1}[v]$;

Figure 2.(c) shows, the same behaviour happens for pull traversal of the second in-hub (vertex 7), and no reuse is experienced for processing 4 edges. This shows that the high degree of an in-hub limits the reuse of the cache contents in pull traversal of in-hubs.

2.4 iHTL Idea

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 random read accesses to data of source vertices in pull traversal**. In a push traversal (Algorithm 2), the new data of out-neighbours (Line 3) are randomly updated by data of source vertices and most of the capacity of **cache is dedicated to random write accesses to destination vertices in push traversal**.

On the other hand, real-world graphs with power-law structure have few in-hub vertices with each having several in-neighbours. In a pull traversal of an in-hub, cache is dedicated to source vertices and the number of these source vertices is greater than the capacity of cache resulting in high rate of cache misses (Figure 1).

iHTL states that for incoming edges to in-hubs, the number of destination vertices (in-hubs) is much less than the number of source vertices, therefore cache can be efficiently used only if it is dedicated to the destination vertices. In other words, push traversal is suitable for traversing incoming edges to in-hubs.

In order to facilitate this, iHTL relabels the graph and assigns the lowest vertex IDs to in-hubs: vertices 3 and 7 of the example graph in Figure 2.(a) become vertices 1 and 2 in Figure 2.(b). Moreover, iHTL processes the graph in two steps. In the first step incoming edges to in-hubs are processed in push direction. In the second step, other edges are processed in pull direction. Figure 2.(c) shows the iHTL execution timeline for Figure 2.(b). For push traversal of incoming edges to in-hubs, first, vertex 1 updates data of vertex 2 by its data and vertex 2 updates data of vertex 1 $\Rightarrow [1, 2]$. Then, vertex 3 reads its data from main memory and updates the data of in-hubs (vertices 1, 2), and one reuse out of two updates is achieved (cache content is $[2, 1]$). In the same way, two more reuses are achieved for push traversal of outgoing edges of vertices 4 and 5 to in-hubs.

The comparison of pull and iHTL execution timelines shows that **reuse is more frequent in iHTL because by push traversal of incoming edges to in-hubs, random accesses are targeted at a small number of in-hubs that are maintained in cache**. In iHTL every edge is traversed exactly once as it should be, even though iHTL mixes push

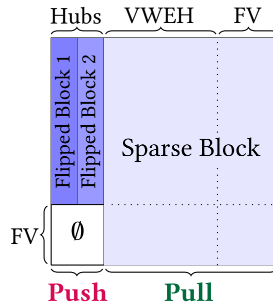


Figure 3. Adjacency matrix of an iHTL graph

and pull traversals. The timeline also shows that by increasing the number of common in-hubs, reuse is increased.

3 In-Hub Temporal Locality

iHTL performs push for incoming edges to in-hubs, and applies pull for incoming edges to non-hubs. To perform efficient push and pull, iHTL creates subgraphs for groups of edges. We explain iHTL graph in Section 3.1. Sections 3.2 and 3.3 discuss the process of creating an iHTL graph, and Section 3.4 shows how SpMV is performed in iHTL.

3.1 iHTL Graph

In order to facilitate push and pull traversals in iHTL, we distinguish blocks (subgraphs) within the graph adjacency matrix. Figure 3 shows the adjacency matrix structure of iHTL graph. Note that 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 three major parts:

- 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 cache. For a graph that has in-hubs more than cache capacity, 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 show in the evaluation section that flipped blocks in iHTL contain 40-70% of the edges for 8 out of 10 graphs.

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 3). 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.

3.2 Creating iHTL Graph

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

(1) Creating 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 a minimal change on the initial neighbourhood of the vertices.

Figure 4 shows the creation of the relabeling array for the example graph in Figure 2.(a). Firstly, in-hubs are selected as a number of vertices with the highest 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 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. Figure 5 shows the adjacency matrix of the example graph, and Figure 6 shows the adjacency matrix of its iHTL graph after relabeling.

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 blocks 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 using the iHTL relabeling array).

(3) Creating A 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.

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,

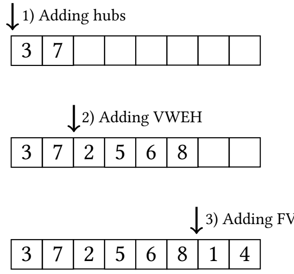


Figure 4. iHTL relabeling array (element v stores the original ID of v)

	#1	#2	#3	#4	#5	#6	#7	#8
#1		1						
#2			1				1	
#3							1	
#4	1							
#5			1				1	
#6			1		1		1	1
#7	1		1					
#8			1					

Figure 5. Adjacency matrix of graph in Figure 2.(a)

	#1	#2	#3	#4	#5	#6	#7	#8
#1		1						
#2	1						1	
#3	1	1						
#4	1	1						
#5	1	1		1		1		
#6	1							
#7			1					
#8							1	

Figure 6. iHTL Adjacency matrix of graph in Figure 2.(a) after relabeling

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 4.7). 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, **HTL needs to balance the benefit of creating more flipped blocks with the drawbacks**. The benefit is improved locality, however, there are two drawbacks for 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.

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 a buffering technique (where each thread operates on copies of the vertex data which are later merged [29]) 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

Algorithm 3: 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   **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     $\mathcal{D}^{i}_{hub} +=$ **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     $\mathcal{D}^{i}_{v} += \mathcal{D}^{i-1}_{u}$;

---

does not take more than 3% of iHTL execution time (each thread buffers $H * \#fb$ vertex data).

Algorithm 3 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).

4 Evaluation

4.1 Evaluation Method and Datasets

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.

HTL has been implemented in the C language using `pthread`, `libnuma`, and `papi` [37] libraries, and compiled by `gcc-9.2`

Dataset	Name	Source	|V| (M)	|E| (B)	In-degree Max. (K)	Out-degree Max. (K)
LvJrnl	LiveJournal	KN	7	0.22	15	15
Twtr10	Twitter 2010	NR	21	0.26	422	302
TwtrMpi	Twitter MPI	NR	41	1.5	770	2,997
Frndstr	Friendster	NR	65	1.8	4	4
SK	SK-Domain	LWA	50	2	8,564	13
WbCc	Web-CC12	NR	89	2	2,329	1,317
UKDls	UK-Delis	LWA	110	4	1,261	15
UU	UK-Union	LWA	133	5.5	6,367	22
UKDmn	UK-Domain	KN	105	6.6	975	975
ClWb9	ClueWeb09	NR	1.7K	7.9	6,445	2

Table 1. Datasets

	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×

Figure 7. Per iteration execution time (in milliseconds) of PageRank

with `-O3` flag. The implementation uses interleaved NUMA memory policy and applies work-stealing [6] for parallel processing of graph partitions created by vertex and edge partitioning [35, 44]. The master-worker model has been used for managing parallel threads.

Table 1 shows the datasets and their sources: "*Konect*" (KN) [7, 22, 26], "*NetworkRepository*" (NR) [10, 13, 15, 28, 32], and "*Laboratory for Web Algorithmics*" (LWA) [7–10, 23]. The first 4 datasets are social networks, and the other ones are web graphs. Numbers of edges are in billions and numbers of vertices are in millions, counted after removing zero degree vertices because of their destructive effect [35]. Graphs are represented in CSR and CSC with $|V|+1$ index values of 8 bytes per index value and $|E|$ neighbour IDs of 4 bytes each as $|V| < 2^{32}$.

GraphGrind (commit `5099761`), GraphIt (commit `c4781d8`, OpenMP), and Galois (V5, commit `6ce5f0d`) are graph processing frameworks we use to evaluate iHTL. GOrder [41] (commit `7ccdfe9`), Rabbit-Order [2] (commit `f67a79e`), and SlashBurn [24] are state-of-the-art locality optimizing relabeling algorithms we use for evaluation of iHTL. GOrder has a limit of $|E| < 2^{31}$ and Rabbit-Order could not complete relabeling of `ClWb9` because of an "out of memory" error.

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

4.2 iHTL vs Pull and Push Implementations

Figure 7 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 [45] locality optimizations (Section 5.4) which make it faster than Galois for some graphs. Figure 7 demonstrates the effectiveness of the iHTL locality optimizations as it is faster than different implementations of pull traversal by 1.5× - 2.4×.

Figure 7 also shows that **iHTL preserves the initial locality of graphs well, even for graphs like "SK-Domain" with high initial locality**.

Dataset	GraphGrind	GraphIt	Galois	iHTL
LvJrnl	16.8	8.5	24.1	32.6
Twtr10	6.3	11.8	7.9	15.7
TwtrMpi	7.1	12.2	11.6	18.3
Frndstr	5.0	6.8	6.6	9.3
SK	13.8	21.4	22.8	35.8
WbCc	3.6	5.8	5.3	9.2
UKDls	6.2	10.6	11.7	14.3
UU	5.0	8.8	9.7	11.9
UKDmn	6.8	12.5	13.5	15.8
ClWb9	2.3	5.0	3.8	7.1
Average	7.3	10.3	11.7	17.0

Table 2. iHTL preprocessing overhead based on PageRank iterations (The numbers show how many SpMV iterations are performed in the time iHTL requires for preprocessing).

Table 2 shows the overhead of iHTL preprocessing as a multiple of PageRank iterations in different frameworks. It shows that, on average, **iHTL requires 7.3 - 11.7 SpMV iterations in other frameworks as the preprocessing time**. The preprocessing overhead can be completely amortized between different executions if the iHTL graph is stored in its binary format (similar to the special file formats that each framework uses) on disk after preprocessing. In Section 6, we present avenues for future work that may reduce the preprocessing time of iHTL.

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 3. Memory accesses (load and store instructions), L3 and L2 cache misses (in millions)

4.3 Memory Accesses and Cache Misses

Table 3 compares the memory accesses (loads and stores of data) 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, (2) updates to local buffers when processing flipped blocks, (3) merging buffers and (4) resetting buffers. Types 1, 3, and 4 are sequential, i.e., assisted by prefetching. Type 2 includes random writes, however, these are captured by the level 2 cache.

Dataset	CSC (GiB)	iHTL (GiB)	iHTL Overhead (%)
LvJrnl	.9	1.	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.	3
UU	22.5	23.3	3
UKDmn	26.6	27.2	2
ClWb9	43.8	44.9	3

Table 4. Size of topology data (in Giga Bytes)

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.

4.4 iHTL Memory Overhead

Table 4 compares the memory size of CSC representation of the main graphs vs. 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 is too large to fit in on-chip caches. The size increase is therefore not a major problem. Section 6 explains methods for reducing topology data.

4.5 iHTL vs Relabeling Algorithms

To have a better scale of locality optimization of iHTL, Figure 8 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 Figure 1 shows 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 (Figure 1). In contrast, iHTL targets locality of hubs (Figure 1), which capture a significant portion of the edges (Table 5). Thus, iHTL outperforms the relabeling algorithms.

Figure 8 compares the preprocessing time of iHTL to relabeling algorithms. GOrder has a sequential implementation. SlashBurn and Rabbit-Order have a parallel code, however the complexity of their algorithms makes them much slower than iHTL. iHTL has a simple preprocessing algorithm (Section 3.2) and does not need to investigate the neighbourhood of each vertex in detail. This gives iHTL a very short preprocessing time.

4.6 iHTL Execution Breakdown

Table 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 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 5 shows that buffer aggregation in iHTL requires less than 2.5% of total processing time. Each flipped block implies buffer merging overhead, however, by inspecting graph structure iHTL incurs this overhead only when there is a corresponding gain in locality.

4.7 iHTL Buffer Size

Table 6 shows the impact of iHTL buffer size. Note that buffer size determines the number of hubs per flipped block. The aim is for random accesses to the buffers to be serviced fast. Aligning the buffers to L1 cache size is inefficient as its 32 KB size is too small to accommodate many hubs. The L2 cache is private to each core, which implies unfettered access.

	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×

Figure 8. Left: Execution time (in milliseconds) of pull traversal after relabeling vs iHTL - Right: The preprocessing time of relabeling algorithms vs iHTL (in seconds)

Dataset	Graph 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 5. iHTL graph statistics and iHTL PageRank execution breakdown (FB: flipped blocks, Topo: topology data)

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. Execution time (in milliseconds) for different buffer sizes

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 suboptimal 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, as Table 6 shows, L2 cache is the best choice for accommodating data of in-hub vertices.

5 Related Work

5.1 Low-Degree vs. High-Degree

The history of using different traversals for different vertices returns to the AYZ algorithm [1] for triangle counting where low-degree and high-degree vertices are differentiated to reduce computational complexity.

PowerLyra [14] reduces the communication cost in a distributed graph processing system by using vertex-cut partitioning for low-degree vertices and edge-cut for high-degree vertices. In this way, PowerLyra ensures that replicas of low-degree vertices are not increased and processing high-degree vertices will experience better load balance.

5.2 Push OR Pull

The effectiveness of push OR pull traversals are discussed in [3, 5, 25, 33, 39]. These studies apply the same traversal direction for all vertices in a single traversal of all edges. The push or pull traversal is selected in these works based on density of frontier or possible convergence optimization that can be applied on a special direction. Also, push and pull locality have different effects on the traversal performance [21].

On the other hand, iHTL applies different traversal directions for different vertex types in one traversal of all edges of the graph.

5.3 Locality Optimizing Graph Reordering

Community detection algorithms like Scan [42] provide better locality. Scan isolates hubs and outliers (vertices marginally appended to clusters) from clusters to prevent unrelated communities to be merged because of only one hub neighbour. ScaleScan [31] removes unnecessary computations in Scan and parallelizes the execution. Graph relabeling algorithms like SlashBurn [24], GOrder [41] and RabbitOrder[2] rearrange vertices in order to improve locality.

In contrast to vertex reordering algorithms, **iHTL concentrates on reordering edges as its primary goal** to

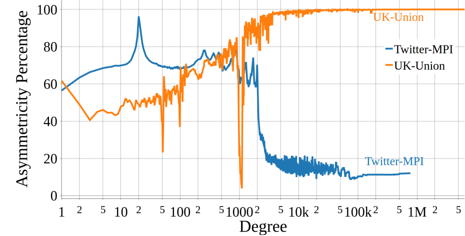


Figure 9. Asymmetricity degree distribution

provide temporal locality and as Section 4.5 shows iHTL provides better locality while reducing the preprocessing time.

5.4 Blocking Strategies

Starting from [18], blocking techniques have been widely used to achieve different goals. GraphGrind [34] and Graptor [38] apply vertical blocking in their push traversals in order to prevent race conditions made by concurrent updates.

Cagra [45] applies horizontal blocking of the adjacency matrix in 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. iHTL provides an efficient buffering limited to in-hubs.

Lav [43] reduces the overheads of Cagra by creating horizontal dense blocks only for those out-hubs that 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. In contrast, iHTL's flipped blocks are easily load-balanced and are processed concurrently (Section 3.4).

Moreover, since **real-world graphs are not truly power-law graphs** [12], 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 from their degree** (Section 3.3), and flipped blocks contain a wide range of 13% - 68% of the edges (Table 5).

Efficient horizontal blocking based on out-degrees is fundamentally impossible in some graphs. Figure 9 compares the asymmetricity of vertices grouped by degree for a social network (`Twitter MPI`) and a web graph (`UK-Union`). Asymmetricity of a vertex is defined 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 9 shows that **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 and can increase 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 has in-hubs and no out-hubs (Table 1), 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 4.6).

iHTL creates flipped blocks using the same type of hubs that experience low locality: in a pull traversal, in-hubs do not experience locality and iHTL creates vertical flipped blocks based on in-hubs that exist.

Moreover, iHTL maintains the relative order of vertices within the VWEH and FV categories, while other locality optimizing algorithms apply degree sorting throughout [4, 43, 45]. This destroys locality expressed in the initial assignment of vertex labels [36].

6 Conclusion and Future Work

This paper represents iHTL that improves temporal locality using both push and pull traversals in one graph traversal but for different vertex types. The evaluation on 10 real-world graph datasets shows that iHTL is much faster than pull and push traversals in graph processing frameworks. Furthermore, iHTL outperforms state-of-the-art locality optimization relabeling algorithms.

This paper concentrates on improving locality in pull traversal which is widely used in several graph analytics. However, the idea that **irregular datasets require irregular traversals** is not limited to pull traversal and can be useful for improving locality in other graph analytics like Triangle Counting, Single Source Shortest Path, and Connected Components. Moreover, iHTL can be improved by:

— The number of flipped blocks (Section 3.3) can be identified in an algorithm with lower complexity by limiting the maximum number of flipped blocks and applying a pass over out-edges of HV_1 to identify all $|HV_i|$.

— The size of topology data of iHTL graph (Section 4.4) can be reduced using light-weight graph compression techniques [9, 10] and vectorization [17, 38, 43].

— iHTL reduces cache misses of hubs and high-degree vertices. Locality optimizing relabeling algorithms like Rabbit-Order improve spatial locality of low-degree vertices [21]. In this way locality of the sparse block may improve by applying Rabbit-Order.

Code Availability

Source code repository and further discussions relating to this paper are available online in https://blogs.qub.ac.uk/GraphProcessing/Exploiting-in-Hub-Temporal-Locality-in-SpMV-based-Graph-Processing/.

7 Acknowledgments

This work is partially supported by the High Performance Computing center of the Queen's University Belfast and the Kelvin supercomputer (EPSRC grant EP/T022175/1) and by DiPET (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] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and counting given length cycles. *Algorithmica* 17 (1997), 354–364.

[2] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.

[3] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 12, 10 pages.

[4] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. 2017. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 32–41.

[5] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (Washington, DC, USA) *(HPDC '17)*. Association for Computing Machinery, New York, NY, USA, 93–104.

[6] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

[7] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.

[8] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2014. 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, 227–228.

[9] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. ACM, New York, NY, USA, 587–596.

[10] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. ACM, New York, NY, USA, 595–602.

[11] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Computer Networks* 30, 1-7 (April 1998), 107–117.

[12] Anna D. Broido and Aaron Clauset. 2019. Scale-free networks are rare. *Nature Communications* 10, 1 (Mar 2019).

[13] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*. Washington DC, USA.

[14] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. ACM, New York, NY, USA, Article 1, 15 pages.

[15] Charles L Clarke, Nick Craswell, and Ian Soboroff. 2009. *Overview of the trec 2009 web track*. Technical Report. DTIC Document.

[16] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1304–1318.

[17] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making Pull-based Graph Processing Performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. ACM, New York, NY, USA, 246–260.

[18] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 319–329.

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

[20] Jon M. Kleinberg. 1999. Authoritative Sources in a Hyperlinked Environment. *J. ACM* 46, 5 (Sept. 1999), 604–632.

[21] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. How Do Graph Relabeling Algorithms Improve Memory Locality?. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 84–86.

[22] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*. 1343–1350.

[23] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600.

[24] Yongsub Lim, U Kang, and Christos Faloutsos. 2014. SlashBurn: Graph Compression and Mining beyond Caveman Communities. *IEEE Transactions on Knowledge and Data Engineering* 26, 12 (Dec 2014), 3077–3089.

[25] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 201–213.

[26] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement* (San Diego, California, USA) *(IMC '07)*. Association for Computing Machinery, New York, NY, USA, 29–42.

[27] Sharan Narang, Gregory F. Diamos, Shubho Sengupta, and Erich Elsen. 2017. Exploring Sparsity in Recurrent Neural Networks. *CoRR* abs/1704.05119 (2017). arXiv:1704.05119

[28] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*.

[29] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 472–488.

[30] Youcef Saad. 1994. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2.

[31] Hiroaki Shiokawa, Tomokatsu Takahashi, and Hiroyuki Kitagawa. 2018. ScaleSCAN: Scalable Density-Based Graph Clustering. In *Database and Expert Systems Applications*, Sven Hartmann, Hui Ma, Günther Pernul, and Roland R. Wagner (Eds.). Springer International Publishing, Cham, 18–34.

[32] Friendster social network. [n.d.]. Friendster: The online gaming social network. archive.org/details/friendster-dataset-201107.

[33] Bor-Yiing Su, Tasneem G. Brutch, and Kurt Keutzer. 2010. Parallel BFS graph traversal on images using structured grid. In *2010 IEEE International Conference on Image Processing*. 4489–4492.

[34] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning. In *2017 46th International Conference on Parallel Processing (ICPP)*. 181–190.

[35] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. ACM, New York, NY, USA, Article 16, 10 pages.

[36] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2018. VEBO: A Vertex- and Edge-Balanced Ordering Heuristic to Load Balance Parallel Graph Processing. *CoRR* abs/1806.06576 (2018). arXiv:1806.06576

[37] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[38] Hans Vandierendonck. 2020. Graptor: Efficient Pull and Push Style Vectorized Graph Processing. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 13 pages.

[39] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 38–52.

[40] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315

[41] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, NewYork, NY, USA, 1813–1828.

[42] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. 2007. SCAN: A Structural Clustering Algorithm for Networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Jose, California, USA) *(KDD '07)*. ACM, New York, NY, USA, 824–833.

[43] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding up SpMV for Power-Law Graph Analytics by Enhancing Locality and Vectorization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 86, 15 pages.

[44] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-structured Analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. ACM, New York, NY, USA, 183–193.

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

[46] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 30 pages.

[47] Xiaojin Zhu and Zoubin Ghahramani. 2002. *Learning from Labeled and Unlabeled Data with Label Propagation.* Technical Report.