Connectivity-Aware Link Analysis for Skewed Graphs

YuAng Chen
The Chinese University of Hong Kong, Shenzhen
yuangchen@link.cuhk.edu.cn

Yeh-Ching Chung
The Chinese University of Hong Kong, Shenzhen
ychung@cuhk.edu.cn

ABSTRACT

Link analysis is a fundamental task for graph analytics, as it enables the identification of important nodes and patterns in the graph. Link analysis algorithms typically require traversing the graph and accessing the links of each node. However, for graphs with a skewed degree distribution, the computing efficiency of link analysis is severely constrained due to irregular connectivity, which results in randomized memory accesses and high cache miss ratio.

In this paper, we conduct a thorough investigation into skewed graphs' structural characteristics, focusing on their interaction with the computational patterns observed in link analysis algorithms and the underlying hardware architecture. Generalizing the understanding, we develop a novel framework named Mixen. Mixen applies a lightweight filtering procedure to enhance graph locality and reschedule computations. Different graph components are selectively processed under distinctive paradigms. Thereby, Mixen allows for efficient graph traversal and performance optimization.

We evaluate Mixen on modern multicore systems, comparing its performance to state-of-the-art frameworks across various graphs and algorithms. The results indicate that Mixen significantly outperforms its counterparts. Over the best-performing alternative, Mixen achieves a 3.42× speedup in execution time. Furthermore, we explore the design space of Mixen, examining its trade-offs and the correlation with cache-memory dynamics.

CCS CONCEPTS

• **Computer systems organization** → **Multicore systems**; • **Graph Processing**; • **Parallel Computing**;

KEYWORDS

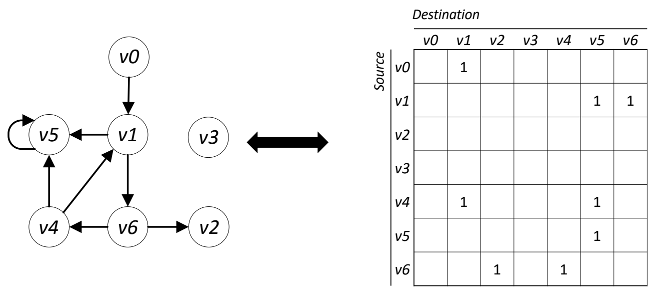
Skewed Graphs, Link Analysis Algorithms, Multicore Systems

ACM Reference Format:
YuAng Chen and Yeh-Ching Chung. 2023. Connectivity-Aware Link Analysis for Skewed Graphs. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605579

1 INTRODUCTION

Link analysis of graphs is a powerful tool for evaluating the relationships and connections between entities in a complex system. In this approach, the entities are represented as nodes in a graph, and the relationships between them are represented as links (or namely edges). Link analysis has been instrumental in understanding the real world and has been applied in diverse applications, such as web search [32], recommendation systems [21] and social network analysis [8].

Let the set of nodes be represented by V with size n, and the set of links is indicated by E with size m. The graph can be denoted by G=(V, E). The input for the link analysis algorithm is the adjacency matrix A of graph G, where $A[i, j]$ equals 1 if there exists a link from node i to node j, and 0 otherwise. The algorithm's output can be an n-dimensional vector y, where the i-th coordinate y_i represents the score of node i in the graph.



Source \ Destination	v0	v1	v2	v3	v4	v5	v6
v0		1					
v1						1	1
v2							
v3							
v4		1				1	
v5						1	
v6			1		1		

Figure 1: A sparse graph and its adjacency matrix

A naive heuristic, referred to as the InDegree algorithm, is often considered as the precursor to all link analysis algorithms [6]. The InDegree algorithm ranks nodes based on the count of links to them; that is, the popularity of a node equals to its in-degree. This algorithm has led to the development of various link analysis algorithms, such as PageRank [32], HITS [20], SALSA [27]. Besides, the InDegree algorithm can be translated into a SpMV operation $y = A^T x$, where A^T is the transposed matrix of A, and x is a vector recording initial properties of nodes. Based on the SpMV, advanced machine learning algorithms can be devised, including Collaborative Filtering [21] and Graph Neural Networks [19].

Despite the importance of link analysis algorithms in various fields, their computing efficiency is often hampered by the intricate structure of graphs. Graphs representing real-world phenomena, such as social networks [33] and website links [22], generally adhere to a skewed degree distribution [14], which can result in issues such as load imbalance, poor cache locality, and redundant memory traffic in multicore systems [11].

In skewed graphs, a substantial portion of links is connected by a small fraction of nodes, referred to as hubs. The presence of hot nodes results in a significant volume of memory traffic. The processor might repeatedly access these hubs, generating random and redundant jumps in memory space that bottlenecks the computation. By contrast, the remaining nodes exhibit sparse connectivity and thus offer limited reuse opportunities. In general, skewed graphs

ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0843-5/23/08…$15.00
https://doi.org/10.1145/3605573.3605579

exhibit irregular access patterns, leading to suboptimal spatial and temporal locality. In other words, accessing a node's connections does not ensure that subsequent node connections are closely stored in time or memory. This results in inefficient memory utilization and increased cache misses.

In this paper, we reveal the inefficiencies and redundancies hidden inside the skewed graphs, which are often ignored by existing graph traversal approaches. To address these issues, we propose a graph framework, named Mixen, that leverages the structural properties of skewed graphs to facilitate link analysis.

In summary, our contribution can be generalized into the following points:

- We analyze the structural characteristics of skewed graphs and their traversing pattern for link analysis algorithms. The performance limitations of graph processing, which arise from the irregular connectivity inherent to graphs, are described.
- Our proposed processing framework, Mixen, is designed to accelerate the link analysis of graphs. Mixen firstly applies a light filtering optimization to the input graph, allowing for efficient rescheduling of computation to avoid redundant memory traffic. Further, Mixen adopts mixed encoding schemes and processing paradigms to different graph portions. This enables the optimization of accessing individual subgraphs and thus achieves an overall performance gain.
- We theoretically demonstrate the effectiveness of Mixen in improving the memory utility. Performance models are derived to quantitatively describe Mixen's behavior, its impact on the multicore systems and its performance limitations.
- To validate the performance of Mixen, we conduct extensive experiments on multicores using a wide range of graphs and algorithms. Mixen is compared to state-of-the-art frameworks. Against the fastest among them, Mixen delivers speedups by 3.42×.

2 BACKGROUND

2.1 Structural Property of Skewed Graphs

In skewed graphs, a few nodes have significantly more links compared to the majority of nodes, which have relatively few links. The degree distribution of such graphs is characterized by a skewed distribution. Also, the skewed graphs can be referred to as power-law or scale-free graphs [14].

Table 1 lists the structural information of skewed graphs, as well as non-skewed graphs for comparison. Here, we define hubs as nodes with in-degrees greater than the average degree of the entire graph. Hubs account for less than 8% of the nodes in the skewed graphs, yet they constitute over 90% of the connections. The uneven distribution in skewed graphs presents both opportunities and obstacles when optimizing graph accessing. For instance, frequently-accessed hubs have a higher chance of remaining in the cache; whereas, their large number of processed messages may exhaust the cache and prevent other nodes from being reused.

Furthermore, many nodes in skewed graphs have no connections in at least one direction. Depending on the connectivity, nodes can be classified into four types : (1) *Regular* nodes have both incoming and outgoing links. (2) *Seed* nodes[1] have only outgoing links. (3) *Sink* nodes have only incoming links. (4) *Isolated* nodes have no links. The existence of zero-degree nodes is a natural outcome of imbalanced degree distributions in graphs: as some nodes acquire more links, other nodes will have fewer links. This fact is often ignored despite its significant impact on graph traversal, which is to be elaborated in Section 3.

Table 1: The structural characteristics of skewed and non-skewed graphs. V_{hub} and E_{hub} are the percentages of hubs and hubs' edges over the graph. The percentages of 4 types of nodes is listed.

	Graphs	V_{hub}	E_{hub}	Reg.	Seed	Sink	Iso.
Skewed	*weibo* [33]	1	99	1	99	0	0
	track [36]	5	88	46	54	0	0
	wiki [22]	11	88	22	33	45	0
	pld [31]	15	82	56	8	28	8
	rmat [9]	7	94	26	7	8	59
	kron [3]	8	92	49	0	0	51
Non-	*road* [22]	50	66	100	0	0	0
	urand [3]	52	59	100	0	0	0

In contrast, non-skewed graphs exhibit completely different feature. Nearly half of the nodes are identified as hubs, but these hubs only compose up to 2/3 of the total connections. Also, the two non-skewed graphs are bidirected, so all nodes are considered regular.

2.2 Link Analysis Algorithms

Link analysis algorithms are used to analyze the structure of a graph. The InDegree algorithm is the simplest link analysis algorithm. It measures the importance of a node based on the number of incoming links. PageRank [32] is arguably the most well-known ranking algorithm, serving as the foundational algorithm for Google's search engine in its early years. HITS assigns two scores to nodes that mutually reinforce each other [20], while SALSA uses a random walk approach to compute two scores per node [27]. Although these algorithms may differ in their specific criteria for determining the importance, they perform similarly to the InDegree algorithm [6]. For simplicity, we will use the InDegree algorithm to illustrate the computing pattern common to link analysis algorithms in the following context.

Algorithm 1 present memory-friendly approaches to implementing the InDegree algorithms. They are designed for graphs formatted in compressed sparse rows or columns (CSR or CSC). CSR and CSC are storage formats for sparse matrices. CSR uses row pointers and column indices, whereas CSC uses column pointers and row indices to keep track of the non-zero elements' positions in the original matrix. Both the CSR and CSC formats are widely used as default representations for graphs and matrices in various scientific computing systems [18, 40].

[1]Conventionally, these are referred to as *source* nodes. However, the term "source" is also used in reference to "destination" nodes to describe message direction. Therefore, to avoid confusion, the term "seed" is used instead.

Algorithm 1: The InDegree algorithm with CSR and CSC

```
1 for i = 0 to n-1 do in parallel                    ▷ Pushing Flow
2     for j = csrPtr[i] to csrPtr[i+1] - 1 do
3         atomAdd(y[csrIdx[j]], x[i])
4 ...
5 for i = 0 to n-1 do in parallel                    ▷ Pulling Flow
6     for j = cscPtr[i] to cscPtr[i+1] - 1 do
7         y[i] += x[cscIdx[j]]
```

With the CSR format, the InDegree algorithm can be processed in a pushing flow as shown in lines 2-4 of Algorithm 1. `x` and `y` are the input and output vectors, respectively. `csrPtr` and `csrIdx` correspond to the row pointers and column indexes of CSR. To avoid multiple threads writing to the same element in the output array simultaneously, an atomic instruction is deployed for the addition assignment. Alternatively, the InDegree algorithm can be executed in a pulling flow as at line 5-7, where `cscPtr` and `cscIdx` refer to the column pointers and row indexes of the CSC format. This approach eliminates the need for atomic operations. Thus, in comparison with the pushing flow using CSR, the pulling flow with CSC is the preferred method for implementing link analysis algorithms .

Algorithm 2: The InDegree algorithm with Blocking

```
1 for b ∈ blocks do in parallel
2     Scatter(b, bins)
3 for b ∈ blocks do in parallel
4     Gather(b, bins)
```

Blocking is another approach for eliminating the need for atomic operations while keeping message propagation in the pushing flow [4, 7, 25]. This technique partitions the graph into blocks (e.g., blocked CSR). Each thread is confined to accessing only a specific block, which reduces thread contentions. A buffering bin is allocated to every block to convert propagations to sequential memory accesses. The execution of a graph workload is scheduled into three phases: Scatter, Gather, and Apply (GAS). Algorithm 2 outlines the InDegree algorithm with blocking optimizations, where the Apply phase is omitted as it is not executed. During the Scatter phase, source nodes push data to the bins. In the Gather phase, the data in the bins are accumulated for the correct sum. Finally, in the Apply phase, user-defined functions are applied to update the sum.

3 MOTIVATION

This section presents the motivation for our work. We compare different execution paradigms of the link analysis algorithms, discussing their respective advantages and disadvantages. Meanwhile, we also analyze the redundancy and inefficiency resulting from the irregular connectivity of graphs.

The performance of graph processing is typically constrained by memory activities rather than arithmetic computations [4]. Thus, our investigation focuses on the memory behaviors of the pulling flow and the blocking approach. The pushing flow is ignored since it behaves similarly to the pulling flow, with the exception of requiring additional atomic instructions. Specifically, two memory activities are examined: memory traffic volume (i.e., read and write operations) and random memory access (i.e., non-sequential address jumps). To simplify our analysis, we assume that data types for nodes, links, and updates each occupy 1 Byte.

Memory Traffic Volume. The GAS paradigm of the blocking InDegree is intrinsically inefficient. This is due to the fact that, in each iteration, data transmissions are initiated twice. Firstly, the Scatter phase reads the CSR of size $n + m$ and the input array x of size n, and subsequently generates m data for writing to the bins. Then, the Gather phase loads m pairs of data and destination for accumulation, and writes n sums. As a result, the total traffic volume is $4m + 3n$ Bytes.

In contrast, the pulling flow propagates updates only once. The CSC of size $n + m$ is scanned, while the elements in x are loaded m times. The output array y of size n is then written back to memory. As a result, the memory traffic for the pull method is only $2m + 2n$ bytes, which is substantially lower than that for the GAS method.

Random Memory Access. Although the pulling approach reduces memory traffic, it suffers from highly randomized accesses. As illustrated in Algorithm 1, requests to the `cscPtr`, `cscIdx`, and `y` arrays are sequential, ensuring high cache locality. However, the reads to the `x` array are random, with up to m reads, resulting in poor cache utility.

The blocking approach, derived from the CSR-based pushing flow, sequentially accesses the `csrPtr`, `csrIdx`, and `x` arrays, while random writes are directed to the `y` array. Fortunately, the random writes can be substantially decreased through sorting the propagation with the bins [4, 25]. Data access within bins is sequential, with random access arising solely from block fetching (line 1 and 3 in Algorithm 2). Consequently, the number of random requests amounts to $(n/c)^2$, where c represents the cache size utilized to establish the block size.

Using the graph *wiki* with n = 18.2 M and m = 172.2 M as an example, we assume a cache size c = 64 KB, resulting in $(n/c)^2 \approx 285^2$ = 80.9 K blocks. Under the worst-case scenario, where locality is absent in *wiki*, accesses are entirely randomized. Approximately, the pulling InDegree incurs 172.2 M random accesses, while the blocking approach only causes 80.9 K accesses. However, the blocking approach generates an additional 362.6 MB of memory traffic compared to the pulling method.

In conclusion, the blocking technique reduces random memory accesses at the expense of increased memory traffic. For graphs exhibiting high locality, random accesses are inherently minimal, rendering the performance gains of blocking negligible or even deteriorated [4]. Based on the quantitative analysis, we are motivated to seek a solution that harnesses the advantages of both blocking and pulling methods for link analysis.

Redundant Message Passing. When represented as an adjacency matrix A, a seed node v_j corresponds to an empty column, denoted as $A[:, j] = 0$. Similarly, a sink node v_i signifies an empty row, expressed as $A[i, :] = 0$. An isolated node implies that both the corresponding row and column are empty.

Although CSR compresses the adjacency matrix by sequentially storing non-zero elements in each row, this format remains inefficient for graph traversal. For example, in the wiki graph, 45% of nodes (i.e., sink nodes) lack outgoing links, indicating that nearly half of the elements in the CSR pointer array are repeated if the graph is represented in CSR. These elements are unnecessarily scanned in every iteration. Analogously, if wiki is stored in CSC, approximately one-third of redundant elements (i.e., seed nodes) stored in the CSC offset array.

Furthermore, the directionality of irregular connectivity results in redundant memory traffic for the link analysis. During computation, regular nodes serve as central points due to their iterative updates. Seed nodes, however, maintain their data unchanged after initialization, leading to the repetitive broadcasting of the same data. Sink nodes, which do not impact other nodes, have their states determined solely by their in-neighbors. Consequently, propagation towards sink nodes can be delayed until the completion of other nodes in the final iteration. Lastly, isolated nodes are entirely independent and thus unnecessary.

The GAS employed by blocking approach amplifies redundancy, as the unnecessary traffic related to seed and sink nodes is doubled by the Scatter and Gather phases. In order to improve the performance of graph traversal, regardless of the employed method (e.g., either pulling or blocking), it is essential to eliminate the redundancy of message passing.

4 MIXEN

In this section, we propose Mixen, our solution for enhancing the performance of link analysis algorithms. Mixen aims to capitalize on the benefits of both blocking and pulling flows, while simultaneously addressing the irregular connectivity intrinsic to graphs.

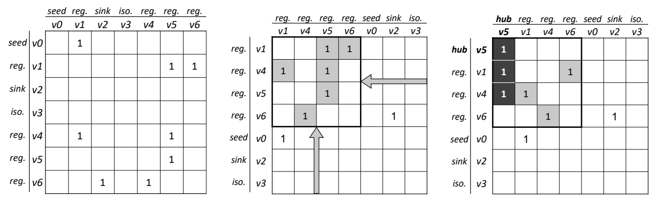
4.1 Graph Filtering and Relabeling

The goal of graph filtering is to avoid redundant memory traffic and improve cache locality. Also, the filtering process is expected to be lightweight, which can be compensated for by a few iterations of downstream applications.

To accomplish these objectives, we develop a strategy that (1) relabels node IDs based on their degrees and (2) then replacing the original nodes with the relabeled nodes in the memory space. The connectivity analysis presented in Section 2.1 offers valuable insights for devising the relabeling algorithm.

Firstly, Mixen exmaines whether a node has a zero-degree. If the node does, Mixens determines the type of the node based on its directionality. Next, regular nodes are assigned the first IDs, followed by seed nodes which are relabelled with the next available IDs. Similarly, sink and isolated nodes are allocated in subsequent own memory addresses. As a result, a subgraph consisting solely of regular nodes is produced, as depicted in Fig. 2b.

Secondly, Mixen implements an additional filtering step on the regular subgraph to enhance graph locality. Hubs within the subgraph, which are regulars nodes with in-degrees higher than the graph's average degree, are relocated to the front of non-hub nodes. This ensures that frequently accessed nodes are co-located, thereby increasing the likelihood of cache reuse. By evaluating hubs based on in-degree, nodes with numerous incoming links from regular and seed nodes are collected, which allows for more effective caching of seed-to-regular messages for later reuse (detailed in Section 4.3). Additionally, we preserve the relative order of nodes within each category in both steps, ensuring minimal disruption to the original graph structure.



(a) Original A **(b) 1st step** **(c) 2nd step**

Figure 2: 2-step filtering of graphs

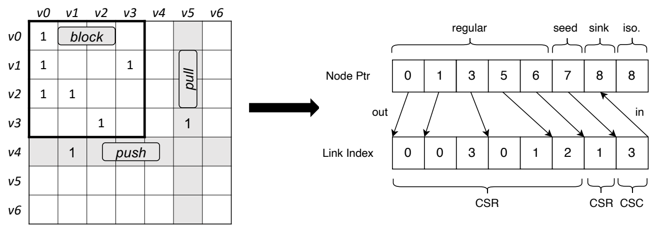


Figure 3: Mixed representation of CSR and CSC for the filtered graph. Regular nodes (including hubs) and seed nodes are encoded in CSR, and sink nodes are encoded in CSC. The three types of nodes are processed in different paradigms.

The subgraphs are represented using a mixed form of CSR and CSC. Regular nodes, including the hubs, are encoded in CSR, preparing for cache blocking. Seed nodes are encoded in CSR too, as seed nodes exclusively have outgoing links. In contrast, sink nodes are encoded in CSC, given that they only possess incoming links. As depicted in Fig. 3, the index array within the CSR part stores the out-neighbors (i.e., column ID) of nodes, while the index array within the CSC part stores the in-neighors (i.e., row ID) nodes. The boundary information separating the regular, seed, sink and isolated nodes are stored as meta-data to facilitate management during graph processing.

It should be emphasized that minimizing the cost of graph filtering is a primary concern in designing our optimization strategy. We reduce the overhead in two aspects. (1) The 2-step filtering procedure is merged into a single graph scan, thus decreasing both computational demands and data loading costs. (2) The CSR and CSC representations of the subgraphs are directly extracted from the existing CSR and CSC of the original graph. The extraction not only eliminates extra overheads of format conversion, but also reduces memory footprints compared to those required by the original CSR plus CSC. In such way, the filtering of the graph is streamlined with the optimization of data representation, delivering higher efficiency.

4.2 Graph Partitioning and Binning

The filtered subgraph of regular nodes is further optimized by 2-D blocking. The block size is determined by a cache-related indicator, denoted as c, which is optimized in Section 6.4. Given that the subgraph contains r regular nodes, the filtered subgraph is partitioned into $\frac{r}{c} \times \frac{r}{c}$ blocks. In other words, denoting $b = \frac{r}{c}$, the submatrix is converted to $b \times b$ blocks.

The cache-sized block serves as the basic data unit for threads to process. It imposes constraints on the thread behavior in two dimensions. Firstly, the horizontal width of the block specifies the node range for a thread to access. Secondly, the vertical height of the block limits the propagation range of updates. Thus, As a result, the access to the graph for each thread is controlled within the cache. The 2-D blocking can be easily implemented by partitioning the CSR of the filtered graph into multiple local CSRs in parallel.

Mixen improves graph locality by filtering the graph and relocating hubs to the beginning of the vertex set. However, the concentration of hubs may result in workload imbalance among threads. To tackle this issue, a load balancing scheme [12, 44] is employed. This scheme estimates the workloads by counting the number of nonzeros within each block. Mixen ensures that the total number of nonzeros per block does not exceed twice the average number of blocks. If a block becomes overloaded, it is divided into smaller blocks.

Moreover, Mixen utilizes *dynamic* and *static* bins for data buffering. The dynamic bins are updated in every iteration. They are used to sort the propagation [4], such that the random memory jumps within each block are converted into sequential memory access. Further, to reduce the memory traffic, the edge compression technique [25] is deployed, which compresses the messages from a single source node to multiple destination nodes into a single transmission within a block.

The static bins are writen only once for preparation and become read-only afterwards. They accumulate and cache the data sent from seed nodes to regular nodes. In subsequent executions, cached data are reused to avoid repeated computation and propagation. Moreover, static bins can be shared by multiple blocks with common rows (i.e., source nodes) because the cached data are identical among them. As a result, the static bins are allocated per block-row as 1-D vector. The operations relating to the bins are further detailed in Section 4.3.

4.3 Graph Scheduling and Propagating

Upon completion of the filtering and partitioning processes, Mixen proceeds to execute link analysis on the resultant subgraphs. The primary workload in graph processing stems from iterative executions on regular nodes, referred to as the *Main-Phase*. In addition, two supplementary phases involving seed and sink nodes contribute to the overall workload: the *Pre-Phase* and *Post-Phase*. The entire procedure is orchestrated in Algorithm 3, offering a systematic method to manage the irregular graph workloads.

Pre-Phase with Seed. During the preparation phase, seed nodes are initialized, and then transfer their data to destination nodes (line 3). Thereafter, they become inactive for future computations. Receiving blocks aggregate the messages and store the resulting data, which is utilized in the *Main-Phase*. To decrease the overhead of preparation, the *Pre-Phase* is incorporated into an earlier stage — the construction of blocks (line 2).

Main-Phase with Regular. The processing of regular nodes is conceptualized as an iterative Scatter-Cache-Gather-Apply (SCGA) model. This model operates on a $b \times b$ blocks and generates updates for regular nodes. Specifically, the Cache step is a novel addition proposed in this paper, where the prepared data in the Pre-Phase are reused by the regular nodes. It serves as a crucial mechanism to avoid redundant computations and communications associated with seed nodes.

The Scatter and Cache steps are grouped within a single parallel region (lines 5-9), which is parallelized in row-major order. In the Scatter step, the thread assigned with index i is responsible for updating all the dynamic bins belonging to the i^{th} block-row (i.e., `bin[i][:]`). The thread is restricted to only access a segment of source nodes' data `x`, the range of which corresponds to the row ID range of the i^{th} block-row. This step buffers the propagation from the source nodes to destination node, offering the benefit of squentialized memory accesses.

Algorithm 3: SpMV performed by Mixen

```
   Input: x, seed, sink, regular
   Output: y, sta_bin, dyn_bin
1  b = num_regular/cache_size
2  dyn_bin[b][b] = Block(regular)              ▷ Blocking
3  sta_bin[b] = Push(seed)                     ▷ Pre-Phase
4  while !converge & !max_iter do
5      for i ← 0 to b-1 do in parallel         ▷ Main-Phase
6          for j ← 0 to b-1 do
7              x update dyn_bin[i][j]          ▷ Scatter
8          x reset by sta_bin[i]               ▷ Cache
9      for i ← 0 to b-1 do in parallel
10         for j ← 0 to b-1 do
11             y accumulate dyn_bin[j][i]      ▷ Gather
12         for v ∈ regular do
13             Function(y[v])                  ▷ Apply
14     Swap(x,y)
15 Pull(sink)                                  ▷ Post-Phase
```

Following the Scatter step, the Cache step resets the properties of the regular nodes to the previously cached values. The data stored in the static bins are written consecutively to the `x` (segment). By reusing the cached values, the Cache step minimizes the need for redundant memory traffic and repeated computation.

The Gather and Apply steps are also combined within a single parallel region (lines 9-13), but are multi-threaded in a column-major order. In the Gather step, the i^{th} thread accumulates the cached data from the dynamic bins along the i^{th} block-column (i.e., `bin[:][i]`), and writes the aggregated sums to the destination nodes `y`. The access range of destination nodes is confined by the column ID of the i^{th} block-column.

Finally, the Apply step incorporates user-defined functions that operate on the sum generated in the previous step. The computational workload of the Apply and Cache steps is relatively small. The majority of the cost is incurred during the cross-block data movements in the Scatter and Gather steps, where bins spanning a column or row are sequentially accessed. Therefore, optimizing these steps is critical for improving the overall efficiency of graph processing.

Post-Phase with Sink. Upon completion of the Main-Phase, either as a result of convergence or reaching a pre-defined number of iterations, the processing transitions into the Post-Phase. During this phase, the sink nodes fetch data from both regular and sink nodes along the incoming links and subsequently compute their own updates. This final phase ensures the accurate calculation of updates for sink nodes.

5 PERFORMANCE ANALYSIS

This section undertakes a theoretical analysis of the Indegree algorithm implemented under Mixen to gain insights into Mixen's behavior and potential contributions.

The analysis primarily focuses on the iterative computation of regular nodes, given that other nodes are processed at most once. In order to simplify the analysis, the influence of the bit mask in reducing cross-block messages is not taken into account. Additionally, we make the assumption that the input graph exhibits no locality, implying that every link would invoke a random memory access. The ratio of regular nodes to the total number of nodes is denoted as $\alpha = r/n$. The number of links within the subgraph of regular nodes is denoted as $\tilde{m}$, while the proportion of these links relative to all links is $\beta = \tilde{m}/m$.

Memory Traffic Volume. During the Scatter step, a scan of regular nodes involves reading r Bytes of updates and r Bytes of destination nodes, followed by writing $\tilde{m}$ Bytes into the data array of bins. In the Cache step, regular nodes are reset using the cached data, resulting in $\tilde{m}$ Bytes of reads and writes, respectively. The Gather step reads $\tilde{m}$ Bytes from the bins and updates r Bytes of aggregated sums, followed by writing r Bytes for destination nodes. The data movement regarding the Apply step is zero, as no user-defined functions are involved in this process. Therefore, the total volume of memory traffic is:

$$\begin{aligned} mem &= 4r + 4\tilde{m} \\ &= 4\alpha \cdot n + 4\beta \cdot m \end{aligned} \tag{1}$$

The memory traffic of Mixen is highly dependent on the proportion of regular nodes α and their connectivity β. As the number of regular nodes decreases ($\alpha \rightarrow 0, \beta \rightarrow 0$), memory consumption is reduced. Conversely, in the worst-case scenario where an undirected graph contains no isolated nodes ($\alpha = 1, \beta = 1$), the advantages offered by Mixen are diminished. In such a situation, Mixen incurs higher memory traffic (e.g., $4n + 4m$) compared to the baseline blocking method discussed in Section 3 (e.g., $3n + 4m$), owing to the additional Cache step.

Random Memory Access. Within a block, the access to data and destination arrays is entirely sequential, thereby promoting high memory efficiency. Random access takes place when a thread switches a bin for updating or accumulation. Given that there are a total of $b \times b$ bins (one bin per block), the number of random memory accesses per iteration can be expressed as:

$$rand = O(b^2) = O((\frac{\alpha \cdot n}{c})^2) \tag{2}$$

The proportion of regular nodes, denoted as α, regulates the randomness by adjusting the number of blocks based on its square number α^2. Consequently, Mixen is anticipated to exhibit high effectiveness on graphs characterized by a low α value. In contrast, when $\alpha = 1$, the performance deteriorates to that of conventional blocking approach, which is $(n/c)^2$.

The above discussion does not consider the locality of the graph, which plays a substantial role in performance. Hence, in real-world implementations, the experimental results might deviate from our analysis. However, this discussion underscores the importance of understanding the underlying graph structure and its implications for the performance of graph processing.

6 EVALUATION

6.1 Setup and Implementation Details

Experiments are conducted on a modern multicore machine. It contains a two-socket Intel Xeon Silver processor with 20 physical cores. The capacity of L1 cache, L2 cache, LLC and main memory are 64KB, 1MB, 27.5MB and 256GB respectively.

Table 2: Graph datasets and their attributes (M: Million).

Graphs	n	m	Skewed	Real	Directed	α	β
weibo [33]	5.8M	261.3M	Yes	Yes	Yes	0.01	0.06
track [36]	12.8M	140.6M	Yes	Yes	Yes	0.46	0.60
wiki [2]	18.2M	172.2M	Yes	Yes	Yes	0.22	0.78
pld [26]	42.9M	623.1M	Yes	Yes	Yes	0.56	0.84
rmat [9]	8.4M	134.2M	Yes	No	Yes	0.26	0.59
kron [3]	67.1M	2.1B	Yes	No	No	0.49	1
road [22]	23.9M	57.7M	No	Yes	No	1	1
urand [3]	8.4M	268.4M	No	No	No	1	1

Our code[2] is implemented in C++ and complied using G++ 9.3.0 with O3 optimization. The parallelization is enabled by OpenMP [13]. During the graph processing, the dynamic scheduler is employed to facilitate workload balance. The data types of nodes, links and properties are all of 32 bits, e.g., `unsigned int` or `float`.The size[3] of a block is set to 256 KB, which contains 64K nodes. Meanwhile, 20 threads are employed for parallelization (with Hyper-Threading disabled).

In order to obtain a convincing result, we conduct experiments comprising 100 iterations (with convergence condition removed, if possible), and report the average outcome. The cache performance is measured using Linux's `perf` tool [41], while memory activities are monitored by `Liwid` [39].

The applications are assessed using a diverse range of graph datasets, covering both real-world and synthetic graphs, as well as directed and undirected ones. This variety ensures a comprehensive evaluation of the performance across different types of graph structures. Graphs *weibo* [33], *track* [36], *wiki* [22] and *pld* [31] are crawled from social network or hyperlinks in webpages. For these graphs, the isolated nodes are usually discarded before being open-sourced. Graphs *rmat* [9] and *kron* [3] are manufactured using specific graph generators. Their attributes are detailed in Table 2.

Mixen is evaluated in comparison to four other frameworks. Ligra [37] is a vertex-centric graph framework specifically designed

[2]https://github.com/yuang-chen/Mixen-ICPP-23

[3]since a block is a square matrix, we use the side length s in one dimension to represent its actual size $s \times s$.

Table 3: Graph processing time in seconds (per iteration except for BFS).

InDegree									PageRank								
Frwk	*weibo*	*track*	*wiki*	*pld*	*rmat*	*kron*	*road*	*urand*	Frwk	*weibo*	*track*	*wiki*	*pld*	*rmat*	*kron*	*road*	*urand*
Mixen	**0.008**	**0.091**	**0.085**	**0.116**	**0.061**	**0.366**	**0.017**	**0.076**	Mixen	**0.009**	**0.096**	**0.087**	**0.121**	**0.062**	**0.377**	**0.021**	**0.080**
GPOP	0.205	0.226	0.115	0.204	0.069	0.556	0.023	0.083	GPOP	0.251	0.266	0.132	0.212	0.077	0.604	0.031	0.090
Ligra	6.000	1.575	1.025	0.840	0.305	3.330	0.031	0.352	Ligra	10.850	3.015	1.940	1.490	0.605	5.750	0.041	0.650
Polymer	4.880	1.365	0.290	0.604	0.185	0.515	0.590	0.289	Polymer	5.300	1.575	0.398	0.166	0.455	0.690	0.253	0.398
GraphMat	0.407	0.182	0.167	0.689	0.089	1.857	0.089	0.174	GraphMat	0.438	0.190	0.190	0.727	0.096	2.019	0.098	0.192
Collaborative Filtering									Breadth-First Search								
Frwk	*weibo*	*track*	*wiki*	*pld*	*rmat*	*kron*	*road*	*urand*	Frwk	*weibo*	*track*	*wiki*	*pld*	*rmat*	*kron*	*road*	*urand*
Mixen	**0.025**	**0.159**	**0.157**	**0.159**	**0.101**	**0.514**	**0.025**	**0.095**	Mixen	0.094	**0.072**	0.229	0.640	0.202	0.720	1.006	0.190
GPOP	0.312	0.321	0.175	0.264	0.108	0.732	0.027	0.115	GPOP	0.062	0.173	0.326	0.956	0.126	1.731	1.199	0.278
Ligra	8.450	2.775	1.635	1.320	0.480	4.635	0.037	0.540	Ligra	**0.020**	0.091	**0.095**	**0.375**	**0.026**	**0.290**	**0.787**	**0.055**
Polymer	-	-	-	-	-	-	-	-	Polymer	1.930	1.610	0.250	1.370	1.740	0.565	11.500	2.000
GraphMat	1.137	0.577	0.561	3.044	0.280	7.776	0.406	0.835	GraphMat	0.521	1.019	1.003	3.488	5.746	4.338	65.208	0.647

for shared-memory systems, facilitating the intuitive expression of graph algorithms. Polymer [43] enhances Ligra by optimizing it for the Non-Uniform Memory Access (NUMA) in contemporary multicore architectures. GraphMat [38] transforms graph-related tasks into matrix-based operations. GPOP [24] divides graphs into blocks to boost cache efficiency, based on which Mixen is developed.

For performance evaluation, we test four graph algorithms. InDegree (IN) and PageRank (PR) are commonly-used link analysis algorithms, while Collaborative Filtering (CF) is a graph learning algorithm derived from the SpMV form of InDegree. However, we were unable to implement CF under Polymer because this would require thousands of lines of codes. Additionally, Breadth-First Search (BFS) is a non-link-analysis algorithm that does not benefit from the Cache step of Mixen at all. The execution of BFS ends when results reaches to convergence. The inclusion of BFS provides a comprehensive assessment of Mixen's applicability across diverse graph tasks.

6.2 Execution Time

Foremost, we evaluate the execution time of Mixen in comparison with other frameworks, the detail of which is listed in Table 3. On average of all results, Mixen outperforms GPOP, Ligra, Polymer, GraphMat by 3.42×, 7.81×, 19.37×, 7.74× respectively.

Ligra exhibits poor performance in link analysis tasks due to its reliance on pushing flows and atomics for data propagation. However, it proves advantageous in the context of BFS, where communication is sparse and collisions are less likely to occur. Polymer, a framework derived from Ligra, outperforms its predecessor in link analysis tasks by evenly redistributing graph data across NUMA nodes. Despite this improvement, the adopted approach deteriorates the performance of BFS, resulting in a trade-off between the selected algorithms.

GraphMat facilitates message propagation in the pulling flow, while being oblivious of the underlying hardware architecture. It often achieves better performance than Ligra and Polymer for link analysis tasks. Nonetheless, GraphMat exhibits suboptimal performance in BFS.

GPOP, being the second fastest framework, is designed with the awareness of cache hierarchy. It optimizes graph processing by organizing propagation into cache-fitting blocks. However, GPOP does not adequately address the irregular connectivity patterns typically observed in power-law graphs, which leads to redundant communications and computations.

Mixen leverages the connectivity of graphs, integrating the advantages of optimization techniques proposed in previous works. By reorganizing graphs, Mixen is able to extract the locality of graphs and to efficient subdivide the graph into subgraphs. The most appropriate approach is used for each individual subgraph, thereby optimizing the overall performance.

Furthermore, although Mixen was originally designed for directed, skewed graphs and link analysis algorithms, it successfully enhances the performance across various other graph types and applications. As a derivative, Mixen outperforms GPOP in terms of undirected graphs (e.g., *road*), non-skewed graphs (e.g., *urand*) and sparser algorithms (e.g., BFS) in most of cases.

6.3 Memory and Cache Utility

In this section, we examine Mixen's memory activities by comparing it with two variants. One variant solely applies the pulling flow to the graph as GraphMat, which referred to as Pull. The other exclusively adopts the blocking strategy as GPOP for the entire graph, which is denoted as Block. The use of these variants minimizes the impact of differing design principles across frameworks, therefore contributing to a more accurate and fair performance evaluation.

Fig. 4 demonstrates that Mixen produces the least amount of traffic, particularly on Graph *weibo* where the majority of traffic is scheduled out of the main phase. Moreover, a notable correlation is observed between memory traffic and execution time across different processing approaches.

The Pull variant frequently generates the highest memory traffic compared to other methods, resulting in reduced processing speed. However, an exception occurs with the graph *road*, which is characterized by a low maximum degree and a large diameter. Accessing this graph induces infrequent and long-distance jumps in memory

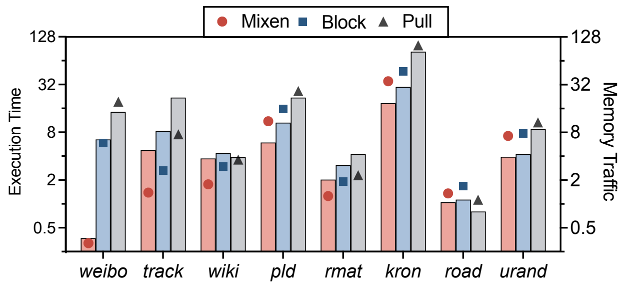


Figure 4: Normalized execution time (plotted as bars) and normalized memory traffic (plotted as dots) across Mixen as well as its Block and Pull variants.

space, inherently showing low temporal and spatial locality. In such instances, the Pull variant generates minimal memory traffic and yields better performance than the Block variant.

The Block variant restricts each thread's access range to a specific block, effectively minimizing random jumps within the memory space and retaining operations in cache. Consequently, the Block variant outperforms the Pull variant on skewed graphs that offer locality opportunities to exploit. Mixen further refines the basic blocking technique by eliminating unnecessary memory traffic. As a result, Mixen consistently achieves the higher processing speed.

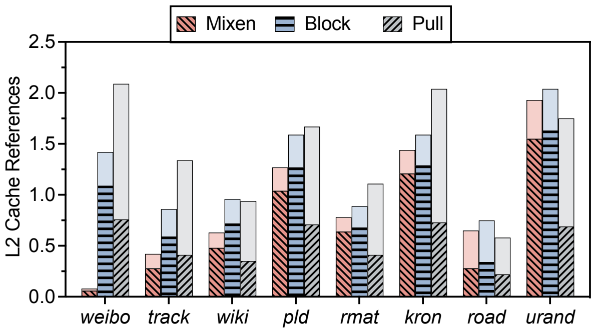


Figure 5: Normalized L2 cache references. Upper empty bars: cache misses; lower shadowed bars: cache hits.

We further investigate into the references (i.e., hits + misses) to the private L2 cache owned by each core. For the Pull variant, cache misses account for a major portion of references, amounting to 62%. In contrast, the cache miss ratios for Mixen and its Block variant are significantly lower, at 27% and 29% respectively. The excessively high cache misses is a crucial factor contributing to the high memory traffic observed in the Pull variant.

Mixen promotes more efficient cache utilization in two aspects. First, a large volume of message passing is avoided. This reduces the numbers of data and arithmetic instructions throughout the cache hierarchy. Hence, Mixen generates less cache references. Second, hubs are relocated to the beginning of the vertex set, allowing frequently accessed nodes to be co-located. Thereby, both the spatial and temporal locality of the data are improved, facilitating lower cache miss ratio.

6.4 Block Size

The block size plays a important role in determining the performance of Mixen. The choice of block size signifies a sophisticated tradeoff between workload distribution, cache locality and memory traffic. In Fig. 6, experimental results demonstrate that the size of a block best fits within either L2 cache or L1 cache.

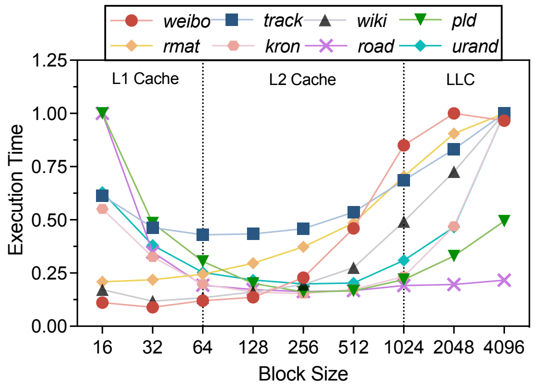


Figure 6: The normalized execution time with varied block size.

For graphs fitting within L1 cache, the key factor impacting the block size is the workload distribution among threads. When running multi-threaded programs, it is essential to generate a sufficient number of tasks to keep the threads occupied and prevent any thread from idling. In the case of Mixen, we found that effective parallelization during the Main-Phase occurs when the number of blocks is at least four times greater than the number of threads. Thus, for graphs that contain only a limited number of regular nodes, it is necessary to choose a correspondingly small block size to create enough blocks.

For graph fitting within L2 cache, the number of regular nodes is large enough to generate a sufficient number of blocks to feed the threads. In such case, other factors become more influential in determining the performance. The optimal performance is reached when these factors reach a balanced point, ensuring the most effective compromise.

Fig. 7 offers a concrete example of graph *pld*, which illustrates the interactions among different performance factors. When the block size is too small, such as 16KB, the accesses to LLC and memory are exceptionally high. In this scenario, the blocking technique cannot effectively enhance data reuse in caches. As the block size increases, the utilization of LLC and memory improves, resulting in a shorter execution time. Nonetheless, if the block size becomes overly large, such as 1MB, the performance would deteriorate again. Although the optimal results for LLC and memory efficiency are attained at different points, the best overall performance is achieved when both factors are accommodated within the L2 cache, at a block size of approximately 256KB.

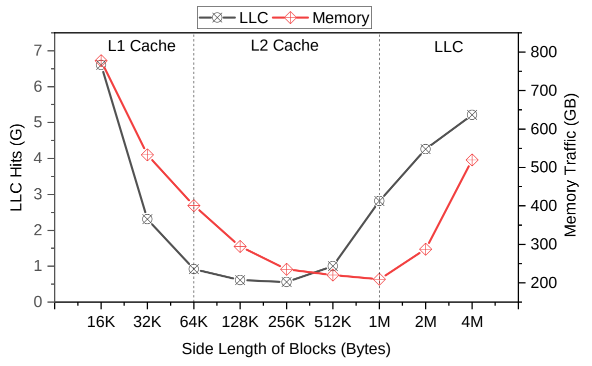


Figure 7: LLC hits and memory traffic with different block sizes for graph *pld*

6.5 Preprocessing Overhead

In this section, we evaluate the overhead associated with Mixen, which can be attributed to two primary factors: (1) graph filtering, and (2) graph partitioning.

Table 4: The preprocessing overheads of frameworks

Graph	GPOP	Ligra	Polymer	GraphMat	Mixen		
					Filter	Partition	Total
weibo	0.94	31.30	26.20	32.64	2.67	0.17	2.85
track	0.55	11.50	10.90	18.30	1.58	0.22	1.80
wiki	0.71	12.80	17.40	21.53	1.31	0.29	1.60
pld	6.32	32.70	65.60	80.14	4.73	2.81	7.54
rmat	0.69	6.88	11.30	18.70	0.91	0.74	1.65
kron	16.45	118.00	188.00	325.14	13.31	4.62	17.92
road	0.88	12.90	31.10	33.97	0.68	0.17	0.85
urand	0.19	1.28	4.88	7.87	1.73	0.73	2.46

Table 4 lists the preprocessing time required by the frameworks. Ligra, Polymer, and GraphMat require a significant amount of time to convert a graph from edge lists into their own customized formats. In contrast, Mixen and GPOP directly accept the CSR binary format without needing any format conversion. Thus, the overheads of Ligra, Polymer, and GraphMat are substantially higher than those of GPOP and Mixen. GPOP achieves lower overhead compared to Mixen due to its simpler optimization strategy. The primary source of Mixen's overhead stems from the filtering process, during which the entire graph is reallocated within the memory space.

7 RELATED WORK

SpMV. SpMV has been drawing enormous research attention over the past decades for diverse computing architectures, including CPUs [42] and GPUs [16]. The essential challenge of SpMV arises from the massive number of zeros in the sparse matrix, which are not engaged in the computation. Hence, the matrices are commonly represented in compressed formats that contain only the non-zeros [5].

However, there is no single format universally suitable for all SpMV algorithms on all architectures. The selection of matrix representation depends on the specific tasks. For instance, when the non-zeros are well structured and form a few matrix diagonals, Diagonal (DIA) is a favored choice [35]. Ellpack (ELL), allowing a general pattern of matrices, suits for regular SIMD programming [17]. For the irregular matrices, Coordinate List (COO) copes with the problem of load imbalance [15]. Furthermore, Hybrid (HYB) approach is devised by decomposing the matrix into two parts of regular ELL and irregular COO [1].

The duality between graphs and matrices remarkably stimulates the advancement in both domains. Graph algorithms, such as PageRank and BFS, serve as the representative SpMV applications in SpMV papers [28]. In reverse, optimized SpMV operations also inspire the high performance of graph processing [38].

Graph Processing. Mixen belongs to the category of shared-memory graph processing. In this case [10, 24, 37], a modern server is deployed, equipped with sufficiently large memory and multiple cores, for example, 1 TB and 128 cores. Thus, graphs can be fitted into the memory and globally shared by all threads.

When the memory volume of multicore systems is not able to accommodate large-scale graphs, the graph processing can be scaled up to distributed systems [46]. The distributed-memory processing requires explicit graph partitioning, so that subgraphs are locally handled by each node. However, the synchronization among subgraphs incurs expensive communication overheads across the network. Thus, the distributed work sometimes deliver suboptimal performance in comparison with the shared-memory counterpart [46] or even a single-threaded implementation [30].

Besides, the graph processing can also be offloaded to disks when the dataset can not fit into the memory [23, 34], which is defined as the external-memory system. The disk stores partial graph data and intermediate results during the execution of programs. A main design concern for such system is the efficient utilization of disk (e.g., improving disk I/O bandwidth) for data streaming [29].

8 CONCLUSION

In this paper, we investigate the computing pattern of link analysis algorithms with skewed graphs on shared-memory multicore systems. We identify the strengths and weaknesses of conventional methods for executing link analysis algorithms, as well as the redundancies and inefficiencies resulting from the inherent irregular connectivity within the skewed graphs.

Then, we propose Mixen, a framework designed to optimize link analysis algorithms for skewed graphs. Mixen firstly filters the graph based on node connectivity and encodes subgraphs with a mix of CSR and CSC. The filtered subgraphs are then rescheduled and processed under selectively tailored paradigms. In particular, a Scatter-Cache-Gather-Apply (SCGA) execution model is devised for the regular nodes, which avoids redundant computations and traffic.

The performance of Mixen is evaluated on diverse graph datasets and algorithms. The experimental results demonstrate that Mixen significantly outperforms contemporary works, not only for skewed graphs and link analysis algorithms, but also for non-skewed graphs and sparser algorithms such as BFS.

In the future, Mixen can be extended to contemporary graph systems, such as GraphMat [38] and Graphit [45], for performance improvement. Also, it is worth to explore the effectiveness of blocking technique and the SCGA model on GPUs.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1003505).

REFERENCES

[1] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M Tsai, and Weichung Wang. 2020. Load-balancing sparse matrix vector product kernels on GPUs. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–26.

[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web*. Springer, 722–735.

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

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

[5] Ronald F Boisvert, Ronald F Boisvert, and Karin A Remington. 1996. *The matrix market exchange formats: Initial design*. Vol. 5935. US Department of Commerce, National Institute of Standards and Technology.

[6] Allan Borodin, Gareth O Roberts, Jeffrey S Rosenthal, and Panayiotis Tsaparas. 2005. Link analysis ranking: algorithms, theory, and experiments. *ACM Transactions on Internet Technology (TOIT)* 5, 1 (2005), 231–297.

[7] Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai-Ching Tuan. 2016. Optimizing sparse matrix-vector multiplication for large-scale data analytics. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.

[8] Peter J Carrington, John Scott, and Stanley Wasserman. 2005. *Models and methods in social network analysis*. Vol. 28. Cambridge university press.

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

[10] YuAng Chen and Yeh-ching Chung. 2021. HiPa: Hierarchical Partitioning for Fast PageRank on NUMA Multicore Systems. In *50th International Conference on Parallel Processing*. 1–10.

[11] YuAng Chen and Yeh-Ching Chung. 2021. Workload Balancing via Graph Reordering on Multicore Systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 5 (2021), 1231–1245.

[12] YuAng Chen and Yeh-Ching Chung. 2023. An Unequal Caching Strategy for Shared-Memory Graph Analytics. *IEEE Transactions on Parallel and Distributed Systems* (2023).

[13] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[14] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review* 29, 4 (1999), 251–262.

[15] Goran Flegar and Hartwig Anzt. 2017. Overcoming load imbalance for irregular sparse matrices. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*. 1–8.

[16] Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 769–780.

[17] Roger G Grimes, David R Kincaid, and David M Young. 1979. *ITPACK 2.0 user's guide*. Center for Numerical Analysis, Univ.

[18] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. SIAM.

[19] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[20] Jon M Kleinberg. 1999. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)* 46, 5 (1999), 604–632.

[21] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.

[22] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. 1343–1350.

[23] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a {PC}. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 31–46.

[24] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2020. GPOP: A Scalable Cache-and Memory-efficient Framework for Graph Processing over Parts. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–24.

[25] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. 2018. Accelerating pagerank using partition-centric processing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 427–440.

[26] Oliver Lehmberg, Robert Meusel, and Christian Bizer. 2014. Graph structure in the web: aggregated by pay-level domain. In *Proceedings of the 2014 ACM conference on Web science*. 119–128.

[27] Ronny Lempel and Shlomo Moran. 2001. SALSA: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems (TOIS)* 19, 2 (2001), 131–160.

[28] Min Li, Yulong Ao, and Chao Yang. 2020. Adaptive SpMV/SpMSpV on GPUs for input vectors of varied sparsity. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1842–1853.

[29] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. 527–543.

[30] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what {COST}?. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[31] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science* 1 (2015).

[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.

[33] Ryan A Rossi and Nesreen K Ahmed. 2016. An interactive data repository with visual analytics. *ACM SIGKDD Explorations Newsletter* 17, 2 (2016), 37–41.

[34] 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*. 472–488.

[35] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[36] Sebastian Schelter and Jérôme Kunegis. 2016. Tracking the trackers: A large-scale analysis of embedded web trackers. In *Tenth International AAAI Conference on Web and Social Media*.

[37] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[38] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241* (2015).

[39] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 207–216.

[40] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[41] Vincent M Weaver. 2013. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, Vol. 13. 5.

[42] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. Cvr: Efficient vectorization of spmv on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 149–162.

[43] 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*. 183–193.

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

[45] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[46] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 301–316.

Received 22 April 2023; revised 1 June 2023; accepted 16 June 2023