2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3195-0602-3/26/$31.00 ©2026 IEEE | DOI: 10.1109/IPDPS65963.2026.00088

Rebo: Locality-Aware Graph Processing via Reordering and Blocking

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*—Graphs are widely used in real-world applications, but graph processing often suffers from poor data locality. Prior works address this issue by reordering vertices to group frequently accessed vertices or by partitioning adjacency matrices into cache-sized blocks. However, simply stacking these techniques can degrade performance on real-world graphs. This paper presents Rebo, a unified framework that effectively integrates graph reordering and matrix blocking. Rebo first reorders the graph to extract dense and sparse subgraphs, capturing locality patterns. The dense subgraph is then divided into cache-friendly blocks for improved cache utilization and load balance, while the sparse subgraph is stored in cache-line-aligned arrays to reduce memory indirection. Rebo processes the dense and sparse regions using partition-centric and vertex-centric paradigms, respectively. Evaluated across diverse datasets and applications, Rebo outperforms the second-best framework by 1.54×. Further analysis of instruction efficiency, cache behavior, and workload balance validates its effectiveness in addressing key challenges in graph processing.**

***Index Terms*—Graph Analytics, Parallel Computing**

I. Introduction

Graphs are non-linear data structures composed of vertices and edges that model relationships in diverse domains, from social networks [1], social influence [2], to biological pathways [3]. A graph with n vertices and m edges can be represented as a matrix $\mathbf{A}$ of size $n \times n$, where a non-zero element $a_{i,j}$ denotes an edge from vertex v_i to v_j. Leveraging the duality between graphs and matrices, graph algorithms are often formulated as sparse matrix operations for optimization [4].

Real-world graphs commonly exhibit a *power-law* degree distribution [5], posing remarkable performance challenges. A small number of "hot" vertices with vast connections cause severe load imbalance, synchronization overhead, and cache spilling due to their computational burden and communication costs. On the other hand, numerous "cold" vertices with few connections result in low computation intensity, introducing significant scheduling and coordination complexity in managing a massive volume of tiny tasks.

Moreover, graph processing typically suffers from poor data locality, both *spatial* and *temporal*, due to the irregular and sparse nature of graphs. Message passing along edges incurs indirect and random memory accesses to scattered vertex and edge data. This prevents sequential reads of contiguous memory blocks, hindering spatial locality. For a large fraction of vertices, particularly cold ones, data is not accessed frequently or predictably in a short time frame, leading to limited temporal locality.

TABLE I: Summary of graph optimization techniques.

Technique	Target Paradigm	Processing Granularity	Optimization Focus
Graph Reordering	Vertex-Centric	Vertex	Spatial Locality
Matrix Blocking	Partition-Centric	Partition	Temporal Locality

In order to optimize the *spatial* locality of graphs, a variety of graph reordering algorithms are designed [6]–[10]. Vertex IDs are relabeled to place neighboring vertices in consecutive memory addresses. Consequently, the computations are rescheduled to maximize the reuse of cached data. Since graph reordering focuses on modifying vertex order, it is primarily designed for and evaluated in *vertex-centric* graph processing frameworks, such as Ligra [11], Polymer [12], Gemini [13], Galois [14] and GraphIt [15]. In these frameworks, graph processing is executed and parallelized at the granularity of vertex.

To improve the *temporal* locality, the adjacency matrix of the graph can be processed as smaller cache-fitting blocks [16]–[18]. For instance, Cache Blocking splits matrices into blocks that fit within the cache, enabling repeated reads and writes within caches [19]. Binning further enhances the locality by allocating the blocks with sorted bins to ensure sequential data accesses [16], [17]. The idea of matrix blocking is adopted by the *partition-centric* graph processing frameworks, where the graph is divided into cache-sized subgraphs to regularize memory access patterns. In this paradigm, graphs are processed and parallelized at the granularity of partition. Examples include Cagra [20], GPOP [18] and CoroGraph [21].

This paper focuses on exploring the potential for combining the two distinct graph optimization techniques: *graph reordering* and *matrix blocking* (summarized in Table I). While these techniques have shown individual benefits in improving spatial and temporal locality, our initial analysis reveals that a naive combination often fails to deliver performance gains and, in some cases, performs worse than conventional vertex-centric methods. The observed performance degradation results from the lack of differentiation in handling hot and cold vertices, highlighting the need for a more refined integration strategy. This leads to the core research question: *how to simultaneously*

1530-2075/26/$31.00 ©2026 IEEE
DOI 10.1109/IPDPS65963.2026.00088

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

improve both spatial and temporal locality while effectively addressing the distinct processing patterns of hot and cold vertices in power-law graphs.

To address this challenge, we propose *Rebo*, a novel method that seamlessly integrates graph- and matrix-based optimizations for accelerating graph processing. These two techniques work together to transform the graph's adjacency matrix into distinct regions with varying memory access patterns. Based on the transformation, a selective execution model is designed to process each region differently, effectively minimizing instruction overhead, memory traffic, and cache misses. To summarize, our contributions can be generalized as follows:

- We investigate the interaction between graph-based and matrix-based techniques for locality improvement of power-law graphs. We showcase that their simple combinations fail to achieve satisfactory results.
- To overcome the issue, we propose *Rebo*, a synergistic integration of graph reordering and matrix blocking optimizations to accelerate graph processing.
- Rebo encapsulates a set of optimizations, including locality enhancement, cache-friendly graph transformation and selective execution model.
- The performance of Rebo is evaluated across 9 power-law and 2 non-power-law graphs. Popular graph frameworks and SpMV implementations are evaluated for comparison, demonstrating Rebo's effectiveness. Also, Rebo's limitation and overheads are clarified.

II. Background

A. Graph Traversing Pattern

In large-scale graphs, the number of vertices can exceed millions, while most cold vertices are connected with only double-digit neighbors. Consequently, the graph's adjacency matrix $\mathbf{A}$ is highly sparse, with the majority of its elements being zero. To store $\mathbf{A}$ efficiently, a compact representation is required. Common formats include Compressed Sparse Rows (CSR) and its transpose, Compressed Sparse Columns (CSC).

In CSC, each column of $\mathbf{A}$ corresponds to a vertex in the graph, and the non-zero elements in that column represent the incoming edges to that vertex. The format is defined by two main arrays: `colPtr` and `rowNdx`. For each column of $\mathbf{A}$, the `colPtr` array identifies the range of positions in the `rowNdx` array. The `rowNdx` array stores the row indices of nonzeros (i.e., the source vertices of the edges) column by column. For unweighted graphs, `colPtr` and `rowNdx` are sufficient to represent $\mathbf{A}$ as a Boolean matrix. For weighted graphs, an additional array `val` is used to record the edge weights. Details of the CSR format are omitted for brevity.

Consider a typical scenario of graph processing, where each vertex gathers messages from its incoming edges and performs an aggregation. The pulling flow can be expressed as in Algo. 1, which executes $\mathbf{A}$ column by column. This approach processes the graph in a pulling flow. During the computation, the random memory accesses mainly result from the random reads to the input vector $\mathbf{x}$, which are determined by the values of `rowNdx[j]`.

Algorithm 1 Pulling flow based on CSC

```
1: for i=0 to n-1 do parallel
2:     y[i]=0
3:     for j=colPtr[i] to colPtr[i+1]-1 do
4:         y[i]+=val[j]·x[rowNdx[j]]
```

The pulling flow is well-suited for link-intensive graph processing tasks, such as PageRank, where the majority of vertices are visited in every iteration. In contrast, the pushing flow, in which source vertices push updates to their out-neighbors, as shown in Algo. 2, is more effective for sparser, dynamic graph computations (e.g., BFS) with an active frontier, as it relies on atomic instructions to avoid write conflicts [22], [23].

Both the pulling and pushing flows process graphs from a vertex-centric perspective, where each vertex independently computes and communicates with its neighbors. This approach allows for intuitive expression of graph algorithms and forms the basis of the *vertex-centric* paradigm, which is implemented in frameworks such as Galois [14], and GraphIt [15].

Algorithm 2 Pushing flow with frontier based on CSR

```
1: y[:]=0
2: for i in frontier do parallel
3:     for j=rowPtr[i] to rowPtr[i+1]-1 do
4:         atomAdd(y[colNdx[j]],val[j]·x[i])
```

Additionally, graph processing can be reformulated as a SpMV problem: $\mathbf{y} = \mathbf{A}^T\mathbf{x}$, where the rows and columns of $\mathbf{A}$ correspond to the source and target vertices, and $\mathbf{x}$ and $\mathbf{y}$ are input and output vectors of size n, and the superscript ^{T} denotes the matrix transpose. This formulation highlights a strong compatibility between graph processing and matrix computation. Notably, several graph processing frameworks are directly built on top of sparse matrix operations, expressing graph algorithms in linear algebraic forms [24]–[27].

B. Graph Reordering

Graph reordering algorithms rearrange the order of computations by relabeling vertex IDs. It functions as a preprocessing step to boost downstream graph tasks. For practicality, the reordering time should be sufficiently short to justify its overall effectiveness. Reordering algorithms can be categorized into two types: *degree-based* and *structure-based*.

Degree-based algorithms rely solely on the values of vertex degrees to redistribute vertices into different segments. For instance, Sorting relabels vertices in the descending order of their degrees. Hub Clustering (HC) separates the hot vertices and cold ones into two groups without sorting, which aims to preserve the original graph structure [8]. Frequency-Based Clustering (HBC) also classifies vertices into hot and cold, and only sorts the hot ones [20]. Degree-Based Grouping (DBG) allocates vertices into multiple groups of distinctive degree ranges and does not sort them within any group [9]. In common, these algorithms concentrate high-degree vertices and assign consecutive IDs to them.

Structure-based reordering conducts in-depth investigation into the internal structure of graphs. Rabbit Order explores

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

the hierarchical communities in a graph [7]. Gorder extensively calculates the locality of vertex pairs [6]. Due to the sophisticated analysis of structural knowledge, structure-based algorithms cost more time than degree-based ones, as listed in Fig. 1. The preprocessing cost of degree-based algorithms can be amortized by $5-7$ PageRank iterations. By contrast, Rabbit and Gorder cost over 58 and even 16K iterations respectively, which are considered impractical.

C. Matrix Blocking

We refer to *matrix blocking* as an encapsulation of matrix optimizations built on Cache Blocking. Cache Blocking is a well-established optimization technique for SpMV [28], which divides a matrix into cache-sized submatrices (i.e., blocks) to ensure that data is efficiently accessed within caches. Binning further enhances memory efficiency by transforming random memory accesses into sequential ones within each block using sorted buffers (i.e., bins) [16], [17]. Leveraging the duality between graphs and matrices, matrix blocking motivates the *partition-centric* paradigm in graph processing, where computation on cache-sized partitions (i.e., subgraphs) improves locality and reduces communication. This paradigm is exemplified by frameworks such as GPOP [18] and CoroGraph [21].

Algorithm 3 Blocking execution model

```
1: for i=0 to b-1 do parallel
2:     for j=0 to b-1 do
3:         Scatter(block[i,j], bins[i,j])
4: for i=0 to b-1 do parallel
5:     for j=0 to b-1 do
6:         Gather(block[j,i], bins[j,i])
```

The blocking technique introduces a two-phase execution model as in Algo. 3: a Scatter phase where updates are written to bins, and a Gather phase where blocks accumulate results. Although it regularizes memory access, the blocking SpMV doubles the matrix scans and can increase instruction overhead, potentially degrading performance on graphs that already possess high locality.

III. Motivation

Although graph reordering and matrix blocking both aim to enhance locality, a simple combination is often counterproductive. To demonstrate this, we combined various reordering algorithms with a state-of-the-art blocking PageRank and measured performance against an unordered blocking baseline.

Fig. 1 shows that reordering preprocessing cannot promise consistent performance boosts for the blocking PageRank. For instance, algorithm Gorder accelerates the processing on graphs *twitter*, *pld* and *host*, but at the same time, decelerates the rest of graphs. In Fig. 1, the majority of results (in dark boxes) are < 1.00, which means slower speed compared to the unordered baseline. HC functions as the best reordering algorithm, since it only brings slowdown to *orkut* and achieves an average speedup of $1.17\times$ over all graphs.

The slowdown of reordering algorithms originates from two factors. First, some open-sourced graph datasets are already reordered [29]. Another round of reordering might disrupt the well-optimized structure of graphs, thus leading to a slowdown. The HC addresses this issue by classifying vertices into two groups without sorting them, thereby preserving the relative order of vertices and minimizing disruptions to original locality.

	live	orkut	track	wiki	twitter	pld	mpi	kron	host	Aver.	Cost
Sort+B	2.26	0.36	0.91	0.83	0.46	0.65	0.76	0.39	0.73	0.87	5.74
HC+B	1.11	0.72	1.06	1.00	1.00	1.07	1.04	1.46	1.48	1.17	6.06
FBC+B	1.64	0.33	0.88	0.74	0.63	0.73	0.78	0.39	0.73	0.81	6.03
DBG+B	1.84	0.84	1.19	0.84	0.69	0.90	0.91	0.83	1.06	1.01	4.61
Rabbit+B	2.76	0.87	2.01	1.21	0.76	0.15	1.12	0.72	1.06	1.11	58.04
Gorder+B	0.91	0.40	0.90	0.80	1.13	1.20	0.90	0.70	2.54	1.17	16k
Blocking	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	--
Pulling	1.24	0.97	0.40	1.20	0.96	0.63	0.92	0.49	0.75	0.88	--

Fig. 1: Speedup of blocking PageRank with various graph reordering algorithms, normalized by the unordered blocking baseline. Lighter colors indicate better results. The bottom row shows pulling PageRank for comparison. The last two columns display average results and reordering cost (k: thousands).

Second, the concentration of high-degree vertices causes workload imbalance among blocks. High-degree vertices typically require more computational effort, leading to heavier workloads in their corresponding blocks. The imbalance delays the thread with the heaviest workload in parallel regions, ultimately degrading overall graph processing performance [10].

Furthermore, blocking PageRank generally outperforms the pulling variant (as shown at the bottom of Fig. 1), although its advantage is not universal. On average, it achieves a $1.14\times$ speedup over the pulling approach. However, graphs such as *live* and *wiki* already have strong spatial locality and offer limited potential for further improvement. In these cases, blocking yields suboptimal performance, as the additional instruction overhead outweighs its benefits.

The limitations of uncooperative graph-matrix technique raise the need for a better solution. This motivates us to create a cohesive integration of graph reordering and matrix blocking that identifies and separately handles regions with different locality characteristics.

IV. Design of Rebo

In this section, we propose *Rebo*, a novel optimization system that accelerates graph processing. It provides a synergistic collaboration between graph reordering and matrix blocking. We firstly introduce the overview of Rebo, and then offer details in the aspects of graph reordering, blocking and execution model in following sections.

Overview. Rebo synergistically integrates graph reordering and matrix blocking. It first uses reordering to clas-

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

sify vertices as hot, cold, and frozen, which extracts dense, sparse, and void submatrices from the adjacency matrix (Section IV-A). The dense submatrix is partitioned into cache-friendly blocks to improve temporal locality and load balance (Section IV-B), while the sparse submatrix is flattened into a cache-line-aligned array to minimize memory indirection (Section IV-C). A selective execution model then processes the dense and sparse regions with partition-centric and vertex-centric paradigms, respectively (Section IV-D).

A. Submatrix Extraction by Graph Reordering

The extraction of dense, sparse and void submatrices is based on the classification of graph vertices through graph reordering. Vertices are grouped into three categories:

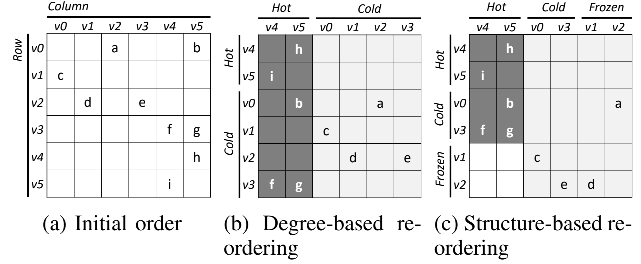
- **Hot vertices:** with in-degree larger than the average degree of the graph.
- **Cold vertices:** with in-degree no larger than the average degree, and comprising at least one outgoing edge to the hot vertices.
- **Frozen vertices:** with in-degree no larger than the average degree, but holding *no* outgoing edges to the hot vertices.

Rebo combines *degree-based* and *structure-based* reordering methods for vertex classification. It first applies a degree-based approach that compares each vertex's in-degree against the graph's average degree to identify hot vertices (i.e., highly populated columns), the threshold of which is later validated in Section V-G. These hot vertices are then relabeled with consecutive IDs starting from index 0, while preserving their relative order to maintain locality. All remaining vertices are temporarily marked as cold.

Rebo performs reordering based on *in-degree* rather than out-degree. Many power-law graphs exhibit asymmetry between in-degree and out-degree distributions, where the highest in-degree is significantly greater than the highest out-degree. This asymmetry reflects real-world phenomena commonly observed in web and social graphs. For example, in the Wikipedia hyperlink network (e.g., *wiki* in Table III), many pages link to the homepage, resulting in a massive number of incoming edges. However, it is rare for a single wiki page to reference millions of other pages as outgoing edges.

As shown in Table III, the number of hot vertices based on in-degree is typically lower than when using out-degree. Since hot vertices correspond to the columns of the dense submatrix, fewer hot vertices result in fewer columns and a more compact layout. Therefore, in-degree is chosen as the basis for degree-based reordering to achieve a more compact dense submatrix.

Fig. 2 illustrates an example of Rebo's reordering procedure. Fig. 2a shows the adjacency matrix of a graph with an average degree of $m/n = 9/6 = 1.5$. The number of non-zeros in a column represents the in-degree of the corresponding vertex. Based on this, v_4 and v_5 are identified as hot vertices, with in-degrees of 2 and 3, respectively. Using degree-based reordering, these hot vertices are shifted to the beginning of the vertex array, as shown in Fig. 2b. The remaining vertices are initially tagged as cold.



(a) Initial order (b) Degree-based reordering (c) Structure-based reordering

Fig. 2: The reordering procedure of Rebo. Firstly, in (b), hot vertices and cold ones are separated via degree-based reordering. Next, in (c) frozen vertices are identified from cold vertices via structure-based reordering. In (b) and (c), the dense, sparse and void submatrices are colored in dark grey, light grey, and white respectively.

Next, the structure-based reordering examines the incoming edges of the hot vertices. If a cold vertex has no incoming edges from any hot vertex, it is re-classified as *frozen*. Identifying these vertices is crucial as they form a void submatrix since they have no interactions with hot vertices, allowing their exclusion from certain computations and reducing the effective size of the dense regions to be processed. Frozen vertices are assigned IDs that come after the cold vertices. The relative orders of vertices within both the cold and frozen groups remain unchanged, minimizing disruption to the graph's original order.

Continuing with the example in Fig. 2b, it can be observed that the cold vertices v_1 and v_2 have no outgoing edges to the hot vertices v_4 and v_5. As a result, they are reclassified as frozen vertices. Subsequently, as shown in Fig. 2c, the structure-based reordering relocates these frozen vertices. Interactions from frozen vertices to hot vertex columns form a void submatrix with no meaningful computation. Excluding the void region effectively reduces the size of the dense submatrix, improving spatial locality and avoiding unnecessary data loading.

The categorization of vertex types naturally results in the formation of dense, sparse, and void submatrices, which are highlighted with distinct colors in Fig. 2c. Let the numbers of hot, cold, and frozen vertices be denoted as n_h, n_c, and n_f, respectively, such that $n_h + n_c + n_f = n$. The regions of the submatrices are defined as follows:

- **Dense submatrix:** Rows correspond to hot and cold vertices (as sources), and columns correspond to hot vertices (as targets). Its size is $(n_h + n_c) \times n_h$.
- **Sparse submatrix:** Rows include all vertices (as sources), and columns represent cold and frozen vertices (as targets). Its size is $n \times (n_c + n_f)$.
- **Void submatrix:** Rows represent frozen vertices (as sources), and columns correspond to hot vertices (as targets). Since frozen vertices do not connect to hot vertices, this submatrix contains only zeros. Its size is $n_f \times n_h$.

The dense submatrix contains the majority of non-zero

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

elements from the original adjacency matrix $\mathbf{A}$, especially in power-law graphs. For example, in the *wiki* graph, the dense submatrix accounts for 87% of the non-zeros while covering only 11% of the columns. The remaining 89% of the columns form the sparse submatrix, which is significantly sparser than the original matrix. Additionally, 51% of the rows within the dense submatrix for *wiki* are empty. These empty rows are moved to the bottom, reducing the size of the dense submatrix. The empty rows form a void submatrix that is excluded from computation. These submatrices collectively represent the structure of the adjacency matrix for the reordered graph, partitioned to optimize spatial and temporal locality for subsequent processing.

B. Blocking Dense Submatrix

After constructing the submatrices, Rebo further divides the dense submatrix into multiple 2-D blocks that fit within the cache for reuse to maximize temporal locality. Unlike traditional blocking techniques, which use uniform block sizes [16], [17], Rebo introduces an elastic blocking technique to adjust block sizes for better workload balance. The block refinement process consists two steps:

(1) Initial 2-D blocking with equal sizes: Initially, Rebo adopts the conventional Cache Blocking approach, partitioning the adjacency matrix into square blocks of equal size. Each block has br rows and bc columns, both equal to the cache size c, i.e., $br = bc = c$. These blocks serve as the fundamental data units for thread computation. Conceptually, the dense submatrix of size $(n_h + n_c) \times n_h$ is divided into $(n_h + n_c)/c \times n_h/c$ blocks. The optimal value of c will be explored in Section V-G.

An example is shown on the left side of Fig. 3. Suppose the dense submatrix has a size of 8×6 after Rebo's reordering, and the cache can accommodate 2 vertices. During the initial partitioning, this submatrix is divided into 4×3 blocks, each of size 2×2.

(2) Column-wise refinement: In Step (1), all blocks are square matrices of equal size. However, due to the formation of the dense submatrix, the most crowded columns (i.e., vertices with the largest in-degrees) tend to cluster vertically. As a result, some block-columns (groups of blocks in the same column) are densely packed with non-zero elements, leading to workload imbalances. For instance, in the left submatrix in Fig. 3, block-column 2 (denoted as B-Col. 2) contains 16 non-zeros, while B-Col. 0 and B-Col. 1 contain only 2 and 3 non-zeros, respectively. This imbalance causes highly disproportionate computational loads across block-columns.

To address this imbalance, Rebo refines the block-columns by further subdividing over-populated ones into "narrower" block-columns. A block-column is considered over-populated if the number of non-zeros in the jth block-column exceeds twice the average number of non-zeros across all block-columns. When this condition is met, the over-populated block-column is iteratively split vertically into two narrower parts until the condition is no longer satisfied.

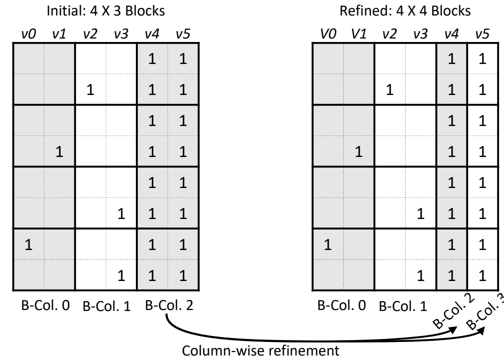


Fig. 3: Column-wise refinement for load balancing. Left: initial blocks. Right: over-populated B-Col.2 is split into narrower B-Col.2 & B-Col.3. (Interleaved colors for readability).

Following Fig. 3, B-Col. 2 initially contains 16 non-zeros, and the average number of non-zeros over three block-columns is 7. Since $16 > 2 \cdot 7$, B-Col. 2 is subdivided into two sub-columns. For the refined submatrix, a new B-Col. 3 is created. Also, the sizes of blocks in B-Col. 2 and 3 become 2×1, which are no more square. The physical layout of the dense submatrix is not modified. Hence, the refinement of block-columns incurs negligible overheads.

For parallelism, the number of block-columns derived from the dense submatrix must be no smaller than the number of available threads. When this requirement is not satisfied, the rows and columns of blocks will be halved repeatedly (so a 4×4 block is split into four 2×2 blocks).

C. Flattening Sparse Submatrix

Traditional CSR/CSC-based vertex-centric paradigms suffer from indirect memory access: retrieving a vertex's neighbors requires first accessing row/column pointers and then looking up column/row indices, which increases memory traffic and causes frequent cache misses.

To address the limitation of vertex-centric paradigms, Rebo flattens the representation and processing of sparse submatrices. It packs column pointers and row indices into a single, cache-line-aligned array. In the CSC case, the row indices of a vertex are stored directly alongside its column pointer. Multiple low-degree vertices are batched into the same cache line, maximizing spatial locality and minimizing memory indirection. If a vertex's neighbor list exceeds one cache line, the remaining entries spill into the next line. To distinguish between pointers and indices, Rebo sets the most significant bit of the column pointer to 1.

The flattened representation is particularly effective for sparse submatrices because most vertices have only a few neighbors and therefore fit entirely within a single cache line (typically 64 bytes). It eliminates indirection between separate arrays and flattens the loop structure: where the conventional nested loop over vertices and neighbors becomes a single linear scan. This flattening improves spatial locality and multithreading efficiency.

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

D. Selective Execution Model

Rebo employs a hybrid execution strategy that adapts to the structural properties of different matrix regions. Dense submatrices are processed using a partition-centric paradigm at the block level, which maximizes data reuse and cache efficiency. In contrast, sparse submatrices, where the data locality is inherently limited, are handled using a vertex-centric paradigm, avoiding the overhead of blocking and thereby reducing instruction counts.

Rebo employs different thread scheduling policies across its execution paradigms based on OpenMP library [30]. The partition-centric paradigm uses *dynamic* scheduling with a chunk size of 1 to effectively balance the varying workloads across dense blocks. Conversely, the vertex-centric paradigm employs *static* scheduling with each thread processing a similar number of cache lines. Further, for dynamic graph algorithms such as BFS, Rebo adopts the hierarchical frontier from GPOP [31] to track active blocks and vertices within each block, thereby skipping inactive elements in the graph.

Algo. 4 outlines the processing model of Rebo. For each block, three bins are created: `inter`, `intra`, and `data`, representing *inter-edges* from source vertices to target blocks, *intra-edges* within the same block, and vertex *updates*, respectively. The allocated bins enable sequential reads and writes, avoiding atomic operations and synchronization. Further, `inter` reduces memory traffic by aggregating messages across blocks into a single transmission, while `intra` enables efficient local broadcasting of messages within a block. The sparse matrix is flattened into a cache-line-aligned array, `flatCSC`, improving memory access efficiency.

Algorithm 4 Graph processing with Rebo

```
Variables: blockRow, blockCol → #block-row and #block-col
    colRng[] → vertex range of every block-column
    inter[], intra[], data[] → bins
    flatCSC → flattened CSC representation of graph
 1: /* Dense Submatrix under Partition-Centric Paradigm */
 2: for i=0 to blockCol-1 do parallel
 3:     for j=0 to blockRow-1 do
 4:         Scatter(inter[i][j],data[i][j])
 5: for i=0 to blockRow-1 do parallel
 6:     for v=colRng[i].first to colRng[i].last do
 7:         Reset(v)
 8:     for j=0 to blockCol-1 do
 9:         Gather(intra[j][i], data[j][i])
10:     for v=colRng[i].first to colRng[i].last do
11:         Apply(v)
12: /* Sparse Submatrix under Vertex-Centric Paradigm */
13: for t to numThreads do parallel
14:     cacheLines = flatCSC[t]
15:     Pull(cacheLines)
```

The execution on the dense submatrix follows the Scatter-Reset-Gather-Apply (SRGA) phases. In the first parallel region (lines 2–4), updates from source vertices (hot and cold vertices) are pushed to the buffer spaces of target blocks via inter-edges using the `Scatter` interface. In the second parallel region (lines 5–11), target vertices (hot vertices) are reset via `Reset`, and buffered data are aggregated to the target vertices through intra-edges using `Gather`. Finally, the `Apply` interface updates the aggregated results.

The third parallel region processes the sparse submatrix (lines 13–15) using a vertex-centric paradigm, parallelized at the granularity of cache lines. To support the pull-based data flow, Rebo introduces `Pull`, where each vertex gathers updates from its incoming neighbors. This mechanism leverages the most significant bit to distinguish pointers from indices in the flattened representation, enabling efficient traversal without costly indirect accesses across separate arrays. Within the execution model, the Scatter, Gather, and Pull phases dominate the computational cost, as they involve edge traversals and message propagation. In contrast, the Reset and Apply phases are lightweight, operating only on local vertex data.

V. Evaluation

Platform. The performance evaluations are conducted on a multicore system with two 10-core Intel Xeon Silver 4210 processors @ 2.2GHz with 64KB L1 cache, 1MB L2 cache, 27.5MB LLC and 256GB DRAM. The index data type is `unsigned int`, while the value data type is either `float` or `double`, depending on the comparison with other works. The code of Rebo is compiled by `g++-13` with `-O3` and `-std=c++23`. The parallelization is enabled by `OpenMP` library and utilizes all cores without hyper-threading. Hardware activities are profiled by `Perf` and `Likwid` [32].

Baselines. Rebo is compared against high-performance graph frameworks, including Ligra [11], GraphIt [15], GPOP [31], and CoroGraph [21]. Ligra parallelizes graph processing at the vertex level. GraphIt advances Ligra with various auto-tuning optimizations. GPOP and CoroGraph are partition-centric frameworks that integrate vertex-centric paradigms for dynamic algorithms: GPOP switches paradigms based on active vertices in each partition, and CoroGraph uses a frontier queue adapted from Galois [14]. Rebo adopts GPOP's hierarchical frontier but applies the partition-centric paradigm exclusively to dense submatrices.

Rebo is further evaluated against four performant SpMV methods using `double` precision. Intel MKL defaults to CSR format [42], while CSR5 enhances CSR with segmented sum algorithms and better SIMD utilization [43]. SELL-C-σ optimizes SIMD via column-wise blocking. LAV partitions the matrix into dense and sparse regions, applying SIMD to the dense part [44]. Due to the unavailability of LAV's source code, we implemented it based on the original paper.

Dataset. We examine Rebo on 11 large-scale graph datasets as listed in Table III. Graphs *live* [33] *orkut* [33], *twitter* [37] and *mpi* [38] derive from online social networks. Graphs *wiki* [35], *track* [34], *pld* [36], *host* [40] are crawled from hyperlinks of websites. Graph *kron* [39] is a synthetic graph following power-law distribution; graph *urand* [39] is also generated but with uniform degree distribution. Graph *road* [41] is a mesh-structured road graph characterized by low degree and balanced distribution. Rebo is designed for power-law

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Performance in second across graph systems and SpMV methods

	PageRank					BFS				
	Rebo	GPOP	CoroGraph	GraphIt	Ligra	Rebo	GPOP	CoroGraph	GraphIt	Ligra
live	**0.029**	0.126	0.521	0.309	2.085	**0.220**	0.239	0.703	0.660	0.438
orkut	**0.037**	0.092	1.236	0.537	4.672	0.299	0.302	1.33	1.837	**0.013**
track	**0.134**	0.165	1.499	0.551	3.025	**0.043**	0.052	0.152	0.288	0.913
wiki	**0.047**	0.077	0.766	0.341	2.305	**0.145**	0.147	0.771	0.467	0.694
pld	**0.203**	0.208	2.336	0.252	1.937	**0.432**	0.809	1.053	1.577	1.010
twitter	**0.290**	0.449	4.141	1.346	9.456	**0.655**	0.945	2.043	5.816	0.961
mpi	**0.452**	0.804	6.457	5.66	42.200	1.063	1.035	6.267	9.450	**1.010**
kron	**0.377**	0.586	8.068	0.882	7.850	**0.729**	1.15	3.13	4.920	0.775
host	**0.590**	0.821	21.317	0.948	7.653	**1.970**	3.991	4.002	7.550	3.700
roadusa	0.122	0.082	0.178	**0.013**	0.081	0.682	1.254	**0.331**	0.644	50.100
urand	0.023	**0.017**	1.293	0.112	0.860	**0.133**	0.298	0.554	1.091	0.391
speedup	1.00×	1.67×	21.28×	5.81×	43.37×	1.00×	1.48×	3.54×	5.47×	10.20×

	Connected Components					SpMV				
	Rebo	GPOP	CoroGraph	GraphIt	Ligra	Rebo	MKL	CSR5	SELL-C-σ	LAV
live	0.781	**0.733**	0.739	1.586	2.751	**0.029**	0.077	0.042	0.046	0.089
orkut	**0.447**	1.464	1.731	3.626	5.523	**0.073**	0.476	0.125	0.169	0.271
track	**0.383**	0.553	2.868	1.506	2.835	0.054	0.065	**0.047**	0.078	0.157
wiki	**0.244**	0.421	1.886	2.501	4.539	**0.048**	0.237	0.102	0.221	0.317
pld	**0.558**	0.584	2.144	0.871	4.032	**0.229**	0.629	0.621	0.967	1.338
twitter	**0.634**	1.349	4.318	4.242	13.310	**0.626**	3.446	1.011	1.536	2.691
mpi	**0.632**	1.639	2.085	14.728	43.351	**0.562**	5.553	1.252	1.534	3.159
kron	**0.642**	1.186	5.799	3.738	14.694	**0.538**	2.196	2.099	2.655	2.060
host	**2.457**	2.247	8.643	4.935	17.748	**0.681**	1.658	1.595	2.088	1.565
road	15.917	19.293	15.065	22.885	216.472	0.029	0.022	**0.013**	0.024	0.047
urand	0.634	**0.270**	1.201	0.599	1.387	**0.149**	0.273	0.277	0.308	0.764
speedup	1.00×	1.60×	4.49×	6.01×	11.57×	1.00×	3.87×	1.94×	2.76×	4.11×

TABLE III: Graph attributes: Vertices (V), edges (E), max in/out-degrees (D_{in}, D_{out}), and counts of hot vertices by in/out-degrees (H_{in}, H_{out}).

	Graph	V	E	D_{in}	D_{out}	H_{in}	H_{out}
power-law	*live* [33]	7.5M	112.3M	1.1M	300	325.2K	1.7M
	orkut [33]	8.8M	327.0M	31.8K	4.0k	688.4K	1.2M
	track [34]	12.8M	140.6M	11.6M	1.1M	1.3M	4.9M
	wiki [35]	18.3M	172.1M	631.4K	9300	1.9M	3.4M
	pld [36]	42.9M	623.1M	1.8M	3.9M	6.3M	5.9M
	twitter [37]	41.7M	1.5B	770.1K	3.0M	4.7M	3.7M
	mpi [38]	52.6M	2.0B	3.5M	780.0K	5M	5.9M
	kron [39]	67.1M	2.1B	1.0M	1.0M	5.7M	5.7M
	host [40]	101.7M	2.0B	2.3M	1.3M	10.4M	12.9M
non-*	*road* [41]	23.9M	57.7M	9	9	11.9M	11.9M
	urand [39]	8.4M	268.4M	64	64	4.4M	4.4M

graphs. Non-power-law graphs *road* and *urand* are included to illustrate its limitations and effective scope.

A. Execution Time

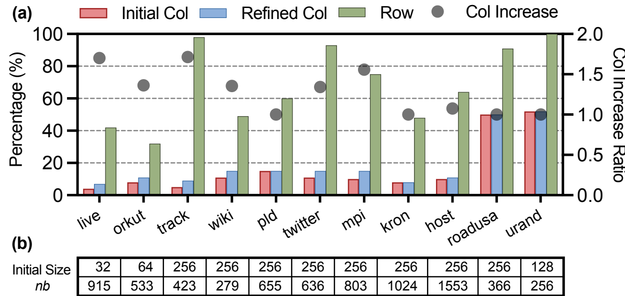
Table II displays the execution times for PageRank, BFS, Connected Components (CC) and SpMV across various systems and datasets, with both PageRank and SpMV times reported on a per-iteration basis. PageRank and SpMV are static algorithms, whereas BFS and CC are dynamic with the set of active vertices changing across iterations. Rebo demonstrates significant performance gains, achieving speedups of $1.67\times$ for PageRank, $1.48\times$ for BFS, $1.60\times$ for CC and $1.94\times$ for SpMV compared to the next best-performing approaches.

For PageRank, GPOP employs a blocking strategy for the entire graph, achieving the second-best performance with optimized cache efficiency. CoroGraph, though partition-centric, is the slowest due to the overhead of frontier queue management (e.g., frequent memory allocation/deallocation) in scatter-gather phases. For static algorithms like PageRank, both Ligra and GraphIt use the pulling flow for execution, as described in Algo. 1. GraphIt outperforms Ligra by removing unnecessary frontiers in static algorithms.

For dynamic BFS and CC, the vertex-centric frameworks Ligra and GraphIt deploy a hybrid push–pull paradigm, which incurs substantial write conflict overhead in the pushing flow. Since BFS involves relatively lightweight computation, it sometimes performs best with Ligra's simple design and minimal overhead. On the other hand, the partition-centric frameworks effectively avoid write conflicts by employing buffering bins. Overall, partition-centric systems outperform their vertex-centric counterparts, highlighting the advantages of cache-aware execution for graph workloads.

For SpMV, Rebo consistently delivers faster execution across nearly all power-law graphs. CSR5 remains competitive despite being introduced several years ago. SELL-C-σ is optimized for HPC matrices with regular structure (e.g., *road*), and therefore shows suboptimal performance on the irregular graphs evaluated here. LAV performs poorly, possibly because our reimplementation may miss certain undisclosed optimizations in the original system.

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.



Initial Size	32	64	256	256	256	256	256	256	256	256	128
nb	915	533	423	279	655	636	803	1024	1553	366	256

Fig. 4: **(a)** Dense submatrix dimensions: block-row ratio and both initial/refined block-column ratios relative to total blocks (nb); **(b)** initial block size (KB) and total blocks (nb).

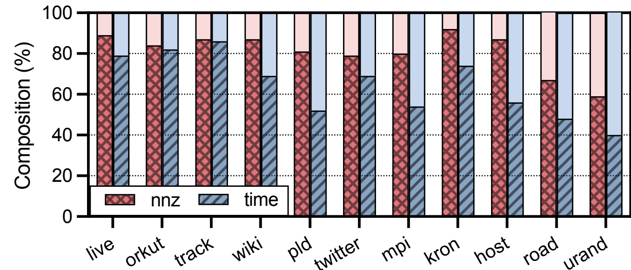


Fig. 5: Nonzero and execution time composition for dense (shadowed) and sparse (empty) submatrices.

B. Decomposition of Matrix

Fig. 4 presents Rebo's matrix decomposition. Fig. 4a shows the dimensions of the dense submatrix, expressed as percentages of block-rows and block-columns relative to nb. Here, nb denotes the total number of blocks if $\mathbf{A}$ were fully partitioned into initial-sized blocks, as detailed in Fig. 4b, which also lists the initial block size in KB. For example, the graph *wiki* has $nb = 279$, and its dense submatrix has a block-row ratio of 47% (Fig. 4a). This indicates that the dense region spans approximately $279 \times 0.47 \approx 131$ block-rows.

Power-law graphs. For power-law graphs, dense submatrices are "tall and narrow," with a high ratio of block-rows (over 32%) and a low ratio of block-columns (below 15%). Their area is small, covering only 2% to 11% of $\mathbf{A}$. This concentration of critical edges into a compact region is advantageous, as it allows Rebo to focus its locality optimizations where they yield the most significant performance gains.

Moreover, in the majority of power-law graphs, the block-columns of dense submatrices are further subdivided by column-wise refinement. The number of block-columns increases by up to $1.71\times$ ($1.35\times$ on average), as indicated by the dots in Fig. 4a. A larger increase reflects greater imbalance in the original graph, highlighting the effectiveness of the column-wise refinement.

Non-power-law graphs: In contrast, *road* and *urand* graphs display different dimensional patterns. The block-rows cover over 90% of nb, while the block-columns account for around 50% and column-wise refinement does not create any additional block-columns. Consequently, the dense submatrices occupy nearly 50% of $\mathbf{A}$'s total area. This larger, less distinct dense region suggests that Rebo's selective processing strategy could be less effective for non-power-law graphs.

C. Contributions of Dense and Sparse Submatrices

To evaluate the contributions of dense and sparse submatrices to overall performance, Rebo's performance for PageRank is further analyzed. BFS and Connected Components are excluded from this analysis due to their performance variability, which depends on the initial vertex and/or varies across iterations. Fig. 5 presents the distribution of non-zero elements and the execution time for dense and sparse submatrices.

Power-law graphs. The dense submatrices pose a dominant impact on performance, accounting for a significant majority of non-zeros and execution time, averaging 85% and 69%, respectively. The proportion of non-zero elements in the dense submatrices consistently exceeds their share of execution time, highlighting the effective acceleration achieved for dense submatrices. This efficiency arises from Rebo's targeted use of fine-grained blocking in dense regions, which enhances spatial-temporal locality and cache utilization.

Non-power-law graphs. In contrast, for non-power-law graphs, dense submatrices contain 63% of the non-zero elements but contribute only 44% of the execution time, a relatively small fraction. Sparse submatrices, primarily composed of low-degree vertices, dominate processing time due to their inherently poor data locality. Furthermore, Rebo's graph reordering, while optimizing dense regions, can sometimes inadvertently worsen the spatial locality of these sparse submatrices, explaining why a whole-graph blocking approach (e.g., GPOP) might be more beneficial here.

D. Memory & Cache Efficiency

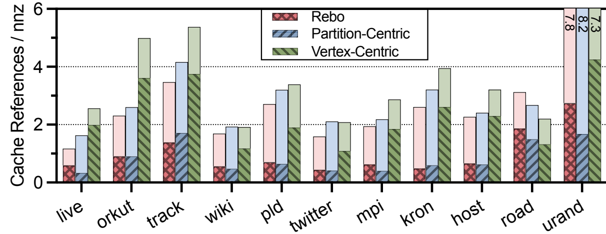
To assess Rebo's effectiveness in exploiting the spatial and temporal locality in graphs, we analyze its memory and cache efficiency. We compare Rebo with two baseline variants that use only the partition-centric (PC) or vertex-centric (VC) paradigm to execute PageRank. All results are normalized by the number of edges across graphs.

Power-law graphs. On power-law graphs, Rebo consistently produces the fewest cache references (hit + miss), as shown in Fig. 6a. It reduces cache references by 20% and 56% compared to the VC and PC variants, respectively. The improved cache efficiency validates Rebo's effectiveness in enhancing spatial and temporal locality.

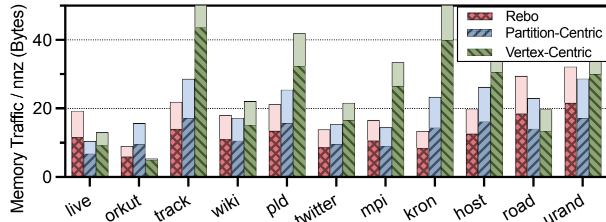
The VC variant suffers from a high cache miss ratio, averaging 66%, due to the poor data locality typical of graph processing. In contrast, the PC variant achieves the lowest cache miss ratio of 25% by regularizing data accesses. Rebo achieves a balance with a slightly higher miss ratio of 33% with its selective execution strategy applied to the dense and sparse submatrices, each processed with a paradigm suited to its locality characteristics.

Regarding cache hits, the approaches rank as VC $<$ Rebo $<$ PC, consistent with the trend observed in instruction counts

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.



(a) Cache references. Upper empty bars: cache hits. Lower shadowed bar: cache misses.



(b) Memory traffic. Upper empty bars: writes. Lower shadowed bars: reads.

Fig. 6: Memory traffic and L2 cache references across Rebo and its variants, normalized by the number of edges.

(see Section V-E). By reducing both instruction count and total cache references, Rebo compensates for its slightly higher miss ratio, resulting in improved overall performance.

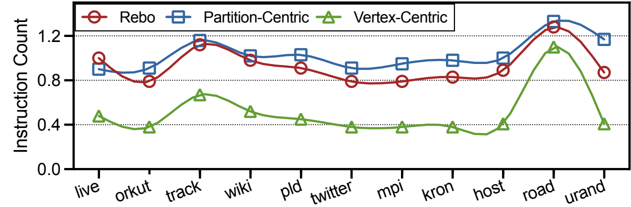
In addition, Rebo typically achieves the lowest memory traffic (reads + writes), as shown in Fig. 6b. Compared to the PC and VC variants, it reduces memory traffic by 20% and 80%, respectively, mainly due to better cache reuse, a direct consequence of enhanced spatial and temporal locality from its reordering and blocking strategy. However, for small (e.g., *live*) or imbalanced (e.g., *mpi*) graphs, small block sizes from column refinement can increase memory traffic, causing Rebo to generate more traffic than the PC variant.

Non-power-law graphs. On the *road* graph, Rebo shows the highest cache references and memory traffic compared to VC and PC because its reordering disrupts the graph's already high spatial locality. On the *urand* graph, which lacks spatial locality, performance is best when the entire graph is processed as cache-sized blocks by PC. These observations explain why Rebo is outperformed by GPOP on these two graphs.

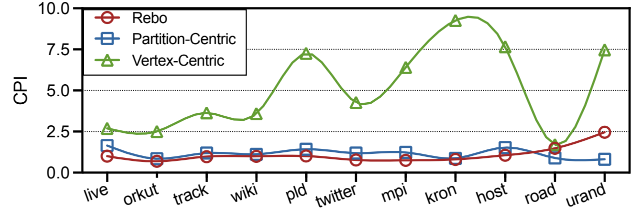
E. Instruction Utility

We further analyze Rebo's performance by profiling its instruction count and cycles per instruction (CPI), as shown in Fig. 7. Rebo achieves the lowest CPI on most graphs, with an instruction count between those of PC and VC. Its instruction behavior closely resembles that of PC, indicating that the highly optimized computation on the dense submatrix, which is the focus of Rebo's optimization, is a key factor in its overall performance, as discussed in Section V-C.

Power-law graphs. The VC variant attains the lowest instruction count due to its simple code structure but suffers



(a) Normalized instruction count



(b) Cycles per instruction

Fig. 7: The instruction count and CPI of Rebo and its variants.

from the highest CPI caused by irregular memory access. The PC variant reduces CPI by 57% but increases instructions by 104%, illustrating a trade-off. Rebo achieves the best trade-off, reducing CPI by 64% while increasing instructions by 86%, leading to the best overall performance.

Non-power-law graphs. Graphs *road* and *urand* exhibit contrasting behaviors. *road* has inherently high spatial locality, allowing the VC variant to achieve low CPI and leaving limited opportunity for further improvement by PC or Rebo. As a result, the increased instruction count in PC and Rebo leads to degraded overall performance. In contrast, *urand* suffers from poor spatial locality due to its random vertex ordering. It benefits significantly from the whole-graph blocking strategy in the PC variant, which achieves the lowest CPI.

F. Preprocessing Overhead

The basic pulling flow in Algo. 1 directly operates on the CSC representation of graphs and serves as the baseline. Blocking approaches require conversion time to transform graphs into blocked representations. Table IV shows the number of iterations needed for the pulling flow to amortize the preprocessing overhead of various methods. The SpMV CSR5 incurs the lowest overhead. The preprocessing costs for GPOP and Rebo are moderate, requiring approximately 10 iterations to amortize. CoroGraph incurs higher overheads due to its queue-based frontier. The preprocessing cost of GraphIt includes C++ code generation, which is difficult to measure and, therefore, not reported.

Rebo's preprocessing consists of two stages: graph reordering and submatrix blocking. As listed in Table IV, the reordering stage takes longer than the blocking stage because reordering modifies the entire graph, whereas blocking only manipulates a subset (i.e., the dense submatrix).

G. Sensitivity Analysis

Block Size. Rebo organizes the dense submatrix into 2D blocks via a two-step process: selecting an initial block size

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: The preprocessing overheads of various methods amortized by the number of the pulling flow's iterations

Graph	CSR5	GPOP	CoroGraph	Rebo		
				Reorder	Block	Total
live	2.0	21.0	4.1	14	4.1	18.1
orkut	4.2	28.6	9.3	10.7	3	13.7
track	0.5	9.0	3.19	4.2	0.6	4.9
wiki	2.9	33.2	7.7	11.8	3.3	15.1
pld	1.8	19.8	11.5	8.6	2.1	10.7
twitter	2.9	27.8	14.4	10.5	6	16.5
mpi	2.0	19.0	9.1	7.9	3.6	11.5
kron	2.2	15.9	16.3	6.4	2.1	8.5
host	2.1	21.3	24.8	6.6	2.4	9.0
Average	2.3	9.2	21.7	8.8	2.9	11.7

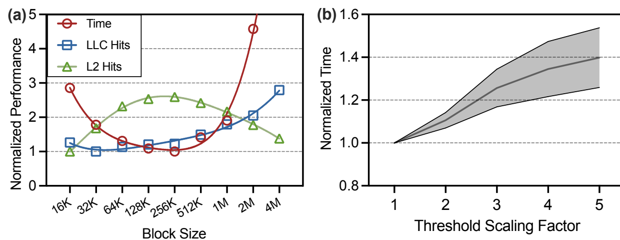


Fig. 8: **(a)** Normalized execution time, L2/LLC hits on *wiki*. **(b)** Execution time with varying degree threshold.

and adjusting block-column widths to balance the workload. Our focus is on identifying the optimal initial block size. We evaluated sizes from 16 KB to 4 MB, spanning L1, L2, and LLC capacities. For the *wiki* graph (Fig. 8a), 256 KB achieves the shortest execution time and highest L2 hit rate, balancing cache efficiency and performance. Larger blocks spill over into LLC and memory, reducing L2 hits, while smaller blocks cause frequent cache evictions and lower data reuse.

The initial block size is also constrained by the graph size and the number of threads. For smaller graphs (e.g., *live*, *orkut*, and *urand*), a 256 KB block may generate fewer block-columns than available threads, limiting parallelism. To ensure sufficient parallelism, Rebo iteratively halves the block size until all threads are utilized. However, this creates a performance trade-off, as smaller blocks increase the instruction count and reduce cache efficiency.

Degree Threshold. Rebo classifies a vertex v as hot if $\deg(v) \geq \alpha \times \overline{\deg}$, where $\overline{\deg}$ is the graph's average degree and α is a scaling factor; otherwise, it is cold. Fig. 8b shows execution time across graphs for $\alpha \in [1, 5]$. We observe that larger values of α lead to higher execution time, as fewer vertices are classified as hot. This leads to a smaller or less-effectively-defined dense submatrix, diminishing the impact of Rebo's optimizations for improving locality in these critical regions. The best performance is achieved when $\alpha = 1$, which is therefore adopted in this paper.

VI. Related Works

Graph processing is playing an increasingly vital role in solving real-world problems [45]. To handle it efficiently, various frameworks are designed for in-memory processing on multicore systems. Ligra generalizes graph operations with vertex-centric programming model [11], which allows intuitive description of graph algorithms. Polymer enhances Ligra with the awareness of Non-Uniform Memory Access (NUMA) architecture [12]. GraphMat [25] translates graph algorithm into matrix multiplications and outperforms many popular graph processing frameworks.

Distributed systems are another important avenue for the development of high-performance graph processing. Google is the pioneer introducing the vertex-centric model as a part of the framework Pregel [46]. Based on it, Facebook and Apache open-source their advanced variant Giraph, which is used in production [47]. However, compared with the distributed frameworks, graph processing on multicore systems avoids expensive network communications. Thus, the performance of in-memory graph processing is sometimes better than the distributed counterpart [13].

Sparse Matrix-Vector Multiplication (SpMV) is a fundamental kernel for scientific computations. The performance of this operation is memory-bound [48], where memory traffic dominates the computing time. Thereby, there are numerous works attempting to improve memory access patterns, e.g., by limiting the access range [28], sorting the propagation [17] and reducing traffic volume [18]. The cache-oriented optimizations are widely adopted by scientific computing software, e.g., PLASMA [49].

Hybrid formats are often employed to leverage the strengths of different representations. For example, VHCC combines COO and CSR to better handle irregular matrices [50]. Similarly, LAV [44] partitions a graph into dense and sparse regions, processing the dense part with vectorized instructions. Vectorization, enabled by SIMD on modern CPUs, is a key optimization for SpMV [44], [51]–[53]. In addition, SpMV on GPUs has been an active area of research [54]–[56], but remains challenging due to the mismatch between SpMV's memory-bound nature and GPUs' design for compute-intensive workloads.

VII. Conclusion

In this paper, we demonstrate that a straightforward combination of graph reordering and matrix blocking does not consistently improve graph processing on power-law graphs. To address this, we propose *Rebo*, a synergistic integration of graph and matrix optimizations. Rebo captures graph locality through degree- and structure-based reordering, reorganizing the adjacency matrix into dense, sparse, and void submatrices, with the dense submatrix further divided into cache-friendly blocks of varying sizes for load balancing. During execution, the submatrices are processed under distinct paradigms. Comprehensive experiments demonstrate significant speedups over state-of-the-art frameworks and clarify Rebo's effective scope. Future work will explore extending Rebo to GPUs and distributed systems.

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] S. M. Hedetniemi, S. T. Hedetniemi, and A. L. Liestman, "A survey of gossiping and broadcasting in communication networks," *Networks*, vol. 18, no. 4, pp. 319–349, 1988.

[2] P. J. Carrington, J. Scott, and S. Wasserman, *Models and methods in social network analysis*. Cambridge university press, 2005, vol. 28.

[3] A. Fabregat, F. Korninger, G. Viteri, K. Sidiropoulos, P. Marin-Garcia, P. Ping, G. Wu, L. Stein, P. D'Eustachio, and H. Hermjakob, "Reactome graph database: Efficient access to complex pathway data," *PLoS computational biology*, vol. 14, no. 1, p. e1005968, 2018.

[4] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–9.

[5] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *ACM SIGCOMM computer communication review*, vol. 29, no. 4, pp. 251–262, 1999.

[6] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1813–1828.

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

[8] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 203–214.

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

[10] Y. Chen and Y.-C. Chung, "Workload balancing via graph reordering on multicore systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 5, pp. 1231–1245, 2021.

[11] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[12] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2015, pp. 183–193.

[13] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 301–316.

[14] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 456–471.

[15] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[16] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[17] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 820–831.

[18] K. Lakhotia, R. Kannan, and V. Prasanna, "Accelerating pagerank using partition-centric processing," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 427–440.

[19] E.-J. Im, *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[20] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 293–302.

[21] X. Zhi, X. Yan, B. Tang, Z. Yin, Y. Zhu, and M. Zhou, "Corograph: Bridging cache efficiency and work efficiency for graph algorithm execution," *Proceedings of the VLDB Endowment*, vol. 17, no. 4, pp. 891–903, 2023.

[22] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "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*, 2017, pp. 93–104.

[23] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 631–643.

[24] Y. Ahmad, O. Khattab, A. Malik, A. Musleh, M. Hammoud, M. Kutlu, M. Shehata, and T. Elsayed, "La3: A scalable link-and locality-aware linear algebra-based graph analytics system," *Proceedings of the VLDB Endowment*, vol. 11, no. 8, pp. 920–933, 2018.

[25] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *arXiv preprint arXiv:1503.07241*, 2015.

[26] T. A. Davis, "Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.

[27] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang, "Lagraph: A community effort to collect graph algorithms built on top of the graphblas," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019, pp. 276–284.

[28] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *ACM SIGOPS Operating Systems Review*, vol. 25, no. Special Issue, pp. 63–74, 1991.

[29] "Laboratory for Web Algorithmics," https://law.di.unimi.it/datasets.php, 2011, [Online; accessed 21-Sept-2021].

[30] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[31] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, "Gpop: A scalable cache-and memory-efficient framework for graph processing over parts," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 1, pp. 1–24, 2020.

[32] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216.

[33] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, 2007, pp. 29–42.

[34] S. Schelter and J. Kunegis, "Tracking the trackers: A large-scale analysis of embedded web trackers," in *Tenth International AAAI Conference on Web and Social Media*, 2016.

[35] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *The semantic web*. Springer, 2007, pp. 722–735.

[36] O. Lehmberg, R. Meusel, and C. Bizer, "Graph structure in the web: aggregated by pay-level domain," in *Proceedings of the 2014 ACM conference on Web science*, 2014, pp. 119–128.

[37] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 591–600.

[38] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in twitter: The million follower fallacy," in *fourth international AAAI conference on weblogs and social media*, 2010.

[39] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[40] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "The graph structure in the web–analyzed on different aggregation levels," *The Journal of Web Science*, vol. 1, 2015.

[41] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, 2013, pp. 1343–1350.

[42] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[43] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350.

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

[44] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Speeding up spmv for power-law graph analytics by enhancing locality & vectorization," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[45] V. Balaji, "Input, representation, and access pattern guided cache locality optimizations for graph analytics," Ph.D. dissertation, Carnegie Mellon University, 2021.

[46] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[47] C. Martella, R. Shaposhnik, D. Logothetis, and S. Harenberg, *Practical graph analytics with apache giraph*. Springer, 2015, vol. 1.

[48] A. Buluc, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 721–733.

[49] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, P. Wu, I. Yamazaki, A. YarKhan, M. Abalenkovs, N. Bagherpour *et al.*, "Plasma: Parallel linear algebra software for multicore using openmp," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 2, pp. 1–35, 2019.

[50] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 136–145.

[51] J. L. Greathouse and M. Daga, "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, 2014, pp. 769–780.

[52] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 273–282.

[53] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: Efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 149–162.

[54] Y. Chen and J. X. Yu, "Bitmap-based sparse matrix-vector multiplication with tensor cores," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 1135–1144.

[55] Y. Lu and W. Liu, "Dasp: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–14.

[56] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 68–78.

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.

Appendix: Artifact Description

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

The paper presents Rebo, a unified framework for graph processing on multicore systems. The main contributions are:

- C_1 A synergistic integration of graph reordering and matrix blocking to capture locality patterns effectively.
- C_2 A graph transformation technique that classifies vertices (hot, cold, frozen) to extract dense, sparse, and void submatrices.
- C_3 A hybrid execution model that processes dense submatrices with a partition-centric paradigm and sparse submatrices with a vertex-centric paradigm.

B. Computational Artifacts

The artifact includes the source code for Rebo, scripts for downloading public graph datasets, and scripts for reproducing the experiments.

A_1 https://github.com/yuang-chen/Rebo-IPDPS26

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3	Table II Figures 4-8

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

Artifact A_1 contains the complete C++ implementation of the Rebo framework. It supports the main contributions by implementing the graph reordering and blocking preprocessing (C_1, C_2) and the hybrid execution engine (C_3).

Expected Results

We expect the artifact to successfully compile on a modern Linux system with a C++23 compliant compiler. The experiments should reproduce the performance results reported in Section V. Specifically, Rebo should demonstrate speedups over baselines like Ligra, GraphIt, and GPOP on power-law graphs for PageRank, BFS, and Connected Components.

Expected Reproduction Time (in Minutes)

The total time is approximately 2 hours, dependent on network speed for downloading datasets.

- Setup: 3 minutes for compilation.
- Data Preparation: 1–2 hours for downloading and format conversion.
- Execution: 30 minutes for running full all executables.

Artifact Setup (incl. Inputs)

Hardware:

- CPU: Multicore x86_64 processor. The paper uses dual Intel Xeon Silver 4210 processors (20 cores total).
- Memory: At least 64GB DRAM is recommended to handle large graphs like *host* and *kron*.

Software:

- OS: Linux (Ubuntu 22.04 LTS tested).
- Compiler: GCC version 13 or higher (required for C++23 support via '-std=c++23').
- Libraries: 'OpenMP' for parallelization, 'fmt' for formatting, 'CMake' for build management.
- Tools: 'perf' and 'likwid' (optional, for hardware counter profiling).

Datasets / Inputs: The artifact requires 11 graph datasets: *live, orkut, track, wiki, pld, twitter, mpi, kron, host, road*, and *urand*.

- Most datasets are available from public repositories like SNAP or KONECT.
- Synthetic graphs (*kron*, *urand*) can be generated using the Gap Benchmark Suite generator.
- A mini dataset *wiki.csr* is included in the repository for quick testing.

Installation and Deployment:

1) Clone the repository from GitHub.
2) Install dependencies ('g++-13', 'cmake', 'libomp-dev').
3) Compile using CMake:

```
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make -j
```

Artifact Execution

Parameters include threads (auto-detected via OpenMP), rounds (`-r`, outer iterations), iterations (`-i`, inner iterations), subgraph size (`-s`, cache block size in KB), degree type (`-d`, 0=in-degree, 1=out-degree), and transpose flag (`-t`).

The framework provides three executable variants:

- `mix`: Full Rebo with reordering and blocking
- `block`: Blocking without reordering
- `pull`: Baseline CSC pull without optimizations

Artifact Analysis (incl. Outputs)

The artifact outputs console logs containing:

- Experimental settings (threads, rounds, iterations, block size, reordering algorithm)
- Graph statistics (#vertices, #edges, hot/cold/frozen vertex counts)
- Hot/Cold/Frozen vertex ratios
- Elapsed time in milliseconds (total execution time across all rounds and iterations)

Authorized licensed use limited to: Chinese University of Hong Kong CUHK(Shenzhen). Downloaded on July 07,2026 at 00:31:01 UTC from IEEE Xplore. Restrictions apply.