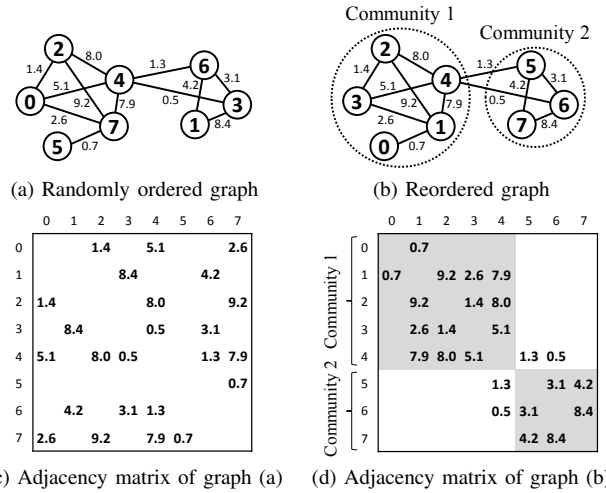
Rabbit Order: Just-in-time Parallel Reordering for Fast Graph Analysis

Junya Arai*, Hiroaki Shiokawa†, Takeshi Yamamuro*, Makoto Onizuka‡ and Sotetsu Iwamura*
* *Nippon Telegraph and Telephone Corp., Japan. Email: {arai.junya, yamamuro.takeshi, iwamura.sotetsu}@lab.ntt.co.jp*
† *University of Tsukuba, Japan. Email: shiokawa@cs.tsukuba.ac.jp*
‡ *Osaka University, Japan. Email: onizuka@ist.osaka-u.ac.jp*

***Abstract*—Ahead-of-time data layout optimization by vertex reordering is a widely used technique to improve memory access locality in graph analysis. While reordered graphs yield better analysis performance, the existing reordering algorithms use significant amounts of computation time to provide efficient vertex ordering; hence, they fail to reduce end-to-end processing time. This paper presents a first algorithm for just-in-time parallel reordering, named *Rabbit Order*. It reduces end-to-end runtime by achieving high locality and fast reordering at the same time through two approaches. The first approach is *hierarchical community-based ordering*, which exploits the locality derived from hierarchical community structures in real-world graphs. Our ordering fully leverages low-latency cache levels by mapping hierarchical communities into hierarchical caches. The second approach is *parallel incremental aggregation*, which improves the runtime efficiency of reordering by decreasing the number of vertices to be processed. In addition, this approach utilizes lightweight atomic operations for concurrency control to avoid locking overheads and achieve high scalability. Our experiments show that Rabbit Order significantly outperforms state-of-the-art reordering algorithms.**



(a) Randomly ordered graph (b) Reordered graph

(c) Adjacency matrix of graph (a) (d) Adjacency matrix of graph (b)

Figure 1. Examples of weighted graphs and their adjacency matrices. The numbers on the edges are their weights. Graph (a) has randomly ordered vertex IDs, and graph (b) is the reordered equivalent. In graph (b), the vertices in each community (shown by the dotted circles) have contiguous ID numbers, and they make dense diagonal blocks (the shadowed squares in matrix (d)).

I. Introduction

Graph analysis plays an important role in a wide range of scientific and industrial fields for knowledge discovery from *real-world graphs*, e.g., web graphs, social networks, and protein-protein interaction networks. In response to the emergence of billion-edge real-world graphs, high-performance graph analysis has attracted significant attention in recent years [1]–[4].

The main obstacle to improving performance is poor locality of memory accesses, which results from unstructured and complex relationships among entities represented by graphs [5]. For example, consider the graph shown in Figure 1(a) and suppose that the vertices are stored in an array. Assuming an analysis algorithm first examines the relationships around vertex 0, it would access vertex 0 itself and its neighbors (i.e., vertices 2, 4, and 7). Generally, it would next examine the relationships around vertex 1 by accessing vertices 1, 3, and 6. Thus, vertices 0, 2, 4, 7, 1, 3, and 6 would be consecutively accessed. Unfortunately, this access pattern incurs cache misses because each vertex is accessed only once (poor temporal locality) and vertices are not co-located in memory (poor spatial locality). Furthermore, the cache misses lead to inter-core communication and memory bandwidth saturation during parallel processing on multicore processors; hence, parallel graph processing shows poor scalability [5], [6].

To improve locality, *reordering* algorithms have been adopted by many applications [2], [7]–[10]. Reordering is a technique that optimizes both the computation order and the data layouts by modifying the vertex ordering. Figure 1(b) shows an example of a reordered graph. In this graph, neighboring vertices have close ID numbers, and so they will be computed consecutively and co-located in memory. As a result, analysis algorithms repeatedly access vertices that have recently been used (high temporal locality) and stored nearby in memory (high spatial locality). For example, when the algorithm examines the relationships around vertices 0 and 1, it consecutively accesses those vertices and their neighbors, i.e., vertices 0, 1, 1, 0, 2, 3, and 4. This pattern shows high temporal and spatial locality since it repeatedly accesses vertices 0 and 1, and it accesses vertices 0 to 4, which are stored contiguously in memory. The major advantage of reordering is its effectiveness in various graph analyses. For example, previous studies [2], [11], [12] have improved the performance of PageRank and breadth-first search (BFS) by reordering graphs ahead of time.

Citation: J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka and S. Iwamura, "Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis," 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, IL, 2016, pp. 22-31. doi: 10.1109/IPDPS.2016.110

While reordering is an effective way of improving locality, it still faces a challenging problem, that is, the high computational cost to produce an efficient ordering. Since real-world graphs continuously change their topologies [13], we need to reorder graphs just in time; however, the long reordering time increases the *end-to-end* runtime, namely the total runtime of reordering and subsequent graph analysis. Hence, it is a worthwhile task to overcome the end-to-end performance limitation by just-in-time reordering that achieves high locality and a short reordering time at the same time.

In this paper, we propose *Rabbit Order*, a just-in-time parallel reordering algorithm. Rabbit Order achieves both high locality and a short reordering time by employing two approaches. The first approach, *hierarchical community-based ordering*, achieves high locality while mapping two hierarchies: (i) hierarchical community structures of real-world graphs and (ii) hierarchical CPU caches. A community has dense inner-edges, and hence, during graph analysis, a vertex in a community involves frequent accesses to other vertices in the community. Moreover, in a hierarchical community, inner communities have denser inner-edges and involve more frequent accesses to the vertices in each community. Rabbit Order exploits this property to improve locality by co-locating vertices in each community and within each subordinate inner community recursively. Since every community is an agglomeration of small inner communities that can be accommodated at each cache level (e.g., L1 and L2), this ordering fully leverages the hierarchical caches. The second approach, *parallel incremental aggregation*, shortens the reordering time by efficiently extracting communities. This approach incrementally aggregates vertices in the same community in parallel and thus decreases the number of vertices to be processed. We also present a scalable parallel implementation of Rabbit Order with a lightweight concurrency control by using atomic operations instead of locks.

Our main evaluations using PageRank show that Rabbit Order can reorder billion-edge graphs in a dozen seconds and that it improves end-to-end performance by 2.2 times on average, whereas other reordering methods do not show such remarkable improvements. Moreover, additional evaluations reveal that Rabbit Order also improves the end-to-end performance of various analysis algorithms such as BFS.

Our contributions can be summarized as follows:

1) **High locality:** Rabbit Order yields an effective ordering that exhibits locality higher than or equal to those of state-of-the-art methods.
2) **Fast reordering:** Rabbit Order has a short runtime, and hence it significantly outperforms the other methods in terms of end-to-end runtime.
3) **Extensive evaluation:** We evaluate the effectiveness of Rabbit Order by comparing it with seven reordering methods with regards to performance improvements for various analysis algorithms.

Table I
NOTATIONS

Symbol	Definition
V	Set of vertices; $V = \{0, 1, \ldots, n-1\}$
E	Set of edges; $E \subseteq V \times V$
n	Number of vertices; $n = \|V\|$
m	Number of edges; $m = \|E\|$
w_{uv}	Weight of edge between vertices u and v
$d(v)$	Weighted degree of vertex v
$N(C)$	Set of vertices connected to vertices in $C \subseteq V$
$\Delta Q(u, v)$	Gain in modularity yielded by merging u and v

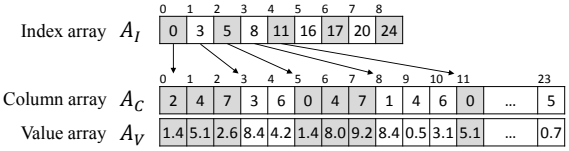


Figure 2. Three CSR arrays representing the matrix of Figure (c).

The rest of this paper is organized as follows. Section II presents the background of this paper. We detail the Rabbit Order algorithm in Section III and show the experimental results in Section IV. After reviewing related work in Section V, we conclude the paper in Section VI.

II. BACKGROUND

This section presents the background on our work and states the problem to be tackled. The notations used in this paper are listed in Table I.

A. Memory Access Pattern of Graph Analysis

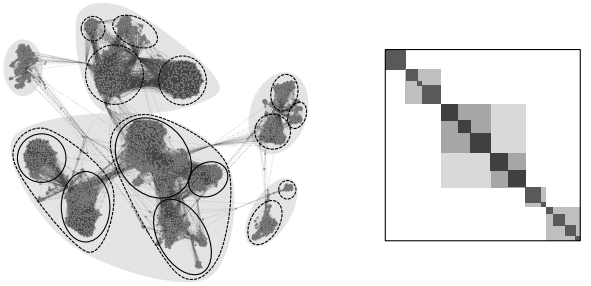
While graph analysis algorithms involve various access patterns, many popular algorithms such as PageRank, Random Walk with Restart [14], Label Propagation [15], and spectral clustering [16] show similar patterns involved in *sparse matrix-vector multiplication* (*SpMV*). SpMV is the computation of $\boldsymbol{y} = A\boldsymbol{x}$, where $\boldsymbol{x}$ and A are a dense vector and a sparse matrix, respectively. The analysis algorithms generally use a weighted adjacency matrix as A. Sparse matrices can be compactly stored in the widely used *compressed sparse row* (*CSR*) format. Figure 2 shows the CSR form of the matrix shown in Figure 1(c). CSR stores nonzero values in a matrix and their position in three arrays: index array A_I, column array A_C, and value array A_V. The elements in A_I act as the starting indices of the elements in A_C and A_V that correspond to each row. Specifically, the elements of row i are stored in indices $A_I[i]$ to $A_I[i+1]-1$ of A_C and A_V. The elements in A_C and A_V are the column number and value in that column, respectively.

The pseudocode in Algorithm 1 is used to solve SpMV $\boldsymbol{y} = A\boldsymbol{x}$. This algorithm sequentially accesses the arrays except $\boldsymbol{x}$, but it involves irregular indirect accesses to $\boldsymbol{x}$ through A_C (line 4). For instance, the CSR in Figure 2 involves consecutive accesses to $\boldsymbol{x}[2], \boldsymbol{x}[4], \boldsymbol{x}[7], \ldots, \boldsymbol{x}[5]$. These accesses exhibit poor locality, and this is why SpMV suffers from cache misses.

Algorithm 1 Sparse matrix-vector multiplication

Input: $n \times n$ CSR matrix $A = (A_V, A_C, A_I)$ and vector $\boldsymbol{x} \in \mathbb{R}^n$
Output: $\boldsymbol{y} = A\boldsymbol{x}$

```
1 for v = 0 to n − 1 do
2   y[v] ← 0
3   for k = A_I[v] to A_I[v + 1] − 1 do
4     y[v] ← y[v] + A_V[k] x[A_C[k]]
```



(a) Hierarchical communities (b) Reordered adjacency matrix

Figure 3. Example of hierarchical communities and the reordered adjacency matrix. Figure (a) shows communities and their inner communities in the Facebook social network [17] as shadows and circles. After reordering, the adjacency matrix of this graph has the distribution of non-zero values shown in Figure (b). Darker blocks contain denser non-zero values.

B. Problem Statement

Although reordering improves the locality of access to array $\boldsymbol{x}$, it also incurs computation time in addition to that taken by SpMV. To reduce end-to-end runtime, we tackle the following problem by presenting fast reordering algorithm:

Problem 1 (Minimizing cache misses by reordering)**.** *Given graph $G = (V = [0, n), E \subseteq V \times V)$ and letting A be the adjacency matrix of G, find permutation $\pi : V \to N$ of the vertex ordering that minimizes the number of cache misses produced by Algorithm 1 for matrix A.*

In the following sections, we assume that the given graph is undirected and unweighted for simplicity. Directed and/or weighted graphs can be handled with small modifications.

III. Rabbit Order

To achieve high locality and fast reordering simultaneously, we propose a novel just-in-time parallel reordering algorithm, Rabbit Order. This section describes its hierarchical community-based ordering for achieving high locality and parallel incremental aggregation for fast reordering.

A. Hierarchical Community-based Ordering

Our approach to improving locality is to make dense blocks of non-zero values in a matrix. A dense block ensures high temporal and spatial locality during the SpMV computation because each row within the block has a homogeneous pattern of accesses to a narrow memory region. For example, running Algorithm 1 with the matrix in Figure 1(d) lets rows 0 to 4 have multiple accesses to the elements of $\boldsymbol{x}$ within indices 0 to 4; thus, they show high temporal locality. In addition, since the accessed elements are gathered within indices 0 to 4 of $\boldsymbol{x}$, spatial locality is also high. Regarding real-world graphs, we can make dense blocks in an adjacency matrix by capturing community structures. Since communities have dense edges, which are represented by non-zero values in the matrix, dense diagonal blocks appear if the vertices within each community are co-located. For example, the two communities shown in Figure 1(b) yield the two dense blocks shown in Figure 1(d).

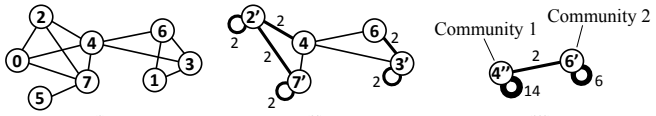
Algorithm 2 Overview of Rabbit Order

Input: Graph $G = (V, E)$
Output: Permutation $\pi : V \to N$ for new vertex ordering

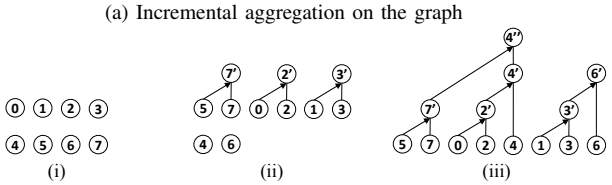
```
  ▷ Perform hierarchical community-based ordering
1 dendrogram ← CommunityDetection()
2 return OrderingGeneration(dendrogram)
3 function CommunityDetection()
    ▷ Perform incremental aggregation
4   for each u ∈ V in increasing order of degree do
5     v ← neighbor of u that maximizes ΔQ(u, v)
6     if ΔQ(u, v) > 0 then
7       Merge u into v and record this merge in dendrogram
8   return dendrogram
9 function OrderingGeneration(dendrogram)
10  new_id ← 0
11  for each v ∈ V in DFS visiting order on dendrogram do
12    π[v] ← new_id; new_id ← new_id + 1
13  return π
```

Moreover, to improve locality further, we present hierarchical community-based ordering by focusing on the hierarchical community structures in real-world graphs. Figure 3(a) shows an example of hierarchical communities. Each community contains inner communities that have denser edges than it. Hence, as shown in Figure 3(b), we can recursively make denser inner blocks within the outer block by recursively co-locating vertices in each inner community. Since frequently accessed data are likely to remain in lower-latency cache levels, this ordering approach lets hierarchical communities make good use of hierarchical caches. Specifically, assuming two-level caches (i.e., L1 and L2), $\boldsymbol{x}$ elements accessed by denser inner communities are cached in L1, and those accessed by sparser outer communities are cached in L2. Thus, hierarchical community-based ordering greatly improves locality while appropriately mapping hierarchical communities into hierarchical caches.

Algorithm 2 outlines Rabbit Order. Given a graph G, the algorithm outputs permutation π that gives a new vertex ordering. Rabbit Order performs hierarchical community-based ordering in two steps (lines 1 and 2). The first step, the CommunityDetection function, extracts hierarchical communities in an agglomerative manner and simultaneously constructs a dendrogram. The second step, the OrderingGeneration function, converts the dendrogram into a new ordering such that, for every community hierarchy, vertices in the same community are co-located.



(a) Incremental aggregation on the graph



(b) Dendrogram construction

Figure 4. Running example of community detection by incremental aggregation. Incremental aggregation merges vertex 5 to 7, 1 to 3, 0 to 2, $3'$ to 6, $2'$ to 4, and $7'$ to $4'$ (the merged vertices are labeled with primes). It eventually detects communities 1 (vertex $4''$) and 2 (vertex $6'$).

The COMMUNITYDETECTION and ORDERINGGENERATION functions are detailed in the following subsections.

B. Community Detection

In this subsection, we describe the sequential approach to community detection and then parallelize it.

1) Sequential Algorithm: Community detection algorithms have been extensively studied. Particularly, *modularity*-based ones are widely accepted and well capture hierarchical structures of communities. Modularity Q [18] is a quality measure that provides a higher value when a graph is partitioned better, namely, each community has denser inner-community edges and sparser intra-community edges. From the viewpoint of performance, one of the most efficient approaches is community detection using the *incremental aggregation* technique [3]. It incrementally aggregates vertices that are clustered into the same community and reduces computational costs by rapidly coarsening the graph. Thus, we design Rabbit Order by employing incremental aggregation to reduce the reordering time.

Figure 4 illustrates a running example of our approach to hierarchical community detection. Each vertex is picked as a *source* vertex and merged into a *destination* vertex, which is a neighbor of the source vertex. After the merge, the destination vertex has weighted edges and represents a community consisting of the source and destination vertices. For example, comparing graphs (i) and (ii) in Figure 4(a), we can see that source vertex 0 has been merged into destination vertex 2. By repeatedly merging vertices, our approach extracts hierarchical communities while constructing a dendrogram at the same time. Figure 4(b) shows the dendrogram constructed by the incremental aggregation shown in Figure 4(a). Merged destination vertices form a tree by aggregating the inner communities.

The COMMUNITYDETECTION function in Algorithm 2 is the specific procedure of our approach. It consists of three steps: (i) picking a source vertex (line 4), (ii) finding the best destination vertex in terms of modularity improvement (line 5), and (iii) merging the source vertex and the destination vertex if the modularity improves (lines 6–7). In the first step, source vertex u is picked in increasing order of degree. This ordering heuristic reduces the computational cost of incremental aggregation [3]. The second step searches for destination vertex v that yields the highest value of the modularity gain, which is defined as follows [3]:

$$\Delta Q(u,v) = 2\left\{\frac{w_{uv}}{2m} - \frac{d(u)d(v)}{(2m)^2}\right\} \tag{1}$$

where m is the number of edges within the initial graph, w_{uv} is the edge weight between u and v, and $d(\cdot)$ is the weighted degree of the vertex. The third step merges u into v if the modularity improves (i.e., $\Delta Q(u,v) > 0$). The merged destination vertex v' has a self-loop edge of weight $2w_{uv} + w_{uu} + w_{vv}$ and has the edges of both u and v (we assume that all the edges initially have weight 1). If u and v have a common neighbor t, $w_{v't}$ is set to $w_{ut} + w_{vt}$. On the other hand, if the modularity does not improve (i.e., $\Delta Q(u,v) \leq 0$), u is not merged and remains in the graph as a *top-level* vertex. Top-level vertices become the roots of the dendrogram. For example, vertices 4 and 6 of dendrogram (iii) in Figure 4(b) are top-level vertices.

Thus, Rabbit Order extracts communities by using incremental aggregation. This approach rapidly decreases the number of vertices, and, in contrast to iterative approaches [19], [20], it does not traverse all the vertices and edges multiple times. For these reasons, Rabbit Order can extract communities and construct the dendrogram quickly.

2) Parallelization: To fully exploit the performance of multicore processors, we parallelize the community detection algorithm by using atomic operations for lightweight concurrency control and avoiding locking overheads.

The COMMUNITYDETECTION function in Algorithm 2 can be parallelized at the for-loop in line 4. However, this naive approach may cause conflicts at the vertex merge (line 7) as it modifies edges incident to the destination vertex and removes the source vertex. Similarly, conflicts are possible in the modification of the dendrogram. This sort of problem can usually be solved by using locks. For example, GraphLab [1], which is a well-established framework for parallel graph processing, provides several data consistency models using locks. The above problem can also be solved by locking the source and destination vertices during the merge procedure. However, locks significantly limit parallelism and degrade scalability. Although modern processors provide atomic operations such as compare-and-swap (CAS) as a low-cost solution instead of locks, CAS can atomically modify only 16 bytes on popular processors (e.g., x86-64 and POWER8).

To improve scalability by taking the advantage of CAS, we introduce a *lazy aggregation* approach that reduces the amount of data necessary to be atomically modified. In lazy aggregation, when our algorithm decides to merge vertex u

Algorithm 3 Parallel community detection

Input: Graph $G = (V, E)$
Output: $Toplevel \in V$, a set of top-level vertices, and a dendrogram represented by $atom[\cdot].child$ and $sibling[\cdot]$

▷ Initialize variables
1 **for each** $u \in \mathbb{V}$ **do**
2 $atom[u].degree \leftarrow d(u)$ ▷ Total edge weight
3 $atom[u].child \leftarrow$ UINT32_MAX ▷ Vertex merged to u
4 $dest[u] \leftarrow u$ ▷ Community that u belongs to
5 $sibling[u] \leftarrow$ UINT32_MAX ▷ List of sibling members
6 $Toplevel \leftarrow \emptyset$ ▷ Set of top-level vertices
▷ Incrementally aggregate vertices
7 **for each** $u \in V$ in increasing order of degree **do in parallel**
8 $degree_u \leftarrow$ UINT64_MAX ▷ Set an invalid degree
9 AtomicSwap($degree_u, atom[u].degree$) ▷ Invalidate u
10 $v \leftarrow$ FindBestDestination(u)
11 **if** $\Delta Q(u, v) \leq 0$ **then** ▷ If the modularity does not improve
12 atom[u].degree $\leftarrow$ degree$_u$ ▷ Restore the value
13 $Toplevel \leftarrow Toplevel \cup \{u\}$ ▷ u is a new top-level
14 **continue** ▷ Move to the next vertex
15 $atom_v \leftarrow$ AtomicLoad($atom[v]$)
16 **if** $atom_v.degree \neq$ UINT64_MAX **then** ▷ Check validity
17 $sibling[u] \leftarrow atom_v.child$
18 $atom'_v.degree \leftarrow atom_v.degree + degree_u$
19 $atom'_v.child \leftarrow u$
20 **if** CAS($atom[v], atom_v, atom'_v$) **then**
21 $dest[u] \leftarrow v$ ▷ u is successfully merged to v
22 **continue** ▷ Move to the next vertex
23 $sibling[u] \leftarrow$ UINT32_MAX ▷ Rollback
24 $atom[u].degree \leftarrow degree_u$ ▷ Rollback
25 Save u to retry merge later

into vertex v, it only registers u as a community member of v instead of modifying edges of v and removing u. The community members are aggregated into v when v is about to be merged into another destination vertex. Registration as a community member can be implemented by using CAS since it requires atomic modification of only a small set of variables. The details are presented in the description of the overall algorithm.

Algorithm 3 is our parallel community detection algorithm. First, it initializes variables (lines 1–6). $atom$ is an array of complex data for lazy aggregation that needs to be atomically modified. Each element in the array consists of the following two variables: (i) $degree$, the total degree of the community members, and (ii) $child$, a vertex ID that is a part of a data structure for the dendrogram. Even assuming large-scale graphs of up to $2^{32} - 1$ vertices and $2^{64} - 1$ edges, the total size of the variables is only 12 bytes, since unsigned 64-bit and 32-bit integers can respectively accommodate all the possible values of $degree$ and $child$. Hence, each element can be atomically modified by CAS[1]. Lazy aggregation also introduces $dest$, a mapping of each vertex to the community that the vertex belongs to. Each vertex is initially a singleton community. $sibling$ represents the dendrogram together with $child$. Figure 5 shows their

[1]Even if a processor supports only 8-byte CAS, Rabbit Order can be implemented by using a 32-bit floating point number for $degree$ instead of an unsigned 64-bit integer. In spite of the inaccuracy in the value, this substitution empirically has negligible impact on the result.

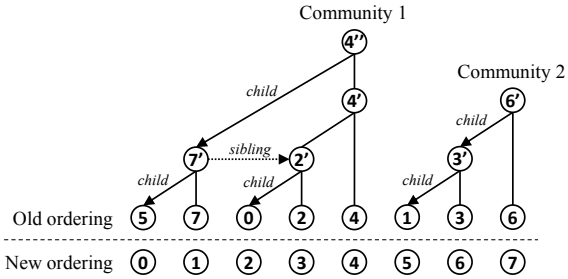


Figure 5. New ordering derived from dendrogram (iii) in Figure 4(b) and its link structure represented by $child$ and $sibling$ in Algorithm 3.

Algorithm 4 FindBestDestination for Algorithm 3

1 **function** FindBestDestination(u)
2 $C \leftarrow$ vertices reachable from u on the dendrogram
3 **for each** $v \in N(C)$ **do**
4 **while** $dest[dest[v]] \neq dest[v]$ **do** ▷ Until a top-level is found
5 $dest[v] \leftarrow dest[dest[v]]$ ▷ Trace the destination of v
6 $Neighbor \leftarrow \{\, dest[t] \mid t \in N(C) \,\}$
7 **for each** $v \in Neighbor$ **do**
8 $w_{uv} \leftarrow \sum_{\{\ (s,t)\ \mid\ (s,t)\in E, s\in C, dest[t]=v\ \}} w_{st}$
9 $E \leftarrow \{\, (s,t) \in E \mid s \notin C \,\} \cup \{\, (u,v) \mid v \in Neighbor \,\}$
10 **return** $\mathrm{argmax}_{v \in Neighbor} \Delta Q(u, v)$

connections as arrows. $atom[u].child$ points the last vertex that was merged into vertex u, and $sibling[u]$ points the previous vertex that was merged into the same vertex as u. For $child$ and $sibling$, we use UINT32_MAX ($= 2^{32} - 1$) as an invalid vertex ID. $Toplevel$, a set of top-level vertices, is initialized to an empty set.

After the initialization, the algorithm starts incremental aggregation. Each thread picks vertex u and, to avoid conflicts, prohibits other threads from merging vertices into u by invalidating u (lines 7–9). Invalidated vertices have an invalid degree represented by UINT64_MAX ($= 2^{64} - 1$). The destination vertex (vertex v) is given by the FindBestDestination function (line 10). We describe this function later. In line 11, the algorithm checks if the merge improves the modularity. $atom[\cdot].degree$ is used in computation of $\Delta Q(u, v)$ (Equation 1) to provide $d(u)$ and $d(v)$. If the modularity does not improve, u is added to $Toplevel$ (lines 12–14); otherwise, the algorithm attempts to merge u into v by using optimistic concurrency control. If v is not invalidated (lines 15–16), u is registered as a community member of v for lazy aggregation (lines 17–20). This pseudocode assumes that CAS($x, x_{\text{expected}}, x_{\text{desired}}$) atomically stores x_{desired} in variable x if $x = x_{\text{expected}}$. If CAS succeeds in storing the new value, v is stored as the destination of u in $dest$ (line 21). If the merge fails due to concurrency issues, the values are rolled back, and the merge of u is tried again later (lines 23–25).

Algorithm 4 details the FindBestDestination function. $N(C)$ denotes the set of vertices connected to a vertex in $C \subseteq V$ (i.e., $N(C) = \{\, v \in V \mid (v, t) \in E, t \in C \,\}$). This

function consists of four steps. The first step (line 2) collects the vertices in the community of vertex u by traversing the dendrogram from u. The second step (lines 3–5) updates $dest$ so that $dest[v]$ points the vertex representing the community that v belongs to. This update is necessary since vertices are repeatedly merged in an agglomerative manner. Consider the merge process shown in Figure 4(b). In this process, vertex 1 is merged into vertex 3 ($dest[1] = 3$), and then vertex 3 is merged into vertex 6 ($dest[3] = 6$). Vertex 6 remains as a top-level vertex ($dest[6] = 6$). As a result, vertex 1 ends up belonging to the community of vertex 6, and so $dest[1]$ is updated to 6 ($= dest[dest[1]]$). The third step (lines 6–9) is a counterpart of lazy aggregation; this step aggregates the edges of all the community members by replacing their endpoints with vertices in $dest$, and all the aggregated edges are reattached to u. In the fourth step (line 10), the function returns a neighbor of u that yields the highest gain in modularity.

While our parallel algorithm avoids the use of locks for high scalability, its results may differ from those of the sequential algorithm due to the asynchronicity in vertex merges. However, Rabbit Order is inherently heuristic for locality improvement, and so we allow this difference if our approach improves parallel reordering performance. Since this difference may impact ordering quality (i.e. a gain in locality), Section IV-C experimentally compares sequential execution and parallel execution.

C. Ordering Generation

After the community detection, Rabbit Order generates an ordering by following hierarchical community-based ordering. Here, we present a sequential algorithm for generating an ordering and its parallel equivalent.

1) Sequential Algorithm: The ORDERINGGENERATION function in Algorithm 2 generates new orderings. To recursively co-locate vertices in each hierarchical community, the algorithm performs depth-first search (DFS) from each top-level vertex on the dendrogram and returns the visit order as permutation $\pi : V \to N$. For example, DFS from top-level vertex 4 in Figure 5 visits vertices in the order of 5, 7, 0, 2, and 4 and generates π such that $\pi[5] = 0, \pi[7] = 1, \pi[0] = 2, \pi[2] = 3$, and $\pi[4] = 4$. The permutation for the right tree is generated in a similar manner. This permutation is the final result of Rabbit Order.

2) Parallelization: The above ordering generation can be easily parallelized in three steps. The first step collects a set of top-level vertices in the dendrogram. The second step performs DFS from each top-level vertex in parallel. The DFS produces *local orderings* for each top-level vertex that represent the offsets of each vertex in the community. The third step concatenates these local orderings. For example, for the dendrogram shown in Figure 5, the first step collects top-level vertices $\{4, 6\}$, and the second step produces the local ordering $(5, 7, 0, 2, 4)$ for community 1 and $(1, 3, 6)$ for community 2. The third step concatenates them and yields $(5, 7, 0, 2, 4, 1, 3, 6)$. This is the new ordering.

Table II
GRAPHS USED IN THE EXPERIMENTS

Name	#vertices	#edges	Reference
berkstan	0.7M	7.6M	WEB-BERKSTAN in SNAP [17]
enwiki	4.2M	101.4M	ENWIKI-2013 in LAW [21]
ljournal	4.8M	69.0M	SOC-LIVEJOURNAL1 in SNAP
uk-2002	18.5M	298.1M	UK-2002 in LAW
road-usa	23.9M	57.7M	ROAD_USA in DIMACS [22]
uk-2005	39.5M	936.4M	UK-2005 in LAW
it-2004	41.3M	1150.7M	IT-2004 in LAW
twitter	41.7M	1468.4M	TWITTER-2010 in LAW
sk-2005	50.6M	1949.4M	SK-2005 in LAW
webbase	118.1M	1019.9M	WEBBASE-2001 in LAW

Table III
COMPETITORS

Name	Description
Rabbit	Rabbit Order
Slash	SlashBurn proposed in 2014 [12]
BFS	Unordered parallel BFS proposed in 2014 [23]
RCM	Unordered parallel RCM proposed in 2014 [23]
ND	Multithreaded Nested Dissection proposed in 2013 [4]
LLP	Layered Label Propagation proposed in 2011 [19]
Shingle	Shingle ordering proposed in 2009 [10]
Degree	Increasing order of vertex degree
Random	Random ordering of vertices (baseline)

IV. EVALUATION

This section mainly evaluates the effectiveness of Rabbit Order by applying it to a SpMV-based analysis algorithm. Additionally, the results of the experiments using other analysis algorithms are presented at the end of the section.

All evaluations are performed on a machine with 256GB RAM and dual-socket Intel Xeon E5-2697v2 12-core processors with Hyper-Threading Technology. Each core has a 32KB L1 data cache, a 32KB L1 instruction cache, and a 256KB L2 cache. In addition, each socket has a 30MB shared L3 cache. Table II shows the scales and references of the real-world graphs used in our experiments.

The performance of Rabbit Order is compared with those of the reordering algorithms listed in Table III. We use random ordering as the baseline instead of the ordering given by each dataset publisher. This is because the publishers have modified the ordering from the natural one that is given by data sources of real-world graphs [21]. For example, the natural ordering of web graphs and social graphs should correspond to the visit order of web crawlers and the user ID order of SNSs. Thus, the experiments avoid using the ordering given by the publishers.

All the benchmark programs and reordering algorithms except Nested Dissection and LLP are implemented in C++ and OpenMP and compiled with GCC 4.9.2 using the `-O3` optimization option. For SlashBurn, we use the best parameters shown in the paper [12] (i.e., S-KH and $k = 0.02n$). To implement parallel BFS and RCM, we use the Galois C++ library 2.2.1 [24], as did the authors of

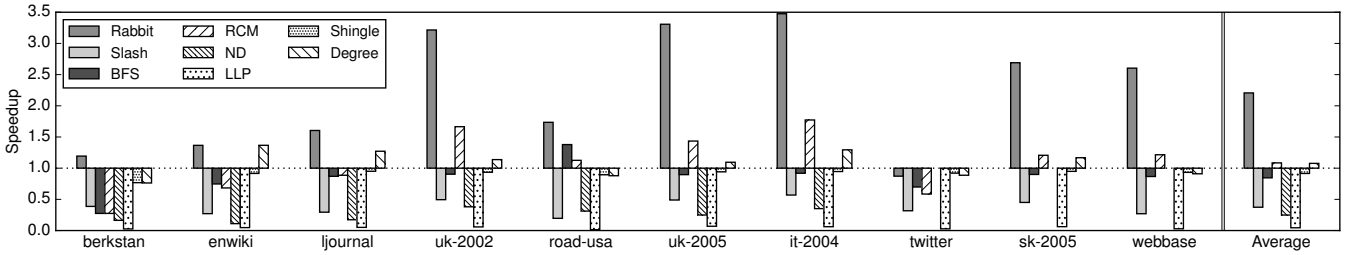


Figure 6. End-to-end performance improvement with 48-thread executions. The speedups are obtained by dividing the analysis time with random ordering by the end-to-end runtime of each reordering algorithm. Results of ND on twitter, sk-2005, and webbase are not available.

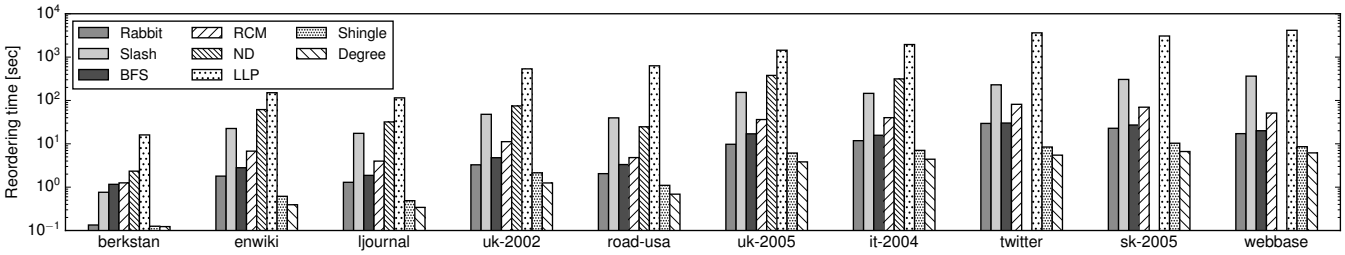


Figure 7. Reordering time with 48-thread executions. Results of ND on twitter, sk-2005, and webbase are not available.

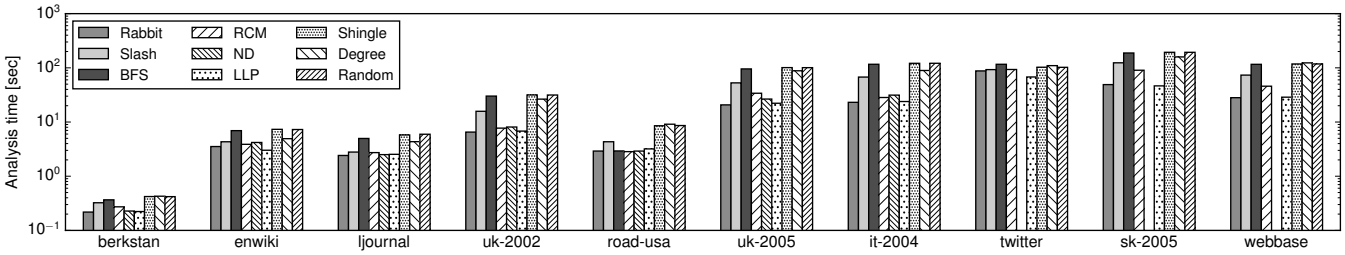


Figure 8. Analysis time of PageRank with 48-thread executions. Results of ND on twitter, sk-2005, and webbase are not available.

those algorithms [23]. We use the public C implementation of multithreaded Nested Dissection version 0.4.4 [25]. We also use the public Java implementation of LLP version 2.3 with the default parameters [21]. All the reordering algorithms except SlashBurn are parallelized. Since Degree and Shingle are essentially simple sorting, we use the `__gnu_parallel::sort` parallel sort function provided by GCC for their parallel implementations.

A. SpMV-based Analysis Algorithm

We use one of the most popular and practical SpMV-based algorithms, PageRank. It iteratively computes the following equation for $k = 0, 1, \ldots$ until convergence:

$$\boldsymbol{s}_{k+1} = (1-c)W\boldsymbol{s}_k + c\boldsymbol{e} \tag{2}$$

where $\boldsymbol{s}_k$ and $\boldsymbol{s}_{k+1}$ is a vector that denotes the PageRank score of each vertex, c is a constant for the teleportation probability, $\boldsymbol{e}$ is a column vector whose elements are $\frac{1}{n}$, and W is a matrix such that $W[u,v] = \frac{1}{d(v)}$ if vertices u and v are connected, and otherwise $W[u,v] = 0$. Following the previous study [9], we define convergence as $|\boldsymbol{s}_{k+1} - \boldsymbol{s}_k| < 10^{-10}$ and $c = 0.15$. Note that $(1-c)$ and $c\boldsymbol{e}$ are immutable during the computation and so have little effect on the memory access. As is common practice, we parallelize SpMV (Algorithm 1) at the outermost for-loop using the thread blocking technique of Williams et al. [26]. In the experiments, we measure the performance of 48-thread execution using all hardware threads on our machine.

B. End-to-end Performance and Breakdown

First, we show the end-to-end performance improvements of the graph analysis. Figure 6 plots the speedups relative to random ordering yielded by each reordering algorithm. Rabbit Order significantly outperforms the other algorithms and improves end-to-end performance on most graphs. Rabbit Order has a 2.21x speedup on average and up to 3.48x speedup on it-2004. On the other hand, most of the other algorithms suffer performance degradations. Even though RCM performs the best among the existing algorithms, it provides only a 1.08x speedup on average. ND fails to reorder the top three largest graphs (i.e., twitter, sk-2005, and webbase) due to memory shortages.

To clarify why performance improves, we breakdown the end-to-end runtime into the reordering time and subsequent

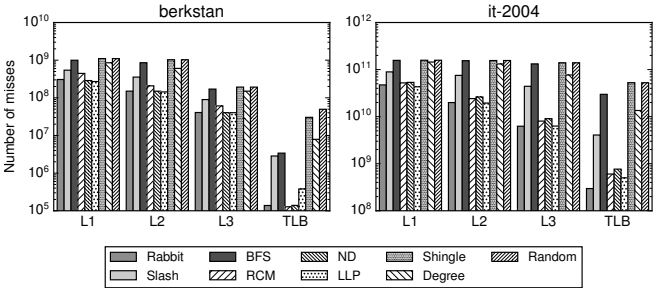


Figure 9. Number of cache misses and TLB misses

graph analysis time. Figure 7 shows the runtimes of the reordering algorithms on graphs with random ordering. Rabbit Order reorders all of the graphs in quite a short time. For example, it reorders webbase, which has billion edges, in 13.8 seconds. Degree and Shingle are lightweight as well. In contrast, LLP takes an order of magnitude longer. For example, it takes about 70 minutes to reorder webbase. This large computational cost is a serious disadvantage as regards end-to-end runtime.

Figure 8 shows the PageRank runtime derived from each ordering. Rabbit Order is better than or comparable to the state-of-the-art algorithms on all the graphs. Compared with random ordering, Rabbit Order yields a 3.37x speedup on average and up to 5.25x speedup on it-2004. These results are slightly better than those of LLP, which yields a 3.34x speedup on average (5.08x maximum). By comparison, ND, RCM, and SlashBurn trail in performance, while Degree, BFS, and Shingle fail to produce significant improvements on most graphs. Note that all of the algorithms performed poorly on twitter, likely due to its extraordinary degree distribution and complex structure [13]. Although the effectiveness of reordering inevitably depends on the graph structure, our results show that, overall, reordering is a promising way to improve performance.

Figure 9 shows the number of misses at the cache memory and the translation look-aside buffer (TLB) produced by PageRank using each ordering approach. We choose to illustrate the results for berkstan and it-2004 since they are the smallest and largest graphs that ND can reorder. The results confirm that the performance improvements derive from locality improvement and consequent reduction in cache misses. Notice that the number of misses is consistent with the PageRank performance shown in Figure 8. Specifically, Rabbit Order and LLP achieve the largest reductions in misses. Each ordering also shows a similar trend on the other graphs. In addition, the reordering algorithms reduce proportionally more misses on it-2004 than on berkstan, particularly misses at the L3 cache. This is because berkstan is so small that the L3 cache can accommodate most of the data, even if the vertices are randomly ordered. For a similar reason, reordering yields only modest performance improvements for small graphs such as enwiki and ljournal.

Table IV
MODULARITY AND PAGERANK RUNTIME YIELDED BY RABBIT ORDER

Graph	Modularity		PageRank runtime [sec.]		
	Sequential	48-thread	Sequential	48-thread	
berkstan	0.930	0.934	0.214	0.218	(+2.0%)
enwiki	0.597	0.602	3.454	3.529	(+2.2%)
ljournal	0.708	0.701	2.421	2.416	(-0.2%)
uk-2002	0.985	0.982	6.415	6.520	(+1.6%)
road-usa	0.997	0.998	2.901	2.914	(+0.5%)
uk-2005	0.979	0.981	21.833	20.719	(-5.1%)
it-2004	0.972	0.975	23.179	23.100	(-0.3%)
twitter	0.360	0.391	90.235	87.709	(-2.8%)
sk-2005	0.971	0.974	47.541	49.067	(+3.2%)
webbase	0.978	0.979	27.474	28.128	(+2.4%)

These results show that Rabbit Order is the only algorithm that yields significant end-to-end performance improvement. While Degree and Shingle have shorter reordering times than Rabbit Order, they fail to achieve high locality. Conversely, LLP matches Rabbit Order in improving locality, but it suffers large computational costs. Rabbit Order differs from these algorithms; it can not only reorder billion-edge graphs such as webbase in a dozen seconds but also offer higher locality than the state-of-the-art algorithms. Hence, Rabbit Order can improve the end-to-end performance.

C. Reordering Quality

As mentioned in Section III-B2, parallel execution of Rabbit Order may degrade community quality (i.e., modularity) compared with that of sequential execution. Because of the differences in the extracted communities, parallel execution may also degrade ordering quality (i.e., speedup of graph analysis). In order to quantify the impact of parallel execution, we reorder each graph by the sequential and 48-thread executions of Rabbit Order and compare their qualities. Table IV shows the modularity of the extracted communities and PageRank runtime yielded by the resulting ordering of each execution. The rightmost column shows the percentage change in runtime relative to sequential execution. Note that as in the other experiments, PageRank is computed using 48 threads; 'Sequential' and '48-thread' of the PageRank runtime denote the numbers of threads used by Rabbit Order. The results indicate that parallel execution makes little difference. The modularity yielded by the 48-thread execution matches or exceeds that yielded by sequential execution. Similarly, the orderings produced by the 48-thread execution preserve their effectiveness on PageRank. From a practical point of view, these results justify our parallelization approach that allows a difference in the resulting ordering to maximize reordering performance.

D. Scalability

We evaluate the scalability of Rabbit Order offered by our parallelization approach. Figure 10 shows the average speedup versus the number of threads for all graphs. SlashBurn is omitted from the figure since it is a sequential

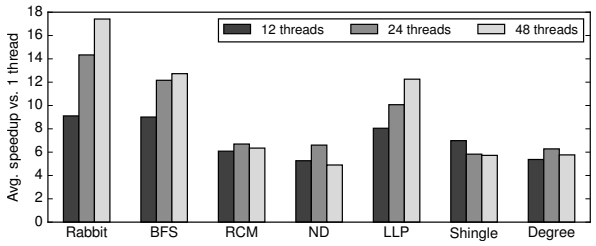


Figure 10. Reordering speedup yielded by parallel executions. The y-axis shows speedups averaged over all the graphs.

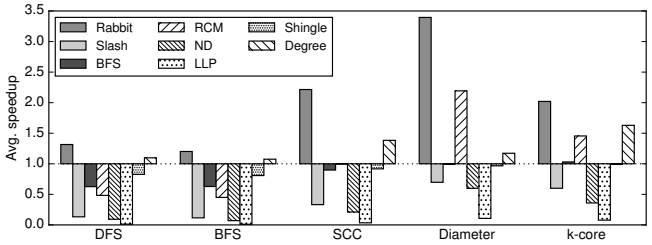


Figure 11. End-to-end performance relative to that yielded by random ordering. The y-axis shows average speedup on all the graphs.

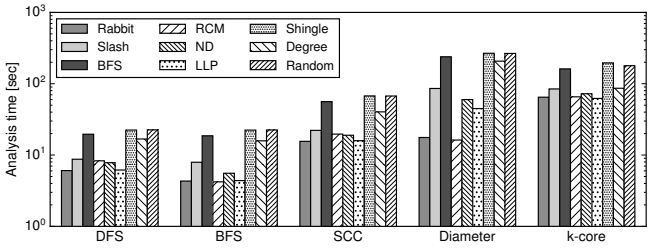


Figure 12. Analysis time of each algorithm on it-2004.

algorithm. The 12, 24, and 48-thread executions utilize 12 physical cores on a single socket, 24 physical cores on two sockets, and 24 physical cores and Hyper-Threading, respectively. As shown in the figure, Rabbit Order achieves the highest speedup (17.4x) for 48-thread execution. BFS and LLP yield the second- and third-best speedups (about 12x). The other algorithms show little difference in scalability. Rabbit Order does not only have plenty of parallelism within incremental aggregation, but also utilizes lightweight atomic operations instead of the locking mechanism. Hence, Rabbit Order can efficiently exploit parallelism in incremental aggregation and shows high scalability.

E. Other Analysis Algorithms

While this paper mainly focuses on SpMV-based graph analysis, it has been shown that reordering can also improve the performance of other analyses [2], [11]. In this subsection, we investigate the effectiveness of reordering on other memory access patterns by using five popular algorithms: DFS, BFS, strongly connected components (SCC), pseudo diameter, and k-core decomposition. We use DFS, BFS, and SCC in the Boost Graph Library [27] and implement algorithms of pseudo diameter and k-core decomposition in the literatures [28], [29]. SCC and pseudo diameter respectively use DFS and BFS as building blocks [27], [28]. All the algorithms are sequential.

Figure 11 shows the end-to-end performance improvement of each reordering algorithm compared with random ordering. The y-axis shows the average speedups for each graph shown in Table II. For all the analysis algorithms, Rabbit Order provides the best speedups among the reordering algorithms. Here, let us point out two aspects of the results. First, whereas Rabbit Order shows large speedups of 2.02–3.39x for SCC, pseudo diameter, and k-core decomposition, DFS and BFS give relatively small speedups (1.32x and 1.20x). This is because DFS and BFS are quite lightweight compared with the other analysis algorithms. Figure 12 shows the analysis times of each algorithm on the it-2004 graph. The analysis times of DFS and BFS are significantly short. Since it is more difficult to amortize reordering time by shorter analysis time, the reordering gives little performance improvement for lightweight analysis algorithms. Second, the analysis times of each ordering show a trend similar to the results for PageRank. As shown in Figure 8 and 12, Rabbit Order, RCM, and LLP are the best in terms of analysis performance, ND and SlashBurn follow them, and BFS, Shingle, and Degree are not so effective. Since reordering gathers data of neighboring vertices nearby in memory, this trend seems to result from the memory accesses to load and modify the data of neighboring vertices. Such accesses occur often in graph algorithms, and thus we believe that Rabbit Order would also be effective on other algorithms that are not investigated in this paper.

V. Related Work

The problem of improving locality in graphs has been studied for several decades in many fields, particularly in the high performance computing and data management communities. To better understand these studies and their properties, below we classify them into three categories and review some of the most successful methods within them.

Community-based methods: The first group encompasses community-based reordering methods, which co-locate densely connected vertices through community detection. *Nested Dissection* [30] recursively partitions a graph into two balanced subgraphs by removing a small set of vertices. *mt-metis* [4] includes a parallel implementation of Nested Dissection for shared memory systems. *Shingle* ordering [10] co-locates vertices that share many neighbors by using MinHash [31]. *Layered Label Propagation* (*LLP*) [19] extracts communities in parallel by repeatedly running Label Propagation [32] while varying parameters for community granularity. *SlashBurn* [12] splits the graph into small communities by removing *hubs*, which is a vertex that connects many communities. As shown in Figure 6, all

of the above methods cannot achieve high locality and fast reordering simultaneously.

BFS-based methods: The second group covers the BFS-based methods. *Cuthill-McKee* [33] and *Reverse Cuthill-McKee* (*RCM*) [7] are natural choices in this group. Cuthill-McKee sorts the vertices in the visit order of a BFS that traverses the neighbors of frontier vertices in increasing order of degree. RCM, a reversed-order variant of Cuthill-McKee, is known to produce better results [23]. Karantasis et al. [23] recently parallelized RCM. Frasca et al. [2] proposed *Adaptive Data Layout*, a parallel variant of RCM. These methods are more lightweight than the methods in the other groups since BFS captures only connections between vertices. However, they fail to capture the properties of real-world graphs, such as hierarchical community structures, and hence offer only modest locality improvements.

Hypergraph-based methods: The third group consists of hypergraph-based methods. They construct a hypergraph that represents temporal or spatial localities between matrix rows and/or columns as hyperedges. Orderings are produced by partitioning the hypergraph [34] or solving the travelling salesperson problem [8]. While they have a solid theoretical background showing that they improve locality, these methods entail a large time complexity. Whereas most reordering methods, including Rabbit Order, have time complexity roughly in proportion to m, the hypergraph-based methods have much larger time complexities, at least $\mathcal{O}(\frac{m^2}{n})$ [35]. Hence, they cannot be used to reorder large-scale real-world graphs such as those in our experiments.

VI. Conclusion

This paper has presented Rabbit Order, the first just-in-time reordering algorithm that can reduce the end-to-end runtime of graph analysis. Rabbit Order achieves both quick graph analysis and quick reordering through two approaches. The first approach, hierarchical community-based ordering, improves locality while appropriately mapping the hierarchical communities into the hierarchical CPU caches. The second approach, parallel incremental aggregation, efficiently extracts hierarchical communities showing high scalability. Our evaluations using billion-edge graphs and various analysis algorithms have shown that Rabbit Order significantly outperforms the state-of-the-art reordering algorithms.

References

[1] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A New Framework for Parallel Machine Learning," in *UAI*, 2010.

[2] M. Frasca, K. Madduri, and P. Raghavan, "NUMA-aware Graph Mining Techniques for Performance and Energy Efficiency," in *SC*, 2012, pp. 95:1–95:11.

[3] H. Shiokawa, Y. Fujiwara, and M. Onizuka, "Fast Algorithm for Modularity-Based Graph Clustering," in *AAAI*, 2013, pp. 1170–1176.

[4] D. LaSalle and G. Karypis, "Multi-threaded Graph Partitioning," in *IPDPS*, 2013, pp. 225–236.

[5] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in Parallel Graph Processing," *Parallel Process. Lett.*, vol. 17, no. 01, pp. 5–20, 2007.

[6] J. Willcock and A. Lumsdaine, "Accelerating Sparse Matrix Computations via Data Compression," in *ICS*, 2006, pp. 307–316.

[7] J. A. George, "Computer Implementation of the Finite Element Method," Ph.D. dissertation, 1971.

[8] A. Pinar and M. T. Heath, "Improving Performance of Sparse Matrix-vector Multiplication," in *SC*, 1999.

[9] A. N. Langville and C. D. Meyer, "A Reordering for the PageRank Problem," *SIAM J. Sci. Comput.*, vol. 27, no. 6, pp. 2112–2120, 2006.

[10] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On Compressing Social Networks," in *KDD*, 2009, pp. 219–228.

[11] A. Prat-Pérez, D. Dominguez-Sal, and J. L. Larriba-Pey, "Social Based Layouts for the Increase of Locality in Graph Operations," in *DASFAA*, 2011, pp. 558–569.

[12] Y. Lim, U. Kang, and C. Faloutsos, "SlashBurn: Graph Compression and Mining beyond Caveman Communities," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 12, pp. 3077–3089, 2014.

[13] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" in *WWW*, 2010, pp. 591–600.

[14] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu, "Automatic Multimedia Cross-modal Correlation Discovery," in *KDD*, 2004, pp. 653–658.

[15] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf, "Learning with Local and Global Consistency," in *NIPS*, 2003.

[16] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 22, no. 8, pp. 888–905, 2000.

[17] "Stanford Large Network Dataset Collection," http://snap.stanford.edu/data/index.html.

[18] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, p. 026113, 2004.

[19] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks," in *WWW*, 2011, pp. 587–596.

[20] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech. Theor. Exp.*, vol. 2008, no. 10, p. P10008, 2008.

[21] "Laboratory for Web Algorithmics," http://law.di.unimi.it/.

[22] D. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner, "Benchmarking for Graph Clustering and Partitioning," in *Encyclopedia of Social Network Analysis and Mining*, 2014, pp. 73–82.

[23] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali, "Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction," in *SC*, 2014, pp. 921–932.

[24] D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," in *SOSP*, 2013, pp. 456–471.

[25] "Karypis Lab," http://glaros.dtc.umn.edu/gkhome/.

[26] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," *Parallel Comput.*, vol. 35, no. 3, pp. 178–194, 2009.

[27] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, 2002.

[28] G. K. Kumfert, "An Object-Oriented Algorithmic Laboratory for Ordering Sparse Matrices," Ph.D. dissertation, Old Dominion University, 2000.

[29] V. Batagelj and M. Zaversnik, "An O(m) Algorithm for Cores Decomposition of Networks," *CoRR*, 2003.

[30] J. A. George, "Nested Dissection of a Regular Finite Element Mesh," *SIAM J. Numer. Anal.*, vol. 10, no. 2, pp. 345–363, 1973.

[31] A. Z. Broder, "On the resemblance and containment of documents," in *SEQUECES*, 1997, pp. 21–29.

[32] U. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Phys. Rev. E*, vol. 76, no. 3, p. 036106, 2007.

[33] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," in *ACM*, 1969, pp. 157–172.

[34] K. Akbudak, E. Kayaaslan, and C. Aykanat, "Hypergraph Partitioning Based Models and Methods for Exploiting Cache Locality in Sparse Matrix-Vector Multiplication," *SIAM J. Sci. Comput.*, vol. 35, no. 3, pp. C237–C262, 2013.

[35] U. V. Catalyurek, M. Deveci, K. Kaya, and B. Ucar, "Multithreaded Clustering for Multi-level Hypergraph Partitioning," in *IPDPS*, 2012, pp. 848–859.