Vertex Reordering for Real-World Graphs and Applications: An Empirical Evaluation

Reet Barik[‡], Marco Minutoli[†], Mahantesh Halappanavar[†‡], Nathan R. Tallent[†], Ananth Kalyanaraman[‡†]

[†]Pacific Northwest National Laboratory, Richland, WA, USA;

Email: {marco.minutoli, hala, nathan.tallent}@pnnl.gov

[‡]Washington State University, Pullman, WA, USA;

Email: {reet.barik, ananth}@wsu.edu

Abstract—**Vertex reordering is a way to improve locality in graph computations. Given an input (or "natural") order, reordering aims to compute an alternate permutation of the vertices that is aimed at maximizing a locality-based objective. Given decades of research on this topic, there are tens of graph reordering schemes, and there are also several linear arrangement "gap" measures for treatment as objectives. However, a comprehensive empirical analysis of the efficacy of the ordering schemes against the different gap measures, and against real-world applications is currently lacking. In this study, we present an extensive empirical evaluation of up to 11 ordering schemes, taken from different classes of approaches, on a set of 34 real-world graphs emerging from different application domains. Our study is presented in two parts: a) a thorough comparative evaluation of the different ordering schemes on their effectiveness to optimize different linear arrangement gap measures, relevant to preserving locality; and b) extensive evaluation of the impact of the ordering schemes on two real-world, parallel graph applications, namely, community detection and influence maximization. Our studies show a significant divergence among the ordering schemes (up to $40\times$ between the best and the poor) in their effectiveness to reduce the gap measures; and a wide ranging impact of the ordering schemes on various aspects including application runtime (up to $4\times$), memory and cache use, load balancing, and parallel work and efficiency. The comparative study also helps in revealing the nuances of a parallel environment (compared to serial) on the ordering schemes and their role in optimizing applications.**

I. Introduction

A graph $G(V, E)$ is a pair of a set of vertices V representing unique entities and a set of edges representing pairwise relationship between vertices. Since this simple abstraction can capture complex relationships between entities, graph-theoretic modeling and analysis has pervaded numerous areas of science and technology enabling efficient solutions of complex problems. Execution of graph algorithms on modern computer architectures with deep memory hierarchies results in loss of performance due to an inherent lack of spatial and temporal locality of memory accesses—e.g., consider a random walk on a graph. Consequently, several efforts have been explored to accelerate graph applications. One such technique is *vertex reordering*. Reordering can be defined as the permutation of the original vertex ordering such that some desirable locality-based property can be achieved.

Several techniques for vertex reordering have been proposed in literature (§III). While some of the techniques were developed for different purposes such as reducing fill in sparse
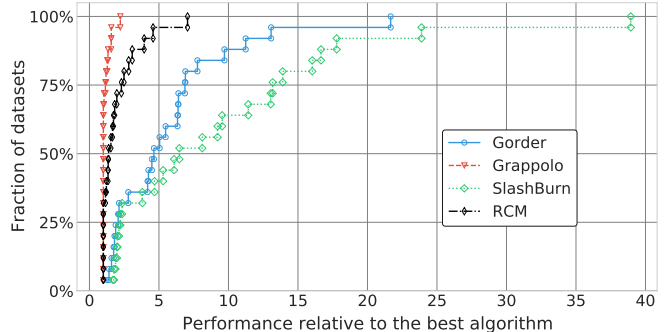

Fig. 1: *Profile of relative performance:* Relative performances of different vertex ordering schemes, as measured using the average linear arrangement gaps produced by each ordering scheme. The Y-axis represents the fraction of input problems with a total of 25 inputs. The X-axis represents the factor by which a given scheme fares relative to the best performing scheme over that fraction of inputs. For example: Gorder (blue) produces an average linear arrangement gap that is $5\times$ worse than the best performing scheme, on 50% of the inputs. The closer a curve is aligned to the Y-axis (like Grappolo), the better its relative performance.

linear algebra, or distributing work among parallel processors, recent work [1, 2, 37] has focused on improving memory performance for graph algorithms. Given the broad scope of available techniques and the objectives that they optimize for (detailed in §III), there is a lack of systematic study comparing them with each other, and their utility to improve performance of prototypical graph algorithms.

Contributions: In this paper, we present an extensive empirical evaluation of up to 11 ordering schemes, taken from different classes of approaches (§III), on a diverse set of 34 real-world graphs from different application domains. Our study is presented in two parts:

First, we present a comparative evaluation of the ordering schemes on their effectiveness to optimize various linear arrangement gap measures (§II-A), which are key indicators of locality preservation. Our results show that there is *no* one scheme that consistently outperforms others in all the metrics; instead, different schemes are better at optimizing different metrics (e.g., RCM [9] is best for graph bandwidth, whereas partitioning-based schemes perform best for average gap profiles). Figure 1, showing a sample of these results, highlights the wide range of factor differences observed, of up to $40\times$ between the best vs. poorest performing scheme.

Secondly, we present a detailed report of testing various ordering schemes on two real-world graph application use-cases, namely, community detection [13] and influence maximization [24]. Our study covers a multitude of performance aspects including impact of ordering on application runtime, quality, memory and cache use, load balancing, and parallel work and efficiency. The key findings are that the choice of ordering schemes *does* in fact matter, with the divide between the best vs. poorest performing schemes as large as $4\times$ in runtime per iteration, and $2.6\times$ in memory latency. Our study also show that this divide among schemes is *more* pronounced in a parallel (multithreaded) application than with a corresponding single threaded execution.

Our study represents one of the first to systematically characterize the ordering schemes by their effectiveness to optimize various (established) gap measures relevant to locality. Furthermore, to the best of our knowledge, this is the first work to study the impact of ordering on real-world graph applications such as community detection and influence maximization, and using parallel implementations.

II. PRELIMINARIES

Let $G = (V, E)$ denote an input graph, where V is the set of (n) vertices and E is the set of (m) edges. We use identifiers in the interval $[1, n]$ to identify the vertices. For ease of exposition, we assume undirected graphs (unless otherwise stated). Edges may be weighted and $\omega(e)$ denoting the weight of edge $e \in E$. Let $\Gamma(i)$ denote the set of neighbors of vertex i in G—i.e., $\Gamma(i) = \{j | (i, j) \in E\}$. The *degree* of vertex i is then given by $deg(i) = |\Gamma(i)|$.

A *vertex ordering* Π of V is a 1-1 mapping (bijection or permutation) of V onto a sequence or linear order. Intuitively, Π represents a rearrangement of vertices in V, $\Pi : i \to [1, n]$. The mapping $\Pi(i)$ is also referred to as the *rank* of vertex i in Π. Since the vertices in V are provided in a certain order at input time, we treat that input ordering to be the *natural ordering* of V. In other words, $\Pi = [1, 2, \ldots, n]$ for the natural order, and any other vertex (re)ordering can simply be expressed as a permutation of the natural order. Note that the overall structure of the graph remains unchanged with reordering.

A. Gap Measures for an Ordering

Here, we define a series of "gap" measures that can be used to evaluate the effectiveness of any given ordering. Intuitively, a lower value for these gap measures would correspond to a shorter separation between vertex ids that are connected by an edge in the graph. Considering most graph algorithms explore vertex neighborhoods, a net lower value for these gap measures is generally preferred out of any ordering Π, as it is a good indicator of preserving locality. In what follows, we define our gap measures (see Figure 2 for a simple example). We note here that similar measures have been developed for at least five decades in the domain of sparse linear algebra [11]. Therefore, we have built on relevant definitions while trying to adapt them to graph algorithms.
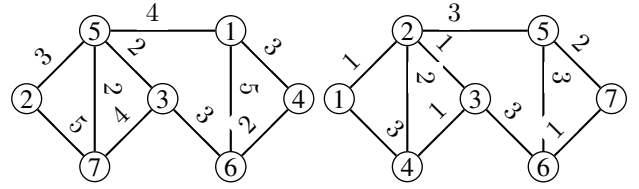

Fig. 2: An example graph with two different orderings (natural on the left, and reordered on the right) and their respective gap measures. The ordering $\Pi = [5, 1, 3, 7, 2, 6, 4]$, where vertex 1 is mapped to vertex 5, 2 to vertex 1, and so on. The gap measures for the natural order are: $\widehat{\xi} = 3.3$, $\beta = 5$, and $\widehat{\beta} = 4.43$; and for the reordered graph are: $\widehat{\xi} = 1.7$, $\beta = 3$, and and $\widehat{\beta} = 2.86$.

Given an ordering Π of V, we define the *linear arrangement gap* (or simply *gap*) between any two vertices i and j connected by an edge, to be the absolute difference between their ranks in Π. We denote it with ξ. Specifically, $\xi_\Pi(i, j)$ denotes the gap in Π between vertices i and j, for some $(i, j) \in E$:

$$\xi_\Pi(i, j) = |\Pi(i) - \Pi(j)|$$

The set of gaps of all the edges is collectively called as the *gap profile* (or simply *profile*) of G as induced by an ordering Π. We define the *average gap profile* (or the *average linear arrangement gap*) as:

$$\widehat{\xi}(G, \Pi) = \frac{1}{|E|} \sum_{(i,j) \in E} \xi_\Pi(i, j)$$

.

The *vertex bandwidth* for any vertex $i \in V$, denoted by $\beta_i(G, \Pi)$, is defined as the maximum gap between i and any of its neighbors, in Π:

$$\beta_i(G, \Pi) = \max\{\xi_\Pi(i, j) | \forall j \in \Gamma(i)\}$$

Similarly, we define the *graph bandwidth* (or the *maximum linear arrangement gap*) as the maximum vertex bandwidth:

$$\beta(G, \Pi) = \max\{\xi_\Pi(i, j) | \forall (i, j) \in E\}$$

We define the *average graph bandwidth* as:

$$\widehat{\beta}(G, \Pi) = \frac{1}{|V|} \sum_{v \in V} \beta_v(G, \Pi)$$

We note that both the bandwidth (β) and the average gap profile ($\widehat{\xi}$) of an ordering are useful indicators of performance and are used as metrics for optimization by different algorithms. For example, the Reverse Cuthill-McKee algorithm attempts to minimize the graph bandwidth, and in contrast, partitioning based methods minimize $\widehat{\xi}$ (§III).

III. VERTEX REORDERING SCHEMES

Vertex reordering has found frequent use among graph algorithm designers as a way to optimize application performance. Given an initial ordering at input (i.e., the "natural" ordering), the reordering step generates a revised permutation and subsequently all computations happen on the reordered graph data structure. Different reordering schemes use different objectives or measures to achieve their goal. In Section II-A, we presented several gap measures that can serve as objectives.

In what follows, we present a quick review of the different reordering strategies that have been used in practice. We first discuss the class of gap-based approaches. Subsequently, we discuss other classes of approaches that use different variants as objectives. Figure 3 shows a schematic organization of these different ordering schemes based on their underlying approaches.

A. Gap-based Schemes

The **Minimum Linear Arrangement (MinLA)** problem [33] formulates the problem as one of identifying a reordered permutation which minimizes the (average) linear arrangement gap. The MinLA problem is NP-Hard with its corresponding decision version being NP-Complete [14] and there have been multiple heuristics and/or approximation algorithms using techniques like simulated annealing [26, 34]. However, even these heuristics do not have efficient implementations in practice and are considered expensive in practice. There also exists a $\log$ variant of the MinLA problem [7, 35], referred to as MinLogA. This variant is motivated toward achieving graph compression [5], and is more relevant in the context of graph storage.
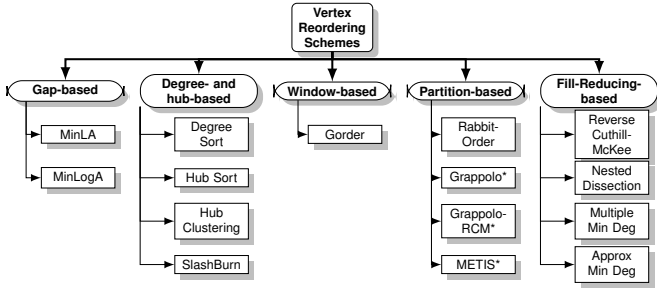

Fig. 3: A categorization of different vertex reordering methods based on key algorithmic ideas. The two Grappolo-based ordering schemes and the METIS-based ordering scheme were generated as part of this paper (by repurposing the original Grappolo and METIS tools, respectively, for ordering), in order to enable a comparative evaluation.

B. Degree- and Hub-based Schemes

Various reordering schemes use the degree information, as it is relatively easy to group/sort vertices by degrees. The simplest approach, **Degree Sort**, is to simply reorder by sorting the vertices by degree (either in non-decreasing or non-increasing order). Degree Sort is easy to implement, and light-weight in reordering cost. However, the scheme is not designed to optimize any of the gap measures.

Hub Sort [38] is a variant of Degree Sort, which sorts the vertices by non-increasing degree and then maintains a contiguous ordering of those "high" vertices (or hubs) defined based on a minimum degree cutoff. The remaining vertices maintain their relative natural order. Intuitively, this scheme is intended to improve better spatial locality among the hub nodes (which are expected to be more frequently accessed during computation).

Hub Clustering [2] is a lighter-weight variant of hub sort, which simply ensures that the hub vertices are labeled in

a contiguous manner, without necessarily maintaining their relative sorted order. Such lightweight techniques have shown to be effective in reducing cache misses for operations such as PageRank [32] and Single Source Shortest Paths [8], provided the input graph is amenable to Degree Sort reordering (satisfies certain characteristics like 'Packing Factor').

SlashBurn [21] is a heavyweight scheme which uses a hub-based approach to identify communities. In doing so, it finds an ordering of nodes such that the corresponding adjacency matrix is close to the block-diagonal form.

C. Window-based Schemes

Gorder is the current state-of-the-art in terms of minimizing cache misses is Gorder [37]. It slides a window of a certain length w over the provided input/natural order, and tries to maximize a score called *Gscore* within each window. This score is defined on every pair of vertices (i, j) that originates within a window as follows: $S(i, j) = S_s(i, j) + S_n(i, j)$, where $S_s(i, j)$ is the number of times i and j share a common neighbor, and $S_n(i, j)$ is the number of edges between i and j. This optimization problem is shown to be NP-hard [37], and the authors present an efficient approximation algorithm that runs in time proportional to the sum of square of degrees of the vertices.

D. Partitioning-based Schemes

Graph partitioners can be used as way to generate an ordering. A given vertex set V can be partitioned into to p partitions, $V = V_1 \cup V_2 \dots V_p$, with several overarching goals such as increasing the connectivity within the partitions, minimizing the connectivity across partitions, and balancing the number of vertices (and edges) across partitions. If vertices within a partition are densely connected, then reordering the vertices based on partitions can be expected to minimize the average gap of the reordered graph. In order to explore the utility of this approach, we employ graph partitioning that attempts to minimize the number of edges across partitions while assigning a roughly equal number of vertices for each partition. The number of partitions is an input parameter. Since graph partitioning is NP-Hard, several heuristics exist, including a multi-level approach where a given graph is coarsened using approximate weighted matching [18], followed by spectral methods to partition, and iterative refinements [25] employed during the un-coarsening phases.

In this paper, we use the popular tool **METIS** [22] for graph partitioning to generate a vertex ordering. We experiment with varying numbers of partitions, on a wide range of inputs, and observe that the best performance in terms of average gap reduction is achieved for a partition size of 32 (results presented in §V).

Community detection (or graph clustering) tools offer an alternative to achieving a partitioning of the vertex set [13]. Here, the output communities are such that the vertices within a cluster are more tightly-knit with intra-cluster edges, than to the rest of the graph. The key difference between graph partitioning and community detection is that the latter does not require the number of communities as a parameter, and

the communities can be of varied sizes—as dictated by the input graph's modular organization [31]. Thus, by ordering the vertices based on these communities, we can expect to reduce the average gap of a reordered graph. Although community detection through modularity optimization is NP-hard [6], several fast heuristics exist.

Rabbit-Order [1] aims to achieve high locality by mapping hierarchical community structures in the input graph to the hierarchical structure of CPU caches by vertex reordering. Communities are first detected using a modularity optimization heuristic prior to generation of the vertex order.

In order to enable scaling to parallel platforms, in this paper, we propose an alternative ordering scheme that uses the multithreaded Grappolo community detection tool [28]. Grappolo also uses modularity to detect its communities, and is a parallelization of the widely used serial Louvain method [4]. The algorithm performs multiple passes ("iterations") on the original graph before compacting it to coarser levels ("phases").

For this paper, we implemented two schemes to generate a Grappolo-induced ordering. In the first scheme, identified as "**Grappolo**" in our results, the vertices are reordered based on the communities, such that all vertices within a community are contiguously labeled, while the relative ordering of the communities is itself arbitrary.

In the second scheme, which we refer to as the **Grappolo-RCM** scheme, we first build a coarsened graph, where each community becomes a vertex and edges represent inter-cluster edges, and then we perform a Reverse Cuthill-McKee (RCM) algorithm (see §III-E) on the coarsened graph. The vertices are then reordered such that all vertices that belong to each community are contiguously labeled (similar to Grappolo), while the communities are themselves ordered based on the RCM order. The intuition is to take advantage of the multi-level hierarchical information exposed by Grappolo to achieve a relative ordering among communities.

E. Fill-Reducing Schemes

In the context of sparse matrix factorization, *fill-reducing* orderings can be loosely defined as matrix orderings that minimize the number of nonzeros (fill) in the factorized matrix after reordering [11]. Given the importance of this pre-processing step, a number of techniques have been developed for fill-reducing orderings, as well as their variants that attempt to minimize the number of floating-point operations or maximize concurrency in parallel execution. Earlier methods were based on selecting the vertices based on their degrees during factorization (or symbolic factorization) of a sparse matrix. Multiple minimum degree (MMD) and approximate minimum degree (AMD) are two such examples [17].

A graph-theoretic interpretation of matrix reordering methods can be employed for vertex reordering. Two particular schemes that we include in our work from this space, are the Reverse Cuthill-McKee (RCM) and nested dissection (ND).

The **RCM** algorithm [9] begins with a vertex of the smallest degree, v, and renumbering it to 1. The algorithm proceeds by finding all the unvisited neighbors of v and then renumbering

them in the non-decreasing order of their degrees[1]. The search continues until all the vertices have been visited and renumbered based on the order in which they are discovered and their degree. The search resumes with another unvisited vertex of the smallest current degree if there are multiple connected components. Finally, the vertex ordering is reversed. Consequently, the RCM algorithm can be viewed as one that does an interleaved breadth-first search (BFS) and depth-first search (DFS) traversal of the graph. The traversal itself can be executed in $O(|E|)$ time, with the additional cost of sorting the vertices based on their degree. While the algorithm is sensitive to the initial choice of vertices, the reverse ordering of vertices was provided to provide a better fill-reducing order [9, 16].

The **Nested Dissection (ND)** ordering is a divide-and-conquer approach for factorization by recursively finding vertex separators of small sizes and partitioning the matrices into two sets [15]. We use the nested dissection implementation of Karypis and Kumar [23] in our work. Although ND is not suited for vertex ordering, we include it our suite as a representative of methods developed for reducing fill.

F. Runtime Cost of Reordering

The different ordering schemes discussed in this section can incur different runtime costs to perform the reordering. However, since the different methods are implemented by different developers using different programming languages, we only present wall clock time for a subset of the schemes that were implemented in C/C++ programming language. These schemes (compared in Figure 4) cover most of the categories in Figure 3.

We present the results in Figure 4, as a *performance profile* for the smaller dataset of 9 larger inputs (listed in Table I). Since the profile captures relative performance to the best performing algorithm for a given input, the closer a given curve is to the Y-axis, better is the performance. For instance, in this Figure 1, we observe that Grappolo and METIS (with 32 partitions) are more expensive than Degree Sort and RCM methods presented in the figure (with the factor slowdown plotted along X-axis, compared to the best performing tool(s)); the methods show comparable performance across the entire spectrum of inputs tested.

IV. EXPERIMENTAL SETUP

Input Data: We perform an extensive empirical evaluation on a total of 34 input graphs, listed in Table I with relevant statistics. This set consists of two sets: i) a set of 25 smaller inputs, used to assess the different qualitative metrics for ordering, described in §VI-A; and ii) a set of 9 larger real-world inputs representing different application domains and used in our application-based evaluation. The inputs were collected from two sources, the Koblenz Network Collection (KONECT) provided by the Institute of Web Science and Technologies at the University of Koblenz–Landau [27], and

---

[1]Note that the Children Depth-First Search method proposed by Banerjee *et al.* [3] represents a relaxation where the renumbering of unvisited neighbors follows an arbitrary order at every level.
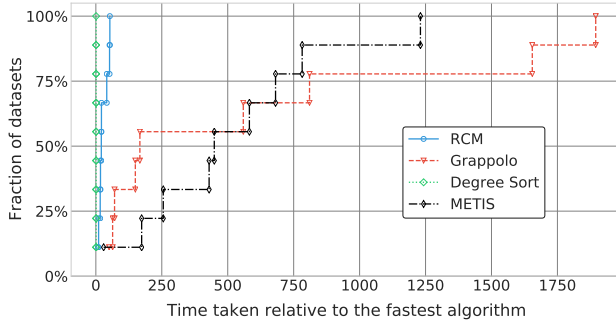
Fig. 4: *Performance profile* of compute time for four representative ordering techniques: RCM, Degree Sort, Grappolo and METIS. The Y-axis represents the fraction of problems with a total of 9 larger inputs detailed in Table I. The X-axis represents the factor by which a given scheme is slower relative to the fastest performing scheme over that fraction of inputs.

the 10th DIMACS Implementation Challenge dataset downloaded through the SuiteSparse Matrix Collection [10].

We summarize the key statistics of the inputs in Table I. While the degree distribution statistics provide insight on the overall size of the graphs, statistics such as clustering coefficient and the number of triangles provide insight on the overall connectivity of a network.

Test Platforms: We conducted all our experiments on a large shared memory server with eight Intel Xeon Platinum 8276 (Cascade Lake) CPUs and 6 TB DDR4-2933 of memory. Each CPU has 28 cores (56 threads) and a nominal frequency of 2.20 GHz, and has 12 memory channels (96 channels total), where each channel is populated (96 DIMMS, each 64 GiB). Each CPU has per-core L1 cache of 32KB; per-core L2 cache of 1 MB, and (socket-wide) L3 cache of 38.5 MB.

V. Qualitative Assessment of Ordering Schemes

We provide a comparative qualitative assessment of 11 ordering schemes. These 11 schemes cover the different classes shown in Figure 3, and include: Degree Sort, SlashBurn (from the degree- and hub-based class), Gorder (from the window-based class), Rabbit-Order, Grappolo, Grappolo-RCM, METIS (from partition-based), and RCM and ND (from the fill-reducing class). In addition, we considered the natural scheme (same as input order) and a "random" scheme (by randomly shuffling the input order).

For our assessment, we use three metrics (defined in §II-A): i) graph bandwidth or maximum gap (β); ii) average gap profile ($\widehat{\xi}$); and iii) average graph bandwidth ($\widehat{\beta}$). Together these three metrics capture the gap at a global level. We present these results in two different forms—performance plots (Figures 5, 6b, and 6a), and violin plots (Figure 8). Since we compare 11 schemes across 25 inputs, we use performance plots that concisely summarize the performance across all the inputs for different algorithms relative to the best performing algorithm for a given input. The overall best performing algorithm will be the closest curve to the Y-axis in the plot.

On the other hand, more detailed insights on the distribution of gaps are provided in the violin plots [19] for gap statistics (which subsume bandwidth statistics).

TABLE I: Summary of 25 small and 9 large instances used in the study. Columns 1 and 2 show the number of vertices and edges; Column 3 lists the maximum degree (Δ), and Column 4 lists the standard deviation of the vertex degrees.

Input	#Vertices	#Edges	Δ	Std Dev
Small Instances for Qualitative Analysis				
Chicago Road	1,467	1,298	12	2.539
Euroroad	1,174	1,417	10	1.189
Facebook (NIPS)	2,888	2,981	769	22.888
U. Rovira i Virgili	1,133	5,451	71	9.340
delaunay_n11	2,048	6,128	13	1.392
Figeys	2,239	6,452	314	17.013
US power grid	4,941	6,594	19	1.791
delaunay_n12	4,096	12,265	14	1.367
Hamster small	1,858	12,534	272	20.731
Hamster full	2,426	16,631	273	19.873
PGP	10,680	24,316	205	8.077
delaunay_n13	8,192	24,548	12	1.343
OpenFlights	2,939	30,501	473	43.216
fe_4elt2	11,143	32,819	12	0.890
Twitter lists	23,370	33,101	239	10.143
Google+	23,628	39,242	2,771	35.285
cs4	22,499	43,859	4	0.302
cti	16,840	48,233	6	0.501
delaunay_n14	16,384	49,123	16	1.348
CAIDA	26,475	53,381	2,628	33.374
Vsp	10,498	53,869	229	16.199
wing_nodal	10,937	75,489	28	2.862
Cora citation	23,166	91,500	379	11.314
Gnutella	62,586	147,892	95	5.701
arXiv astro-ph	18,771	198,050	504	30.565
Large Instances for Application Performance Analysis				
Livemocha	1.04E+05	2.19E+06	2,980	1.10E+02
California Roadnet	1.97E+06	2.77E+06	12	9.95E-01
Hyves	1.40E+06	2.78E+06	31,883	4.53E+01
arXiv hep-ph	2.81E+04	4.60E+06	11,134	5.91E+02
Youtube	3.22E+06	9.38E+06	91,751	1.28E+02
Skitter	1.70E+06	1.11E+07	35,455	1.37E+02
Actor collaborations	3.82E+05	3.31E+07	16,764	4.22E+02
LiveJournal links	5.20E+06	4.87E+07	15,016	5.06E+01
Orkut	3.07E+06	1.17E+08	33,313	1.55E+02

A. Relative Performance using Gap Measures

Based on the results presented in the form of performance plots, in Figures 5, 6b, and 6a, we make the following observations about the different schemes:

1) METIS (32 partitions[2]), Grappolo and Rabbit-Order outperform the remaining schemes in minimizing the average gap profile ($\widehat{\xi}$; Figure 5), with RCM showing competitive performance as well. In fact, we can observe four distinct performance tiers of schemes (as shown in the different color groups). The top performing group is constituted by METIS-32, Grappolo, and Rabbit-Order; followed by RCM (shown in black) which generates an average gap profile that is between roughly $1\times$–$8\times$ more than the first group for at least 50% of the inputs. The third group (shown in blue) consisting of a mixture of schemes from different categories, generates an average gap profile that is roughly between $5\times$–$25\times$ larger; and the final group (shown in green) constituting of degree-/hub-based schemes is roughly between $10\times$–$40\times$ larger. We can conclude that the partition-based schemes and

---

[2]Our choice of 32 partitions is based on our empirical evaluation as shown in the performance profile for different METIS configurations in Figure 7.
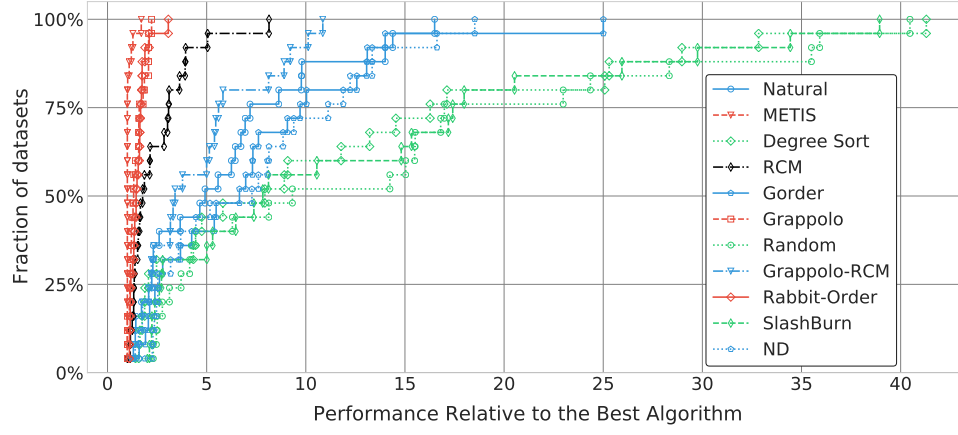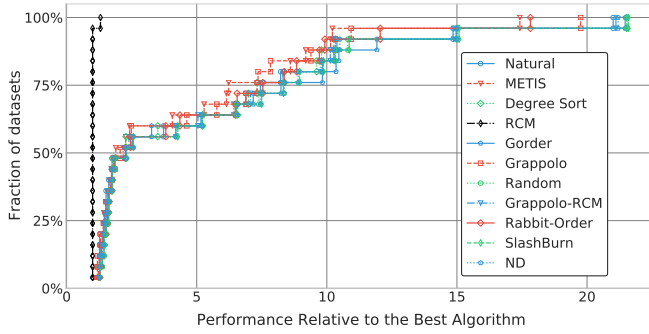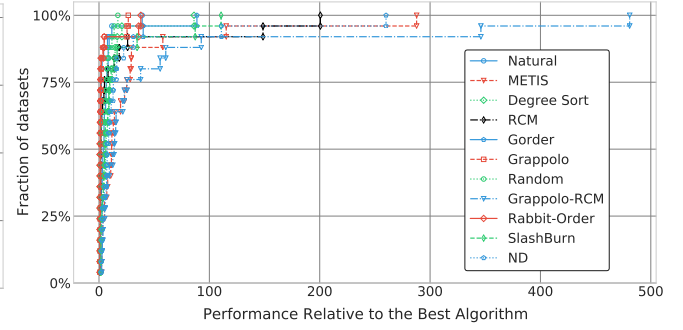
Fig. 5: *Profile of relative performance* of the average gap profile ($\widehat{\xi}$): The Y-axis represents the fraction of problems with a total of 25 inputs. The X-axis represents the factor by which a given scheme fares relative to the best performing scheme over that fraction of inputs. The closer a curve is aligned to the Y-axis the superior is its performance relative to the other schemes.

(a) Graph bandwidth (β)

(b) Average graph bandwidth ($\widehat{\beta}$)

Fig. 6: *Profile of relative performance* of graph bandwidth (left) and average graph bandwidth (right). The Y-axis represents the fraction of problems with a total of 25 inputs. The X-axis represents the factor by which a given scheme fares relative to the best performing scheme over that fraction of inputs.
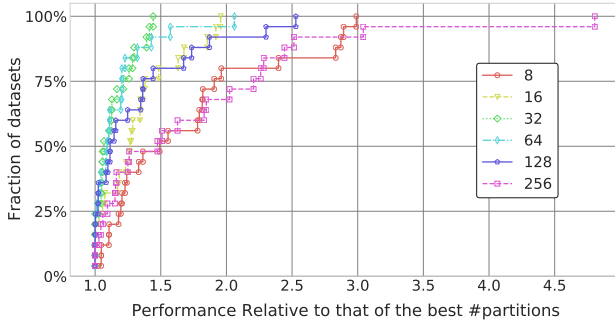

Fig. 7: *Profile of relative performance* of the average gap profile ($\widehat{\xi}(G, \Pi)$) for different number of partitions (from 8 to 256; 32 is the best) in the METIS-based ordering. The Y-axis represents the fraction of problems with a total of 25 inputs. The X-axis represents the factor by which an ordering based on a given number of partitions fares relative to the best performing one over that fraction of input.

RCM are superior to other schemes by this edge gap statistic. Notably, schemes like Gorder and Slashburn that implement sophisticated algorithms, do not necessarily yield better results than the natural and random orderings, respectively.

2) With respect to the graph bandwidth measure β, RCM clearly outperforms all other schemes in minimizing the metric (Figure 6a)—with all other schemes generating anywhere between roughly $2\times$–$22\times$ larger bandwidth. This can be expected since RCM's algorithm that orders based on an interleaved BFS/DFS traversal and its use of degree, is designed to reduce the overall bandwidth by concentrating the non-zeroes of the adjacency matrix along the diagonal. Recall that the bandwidth is the *maximum* of all gaps from any vertex to its neighbors.

3) When measuring the *average* graph bandwidth ($\widehat{\beta}$) though, there is no clear winner (Figure 6b) as most schemes yield comparable results for most inputs. This lack of divergence could be attributed to the fact that this average measure is over the number of vertices and there is typically large skew in the vertex degree distribution (as confirmed by the large standard deviations in Table I).

B. Characterizing Gap Distributions

Although global metrics provide a good insight on the quality of an ordering, the distribution of gaps for all the edges can provide a better insight. We present gap distribution
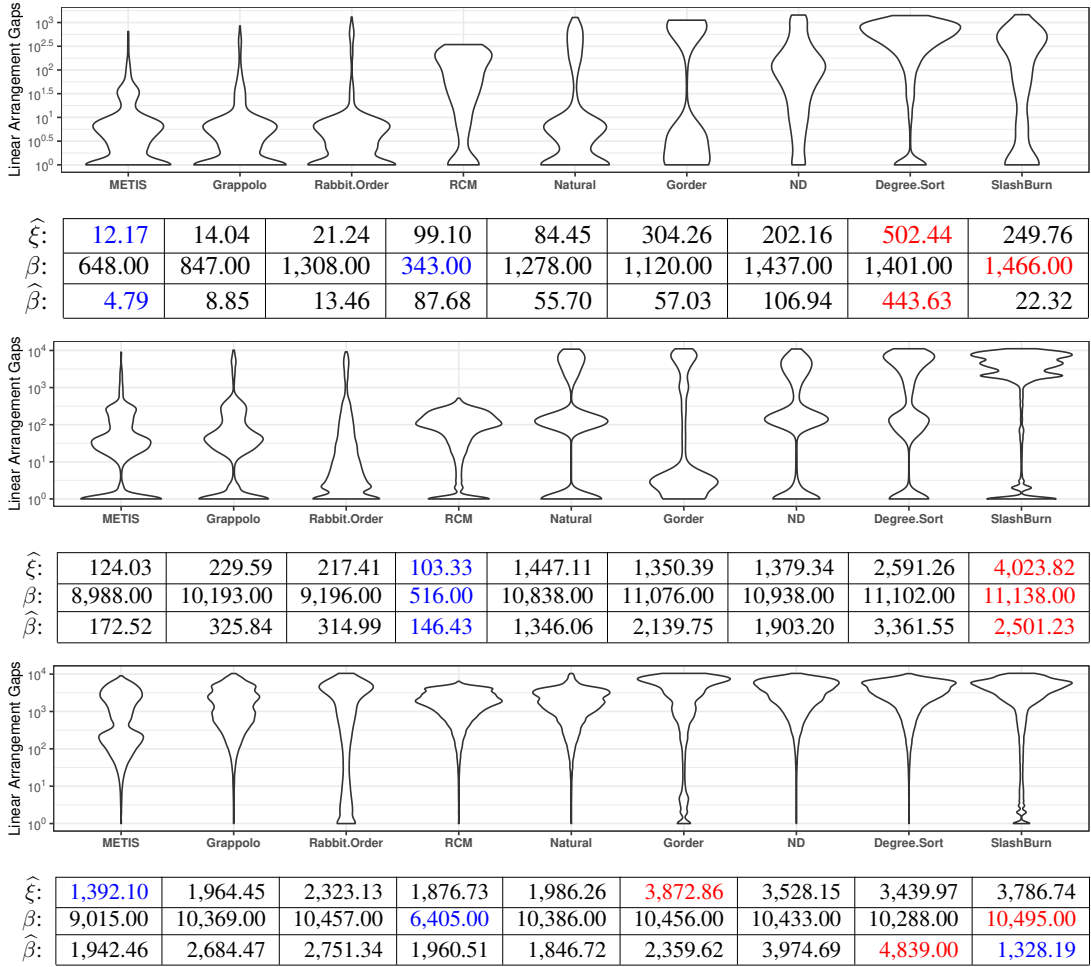
Fig. 8: *Violin plots of the gap distribution and gap metrics for different methods (blue is best, and red is worst) for* **Chicago** *(top),* **Fe_4elt2** *(middle), and* **vsp** *(bottom).*

	METIS	Grappolo	Rabbit.Order	RCM	Natural	Gorder	ND	Degree.Sort	SlashBurn
$\widehat{\xi}$:	12.17	14.04	21.24	99.10	84.45	304.26	202.16	502.44	249.76
β:	648.00	847.00	1,308.00	343.00	1,278.00	1,120.00	1,437.00	1,401.00	1,466.00
$\widehat{\beta}$:	4.79	8.85	13.46	87.68	55.70	57.03	106.94	443.63	22.32

	METIS	Grappolo	Rabbit.Order	RCM	Natural	Gorder	ND	Degree.Sort	SlashBurn
$\widehat{\xi}$:	124.03	229.59	217.41	103.33	1,447.11	1,350.39	1,379.34	2,591.26	4,023.82
β:	8,988.00	10,193.00	9,196.00	516.00	10,838.00	11,076.00	10,938.00	11,102.00	11,138.00
$\widehat{\beta}$:	172.52	325.84	314.99	146.43	1,346.06	2,139.75	1,903.20	3,361.55	2,501.23

	METIS	Grappolo	Rabbit.Order	RCM	Natural	Gorder	ND	Degree.Sort	SlashBurn
$\widehat{\xi}$:	1,392.10	1,964.45	2,323.13	1,876.73	1,986.26	3,872.86	3,528.15	3,439.97	3,786.74
β:	9,015.00	10,369.00	10,457.00	6,405.00	10,386.00	10,456.00	10,433.00	10,288.00	10,495.00
$\widehat{\beta}$:	1,942.46	2,684.47	2,751.34	1,960.51	1,846.72	2,359.62	3,974.69	4,839.00	1,328.19

results using violin plots [19] (Figure 8). Violin plots enable us to present variability in the data, not only for a given re-ordering, but also across different ordering techniques. Ridges in the figure show multimodal distribution, and the long tails characteristic of lognormal distribution show the skewedness of the gap distribution plotted along the Y-axis. We note that a normal distribution produces a smooth violin plot. From the set of 25 inputs, we pick three instances that provide three different distribution characteristics.

As illustrated in Figure 8 (top) for the input Chicago, we see striking similarity between the distributions for METIS (32), Grappolo and Rabbit-Order. The wider ridges on the lower portions indicate that a larger fraction of the gaps are small (between one and ten). In contrast, degree sort has a larger fraction on the higher side (between 100 to 1000). Unlike Chicago, the gap distribution for vsp, as illustrated in Figure 8-bottom, does not present enough perceivable differences between orderings. Consequently, we observe that for inputs like vsp, the expected performance benefits from reordering will be minimal. RCM is consistently shorter. We also present the distribution for Fe_4elt (Figure 8-middle), where Gorder and Rabbit-Order provide better orderings that

are in stark contrast to the ordering produced by SlashBurn. We also see a strong correlation of the shapes of the violin plots with the best/worst performing scheme. For $\widehat{\xi}$, we see factors of $41\times, 39\times, 28\times$ difference between the best and worst scores for Chicago, Fe_4elt and vsp respectively. Similarly, for β, we see $4\times, 22\times, 2\times$, and for $\widehat{\beta}$ a difference of $93\times, 17\times, 4\times$ respectively for the three inputs.

VI. IMPACT ON APPLICATION PERFORMANCE

We demonstrated significant qualitative differences (under gap measures) between the different ordering schemes in §V. However, an important question arises if these qualitative differences lead to predictable improvements in performance for prototypical graph algorithms. We hypothesize that because graph algorithms are memory-bound, a graph ordering that improves data locality will improve memory system performance during graph traversal. If graph traversal costs are a significant fraction of an algorithm's execution time, we expect graph ordering to improve execution performance. We set out to address this question using two prototypical graph algorithms, community detection [13] and influence maximization [24].

Note that prior works on graph orderings [2, 12] have predominantly focused on a standard suite of prototypical

graph operations such as PageRank, Single Source Shortest Paths, and Betweenness Centrality computations. Our choice to focus on community detection and influence maximization as the targets for application studies in this work, is motivated by the fact that these applications represent more advanced and complex graph operations that feature in several large-scale scientific discovery pipelines. Furthermore, they also encapsulate two very different types of graph operations. More specifically, fast community detection tools [4, 28] are a classic representative of multi-level, iterative graph methods [20]. On the other hand, the influence maximization implementations [30, 36] entail running numerous stochastic BFS over the entire graph to collect samples. In this work, we present the impact on the performance of community detection (§VI-B) and influence maximization (§VI-C), placing more emphasis on the memory access patterns and statistics, while also including correlations to gap statistics where applicable. The implementations we use [28, 30] are parallel (multithreaded) providing an added dimension to the memory performance analysis (shared L3 and DRAM).

A. Evaluation Metrics

To capture the effects of graph ordering on memory performance, use Intel VTune (2021.1.0 beta07) and report the following two metrics: (i) **Memory latency:** is measured as the average latency of loads to memory, in cycles. Although the latency of a load may be overlapped with other instructions, a longer load latency could decrease efficiency of a pipeline that can retire six instructions per cycle. (ii) **Memory hierarchy boundedness** captures the level of the memory hierarchy that bounds performance, where:
L1/L2/L3 Bound: shows cycles that CPU cores were stalled waiting on data from the private L1 cache, private L2 cache, or shared L3 cache, respectively. (Because L3 is shared, it can reflect contention from a sibling core.)
DRAM Bound: Cycles that CPU cores were stalled on DRAM (main memory) because of demand (not prefetched) loads or stores. When bounded by memory, it is preferable to be bounded by close memory levels (e.g., L1). That is, a bound of L1 (vs. DRAM) indicates better data locality, lower memory latencies, and higher effective memory bandwidth. It is also important to note that the memory bound metrics are not a decomposition—two instructions concurrently waiting on memory during a cycle count as two cycles—so their sum can be over 100%.

B. Impact on Community Detection

We use a state-of-the-art multithreaded tool, Grappolo, for community detection [28]. Since we also use this tool for generating one of the orders, we note that benefiting from such an order is not readily available. In fact, by initiating the execution from scratch, the algorithm does not benefit in any perceivable manner of a given order other than expected benefits through coalesced memory accesses.

Grappolo is compiled with Intel 19.0.5.281-20190815. We select OpenMP threads to ensure at least 2 million work units (vertices plus edges) per thread: four smallest graphs use 2

threads; next three 16, and largest two, 32. We distribute threads across sockets to provide the most L3 cache and DRAM bandwidth. For instance, with 32 threads, there are 4 threads/socket so that each thread has 3 memory channels (12 channels/socket).

Figure 9 shows the impact of vertex ordering on Grappolo's performance and output quality (modularity [31]) using heat maps. To best capture each scheme's effect, we report metrics for the first phase (or level); subsequent Grappolo phases analyze a derivative, compressed graphs that may have little relationship to the input ordering.

The first four heat maps capture runtime performance and quality. The 'Phase' heat map reports average phase runtime. Each phase conducts multiple iterations until a modularity gain threshold is met. The 'Iteration' heat map shows time per iteration. Each iteration visits all vertices (in an order determined by the parallel schedule), and for every vertex, all neighbors are accessed. Consequently, we hypothesize a correlation between iteration time and the vertex order. 'Iteration Count' heat map shows the number of iterations required. The 'Modularity' heat map shows the final output modularity (value between $[0, 1]$ and higher the better).

The final two heat maps represent different aspects of execution performance. 'Work%' shows time the CPUs spent in useful (non-synchronization) work. It measures parallel efficiency; higher values indicate less load imbalance. As an irregular (input dependent) algorithm, load balance is always a concern. Therefore, to focus metrics on graph traversals (instead of OpenMP synchronization) the final two heat maps characterize the work efficiency of Grappolo's hot routine, which inspects a vertex's neighboring communities.

The 'Work/edge' heat map shows average work, in loads, per graph edge. This metric captures the fact that while traversing a vertex's neighbors, auxiliary structures are necessary for calculating community attributes. In particular, the hot routine uses a C++ `map` to store each community's contribution to modularity. The number of loads required is data dependent.

Figure 10 shows memory performance using the metrics defined above. The figure focuses on the five largest graphs (Table I) because the working sets of the the smaller graphs may not exercise all memory levels on our test platform. For each graph, the first column shows average load latency. The subsequent columns show L1, L2, L3, DRAM Bound.
Key observations: For modularity, no ordering is clearly better, though RCM and Natural edge the others. The modularity spread is usually small, especially as it increases, which is indicative of the algorithm's popularity.

For performance metrics, there are some clear patterns. First, in terms of phase and iteration times, Grappolo usually outperforms Degree Sort, at times by factors $2\times$–$4\times$ or more. Further, there is a clear correlation between this time benefit and work metrics. The Grappolo ordering usually has the highest parallel efficiency (Work%) and lowest work per edge. It also typically has the lowest memory latency. However, comparing the three metrics, Grappolo's performance is better explained by parallel efficiency (Work%) than by memory.

Graph	Phase (s)				Iteration (s)				Iteration Count				Modularity (final)				Work% (CPU Time)				Work/edge (loads)			
	Grappolo	RCM	Natural	Degree	Grplo	RCM	Ntrl	Degr	Grplo	RCM	Ntrl	Degr	Grplo	RCM	Ntrl	Degr	Grplo	RCM	Ntrl	Degr	Grplo	RCM	Ntrl	Degr
LiveMocha	1.4	1.0	1.5	1.5	0.20	0.26	0.37	0.51	7	4	4	3	0.040	0.047	0.027	0.019	28%	24%	20%	19%	3.1	3.2	3.8	2.8
CA RoadNet	0.9	1.5	0.9	1.9	0.13	0.38	0.23	0.65	7	4	4	3	0.992	0.992	0.992	0.992	9%	12%	9%	16%	0.5	1.5	0.8	2.5
Hyves	1.6	1.5	1.7	2.4	0.39	0.39	0.44	0.60	4	4	4	4	0.608	0.731	0.722	0.715	22%	19%	18%	17%	2.5	2.5	2.1	2.1
arXiv hep-ph	3.3	3.8	4.2	5.8	0.09	0.10	0.12	0.14	36	37	36	42	0.516	0.530	0.531	0.535	50%	45%	39%	36%	0.9	1.0	0.9	0.9
YouTube	4.2	3.4	9.5	7.3	0.28	0.85	0.48	1.82	15	4	20	4	0.644	0.636	0.644	0.633	21%	12%	19%	13%	6.4	7.1	6.0	7.7
Skitter	6.3	2.9	8.1	5.8	0.14	0.95	0.21	1.45	44	3	39	4	0.842	0.833	0.840	0.827	20%	11%	15%	12%	3.2	7.0	3.4	7.3
Actor collab	7.7	12.4	8.5	29.2	0.16	0.26	0.21	0.58	49	48	41	50	0.708	0.715	0.714	0.717	32%	24%	27%	20%	2.3	2.3	2.4	2.5
LiveJournal	24.0	49.5	52.3	90.6	0.25	0.92	0.66	1.41	96	54	79	64	0.746	0.751	0.749	0.741	35%	29%	27%	30%	9.9	10.9	10.4	11.7
Orkut	70.1	131.7	52.9	131.1	0.55	1.25	0.62	2.11	128	105	85	62	0.623	0.622	0.635	0.623	38%	44%	38%	41%	8.8	10.9	9.4	11.9
	lower (redder) better				*lower (redder) better*				*lower (redder) better*				*higher (redder) better*				*higher (redder) better*				*lower (redder) better*			

Fig. 9: *Community detection:* Impact of graph ordering on performance and modularity. Each metric is represented as a row-based heat map, where 'redder' is better.

Order	YouTube					Skitter					Actor collab					LiveJournal					Skitter				
	Lat	L1	L2	L3	DRAM	Lat	L1	L2	L3	DRAM	Lat	L1	L2	L3	DRAM	Lat	L1	L2	L3	DRAM	Lat	L1	L2	L3	DRAM
Grappolo	13	9%	16%	26%	65%	9	14%	21%	15%	53%	11	13%	49%	15%	16%	19	5%	11%	15%	31%	14	8%	14%	21%	8%
RCM	34	14%	14%	26%	62%	8	25%	11%	22%	39%	12	13%	48%	17%	15%	21	8%	8%	19%	29%	23	10%	11%	26%	15%
Natural	11	10%	7%	19%	78%	11	13%	21%	12%	52%	9	15%	49%	11%	16%	25	7%	8%	14%	37%	16	8%	13%	20%	14%
Degree	16	14%	13%	18%	60%	13	18%	12%	28%	34%	13	10%	51%	18%	11%	31	8%	7%	19%	35%	20	10%	12%	24%	14%

Fig. 10: *Community detection:* Impact of graph ordering on memory metrics (§VI-A) for largest graphs. For each graph, the first column shows memory latency (cycles) as a column heat map, where 'redder' is better; subsequent columns show L1, L2, L3, DRAM Bound as a chart heat map.

Thus, the Grappolo ordering tends to result in a better load balance, at least for vertex based parallelism.

Second, RCM is often better than natural and degree. However, in contrast with the Grappolo ordering, the explanation is a combination of low iterations (better than Grappolo) and average parallel efficiency (better than Degree Sort). Conversely, the Degree Sort ordering frequently requires the fewest iterations, but each iteration takes the most time.

Third, although memory metrics provide more information than the traditional cache miss metrics, interpreting them is involved. Nominally, one expects lower memory latency to correspond to memory boundedness at lower memory levels, i.e., low DRAM values and higher values toward the left (in Figure 10). Further, one might expect lower iteration time to correlate to lower memory latency. Neither holds in all cases. For instance, Grappolo tends to be more DRAM bound than Degree, even though average memory latency is lower. There is a much clearer correlation with graph ordering than with latency, but the magnitudes are not always large.

We believe the explanation is that graph traversal costs *may not* be the dominant fraction of an algorithm's execution time. An algorithm's use of auxiliary data structures can result in additional memory access patterns that negate the benefits of vertex orderings in graph traversals. Larger graphs, as well as different graph structures, can collectively result in increased auxiliary work per edge as well as longer access costs and memory latency. Further, memory hardware itself is complex: a range of latencies can occur at the same memory level, so that equivalent boundedness metrics may not correspond to the same average memory latency. If memory latency and boundedness metrics are sometimes ambiguous, the more typical but less informative cache miss metrics can be misleading. We believe this is why memory performance is better explained by input graph rather than by graph ordering.

Finally, to contrast parallel behavior with serial, we also conducted similar experiments with a single thread execution of Grappolo (results not shown due to space). We observed that the same trends hold in the relative performance of the schemes, except that in the serial case, the magnitudes of difference between the schemes are less pronounced. More specifically, the factor of increase in the time per iteration from the best scheme (Grappolo) to the poorest scheme (Degree Sort) is between $1.3\times$–$2.5\times$.

Summary: We find that some graph orderings significantly affect both execution time (up to a factor of $4\times$) and iteration count (up to a factor of $10\times$) of Louvain-based community detection. As expected, graph ordering can be highly correlated with average memory latency. Interestingly, we also find that graph ordering can consistently improve load balance. Finally, the performance of auxiliary data structures can be more important than graph ordering.

C. Impact on Influence Maximization

Influence Maximization is the problem of selecting a small population of actors from a social network that maximize the cascading effect of a diffusion process over the network. The problem has wide applicability in studying dynamic network diffusion phenomena (e.g., disease spread).

Our evaluation uses Ripples [30], which is a scalable parallel implementation for the state-of-the-art IMM method [36]. The core computational task in Ripples is a Sampling procedure that generates a large collection of Reverse Reachability information from random vertices in the input graph by performing simulations of the targeted diffusion process. In its current implementation, Ripples supports two diffusion processes, the Independent Cascade Model (IC) and the Linear Threshold Model (LT). Among the two, the IC model has been shown to be the more computationally challenging [30, 29], and therefore our evaluation focuses on the IC model. Simulating the IC model during the Sampling procedure

Fig. 11: Impact on performance of reordering schemes on Ripples under small probability regimen. We report throughput of the Sampling procedure and the Total execution time of the application.

requires tens or hundreds of thousands of probabilistic BFS traversals. During each BFS, the neighbors j of each visited vertex i enter the new frontier with probability $p_{i,j}$ limiting consequently the portions of the graph being explored and the work available in each BFS. To mitigate the issue and increase resource utilization, Ripples implements an engine [29] that uses available CPUs to run many randomized BFS in parallel.

Ripples is compiled with GCC 9.2, and was configured to run the sampling phase on one OpenMP thread per physical core, while using 32 threads for seed selection. We distribute threads across the sockets and we used interleaved allocation of memory pages to provide the most L3 and DRAM bandwidth. We found this configuration to be the best performing on the machine. We tested with lower and higher edge probability settings. In the interest of space, we present results for a practically relevant probability setting of 0.25.

Key observations: Figure 11 shows the impact of vertex ordering on Ripples' total execution time and on the throughput of the Sampling routine. First, we can observe that these two measures have good correlations (visible as similar coloring patterns in Figure 11) and confirming the importance of Sampling on the overall performance of the application. Secondly, in terms of sampling throughput, there is a slight preference to the natural order in terms of sampling throughput for the smaller inputs (top half); whereas for the larger inputs (bottom half), more sophisticated schemes such as Grappolo and RCM start to deliver better throughput. However, we observed that these throughput improvements have a marginal impact over the total execution time.

To assess memory performance, we studied the effects of ordering on memory related performance counters by profiling

Fig. 12: *Memory performance* counters for the hotspot function in Ripples. We report Average Load Latency (LL) and how often the machine was stalled at all the layers of the memory hierarchy (L1/L2/L3/DRAM).

Ripples with Intel VTune on the skitter graph, the biggest input for which the analysis was possible. Figure 12 reports performance counters for the method generating the reverse reachability information inside the sampling method that shows up as the hot-spot for the application on the profiling data. One might expect that reordering schemes should shift the runtime profile to be more cache bound rather than memory bound and a consequent performance improvement. However, we have observed that the overall improvements on Ripples are marginal, with no particular reordering scheme standing out. Degree Sort and Grappolo based orderings show a significant improvement on the percentage of memory operation bound by the L1 cache. The expectation would be that of observing corresponding good performance on the skitter line in Figure 11. Interestingly, we observe that Degree Sort and Grappolo are at the opposite of the execution time and sampling throughput spectrum.

Summary: We find that vertex ordering schemes have marginal effects on applications that performs many BFSs in parallel. We hypothesize this observation to parallel threads competing for memory bandwidth and cache space. Even though, these results suggest a modest role for ordering in such applications, we posit that if the underlying implementation can be made locality-aware such applications can also benefit from ordering schemes.

VII. CONCLUSIONS

In this study, we presented a thorough empirical evaluation, first of its kind, to characterize and quantify the effectiveness of up to 11 vertex ordering schemes to optimize locality-relevant measures, and their impact on two important real-world graph applications. Our work provides detailed insights into the gap profiles and application memory and runtime footprints generated by the different ordering schemes—effectively indicating that the choice of ordering schemes *do* matter, more so for iterative graph applications than for applications like influence maximization; and more so in a parallel environment.

Future research directions include: application tuning to make them more locality/ordering-aware; potential use of coarsening to explore the benefits of a multiscale and/or hybrid ordering engines; and large-scale application study in

heterogeneous parallel platforms (including CPUs and GPUs) and mixed graph analytics workloads.

REFERENCES

[1] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium*, pages 22–31. IEEE, 2016.

[2] Vignesh Balaji and Brandon 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*, pages 203–214. IEEE, 2018.

[3] Jay Banerjee, Won Kim, S-J Kim, and Jorge F. Garza. Clustering a dag for cad databases. *IEEE Transactions on Software Engineering*, 14(11):1684–1699, 1988.

[4] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, oct 2008.

[5] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[6] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE transactions on knowledge and data engineering*, 20(2):172–188, 2007.

[7] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, 2009.

[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

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

[10] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011.

[11] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016.

[12] Priyank Faldu, Jeff Diamond, and Boris Grot. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization*, pages 1–13. IEEE, 2019.

[13] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[14] Michael R Garey, David S Johnson, and Larry Stockmeyer. Some simplified np-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63, 1974.

[15] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[16] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981.

[17] Alan George and Joseph W.H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[18] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *The International Journal of High Performance Computing Applications*, 26(4):413–430, 2012.

[19] Jerry L. Hintze and Ray D. Nelson. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician*, 52(2):181–184, 1998.

[20] Ananth Kalyanaraman and Partha Pratim Pande. A brief survey of algorithms, architectures, and challenges toward extreme-scale graph analytics. In *2019 Design, Automation & Test in Europe Conference & Exhibition*, pages 1307–1312. IEEE, 2019.

[21] U Kang and Christos Faloutsos. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *2011 IEEE 11th International Conference on Data Mining*, pages 300–309. IEEE, 2011.

[22] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[23] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[24] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 137–146, 2003.

[25] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[26] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*,

220(4598):671–680, 1983.

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

[28] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[29] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. cuRipples: Influence maximization on multi-GPU systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–11, 2020.

[30] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun Sathanur, Ryan Mcclure, and Jason McDermott. Fast and scalable implementations of influence maximization algorithms. In *2019 IEEE International Conference on Cluster Computing*, pages 1–12. IEEE, 2019.

[31] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

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

[33] Jordi Petit. Experiments on the minimum linear arrangement problem. *Journal of Experimental Algorithmics*, 8, 2003.

[34] Ilya Safro, Dorit Ron, and Achi Brandt. Multilevel algorithms for linear ordering problems. *Journal of Experimental Algorithmics*, 13:1–4, 2009.

[35] Ilya Safro and Boris Temkin. Multiscale approach for the network compression-friendly ordering. *Journal of Discrete Algorithms*, 9(2):190–202, 2011.

[36] Youze Tang, Yanchen Shi, and Xiaokui Xiao. Influence Maximization in Near-Linear Time: A Martingale Approach. In *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, pages 1539–1554. ACM, 2015.

[37] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828, 2016.

[38] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Matei Zaharia, and Saman Amarasinghe. Optimizing cache performance for graph analytics. *arXiv preprint arXiv:1608.01362*, 2016.