SAND2015-0201C

Distributed Graph Layout for Scalable Small-world Network Analysis

George Slota
Computer Science and Engineering
Pennsylvania State University
State College, PA
gslota@psu.edu

Kamesh Madduri
Computer Science and Engineering
Pennsylvania State University
State College, PA
madduri@cse.psu.edu

Sivasankaran Rajamanickam
Scalable Algorithms Departement
Sandia National Laboratories
Albuquerque, NM
srajama@sandia.gov

ABSTRACT

The in-memory graph layout or organization has a considerable impact on the time and energy efficiency of bulk-synchronous distributed-memory graph computations (BSP mode). It affects memory-hierarchical locality, inter-task load balance, communication time, and overall memory utilization. Graph layout could refer to partitioning or replication of vertex and edge arrays, selective replication of data structures that hold meta-data, and reordering vertex and edge identifiers. In this work, we present DGL, a fast, parallel, and memory-efficient distributed graph layout strategy that is specifically designed for small-world networks (low-diameter graphs with skewed vertex degree distributions). Label propagation based partitioning and a new scalable BFS-based ordering are the main steps in the layout strategy. We show that DGL layout can significantly improve end-to-end performance of four challenging graph analytics workloads: a parallel subgraph enumeration program, tuned implementations of breadth-first search and single-source shortest paths, and RDF3X-MPI, a distributed SPARQL query processing engine. DGL improves the total time for complex subgraph counting by 1.28× and RDF3X-MPI query times by 1.05× for 16 parts, and the communication time for BFS and SSSP by 1.48× and 1.43×, respectively, for 64 parts.

1. INTRODUCTION

Layouts of graphs and sparse matrices in distributed memory and shared memory have been well-studied for regular graphs that arise in scientific computing domain. Various partitioning methods, such as coordinate based partitioning [8,22], graph [3,31] and hypergraph partitioning [12] can be used to partition the graphs between MPI ranks. In addition to partitioning, the actual ordering of vertex identifiers can also have a significant impact on parallel performance of graph algorithms [18,24]. Cuthill-McKee (CM) [20], Reverse Cuthill-McKee (RCM), nested dissection [30], and Approximate Minimum Degree (AMD) [2] orderings are a few examples of sparse matrix reordering strategies that were used in the past. Several new open-source distributed-memory graph processing frameworks have emerged in the past few years (e.g., PowerGraph [25], Giraph [17], Trinity [41], PEGASUS [28]). The primary goal of these frameworks is to analyze web crawls and online social networks, which are low-diameter graphs with skewed vertex degree distributions. Most of these frameworks assume an initial topology-agnostic vertex and edge partitioning. With these graphs in mind, this paper attempts to answer the following questions:

1. Will the layout of the graphs impact the performance of irregular, data-analytic frameworks ?
2. Can such a layout be computed in a scalable and efficient fashion to be applicable in graph analytics ?
3. What kind of graph computations will be impacted by the graph layouts and how ?

Using traditional layout strategies based on graph/hypergraph partitioners and orderings for data layout of highly irregular small-world graphs may not be ppropriate for the following reasons:

1. Traditional partitioners and even some ordering methods, for example nested dissection, are heavyweight tools that are expensive both in terms of memory usage and time. They are appropriate when followed by even more expensive linear solvers or when they can be computed once and used for multiple solves. In contrast, graph analytic workloads are constantly evolving and each analytic operations is typically much more cheaper than a linear solver.
2. Previous ordering algorithms are designed for metrics appropriate for linear solvers such as minimizing a bandwidth [20] or minimizing the fill-in in a LU factorization [2, 30]. In contrast, ordering methods that improve the layouts in a shared memory context for small-world graphs are needed.
3. The performance of distributed-memory graph algorithms can be dependent on both local and global graph topology. Global topology affects the number of parallel phases and synchronization overhead, while local topological changes impact per-phase load balance. Optimizing for aggregate measures such as conductance or edge cut would ignore

local topology changes and may not account for dynamic variations in per-phase execution.

Graph computations on highly irregular graphs require a layout that depends on parallel partitioners and ordering methods that are highly scalable for very large graphs. Label propagation based partitioners are shown to be useful for partitioning small-world graphs [34, 44]. We utilize such partitioning algorithms to compute the distributed memory layout. Label propagation exploits the community structure inherent in many real small-world graphs to quickly partition even multi-billion edge networks. Label propagation also allows for optimization of various objectives under multiple constraints [44], which allows us to explore the impact of these bjectives and constraints on total execution and communication times for our various test analytics. In addition, we also introduce a breadth-first search based ordering that is more scalable than other ordering schemes and suitable for small-world graphs in the shared-memory layout. In case of distributed graph processing, we consider various partitioning-ordering possibilities, a simultaneous global partitioning and ordering of all vertices, and a local ordering of vertices after the partitioning phase.

In short, we propose a "distributed-memory graph layout" based on vertex partitioning using label propagation and a BFS-based parallel ordering strategy. The proposed DGL (Distributed Graph Layout) is a fast, memory-efficient, and scalable graph layout strategy. We demonstrate the new DGL layout scheme is about 10-12× times faster to compute than METIS partitioning [30], and about 2.3× faster to compute than RCM-based orderings. Throughout the paper we use the term DGL to also refer our new partitioning and ordering strategies depending on the context.

We demonstrate the impact of DGL and present detailed analysis on the end-to-end performance of four distinct graph analytic workloads. The graph analysis routines include subgraph counting, breadth-first search (BFS), single-source shortest paths (SSSP), and resource description framework (RDF) queries. The four algorithms were chosen to be representative of the diversity in modern graph analytics. We chose a recent algorithm for subgraph counting [42, 43] which is a randomized parallel algorithm to generate approximate counts of tree-structured subgraphs. Recent related work [15, 16] primarily looks at strong scaling of BFS and related computations on massive synthetic Graph 500 networks. Our work optimizes the subgraph counting algorithm, an analytic that is computationally very different from BFS. We also do an in-depth evaluation of BFS and SSSP performance. The fourth benchmark evaluates distributed-memory implementation of the popular RDF store RDF-3X [36]. We use the *end-to-end* graph analysis times for partitioning-ordering-workload in both single-threaded (MPI) and multi-threaded (MPI+OpenMP) distributed programming models. We primarily consider *real-world* rather than synthetics graphs in our study. We use tuned implementations, all developed by us, in order to ensure consistency. We also analyze trade-offs between partitioning quality on computational load balance and communication overhead for several large real-world networks. The following are the key contributions of this workload analysis.

1. A comprehensive study of the performance of the four analytics with several partitioning-ordering combinations.
2. We show that DGL layout improves subgraph counting performance by 1.28× in comparison to random partitioning. Partitioning with DGL would enable end-to-end processing (partitioning & computation) of the counts of ten vertex subgraphs on the 2 billion edge Twitter graph to complete in under fifteen minutes on 16 nodes of Blue Waters.
3. DGL layout improves the communication time of BFS and SSSP by 1.48× and 1.43× in comparison to random partitioning.
4. An informed topology-aware graph layout benefits external memory computations as well, improving the performance by 1.05× when compared to a random partitioning, as we show with RDF3X-MPI, our distributed-memory implementation of the popular RDF store RDF-3X [36].
5. A cross-analytics comparison reveals new and interesting trade-offs of communication time, load balance, and memory utilization for various graphs.

We finally mention that DGL is not inherent to the MPI processing models considered in this work, and can therefore be utilized as a preprocessing step while running under other graph engines and parallel execution environments.

2. DISTRIBUTED GRAPH LAYOUT

In this section, we discuss the distributed graph layout using label propagation based partitioning and BFS based ordering methods. We define a distributed graph layout as the pair of *partitioning*×*ordering*. The partitioning part of the layout affects the number of parallel phases and synchronization overhead in a graph computation. It is important to balance the computation in different parallel phases as well as minimize the communication overhead. We explore trade-offs in work and memory balance and communication minimization between tasks with different partitioning strategies. Work performed and memory utilization per-task roughly correlates with the number of vertices and adjacent edges stored on each task. The communication requirements roughly correlates with the number of inter-task edges, or edge cut resulting from partitioning. The ordering part of the layout affects the per-phase computation time in graph computations. We ideally want to increase intra-node memory access locality to reduce cache misses and improve execution times. In order to be practical the *partitioning*×*ordering* pair must be computed in parallel, scalable fashion.

2.1 Partitioning

We utilize three partitioners in this work. We use a random partitioning to establish a baseline for benchmarking. We use the well-known METIS [30] partitioner. We also utilize our implementation of label propagation based partitioner, which is specifically optimized to partition the small-world graphs we are considering in this work. We consider balancing partitions for vertices as well as vertices and edges. We attempt to minimize total edge cut for both DGL and METIS. Additionally, for DGL, we also attempt to balance communication among parts by minimizing the maximal number of cut edges coming out of any single part.

The DGL partitioner is based off of the community detection label propagation algorithm [38,44]. Label propagation methods are attractive as they have low computational overhead, low memory utilization, and they are easy to parallelize. Weighted label propagation is utilized in three separate stages during execution of DGL. In the first stage, we initialize data structures and create an initial partitioning of vertices into communities or clusters. The clusters are balanced and refined iteratively in the second and third stages.

Algorithm 1 DGL Multi-Constraint Multi-Objective Algorithm

```
Initialize p random partitions.
Execute degree-weighted label propagation.
for k1 iterations do
    for k2 iterations do
        Balance partitions to satisfy constraint 1.
        Refine partitions to minimize objective 1.
    for k3 iterations do
        Balance partitions to satisfy constraint 2
          and minimize objective 2.
        Refine partitions to minimize objective 1.
```

Algorithm 1 gives an overview of the DGL partitioner algorithm. Here, we are considering our constraint 1 as the maximal number of vertices per part and constraint 2 as the maximal number of edges per part. Objective 1 is the total edge cut and objective 2 is the maximal per-part edge cut. Slota et al. [44] describe a similar label propagation based algorithm and demonstrate its effectiveness in terms of cut quality and runtime with respect to other traditional partitioners. However, it is critical to show that such label propagation based partitionings are not only easy to compute, but they improve the end-to-end runtimes of graph analytic applications. With DGL we are able to utilize such a partitioner in the layout strategy and demonstrate its applicability for the first time.

2.2 Ordering

For a distributed graph computation, a good graph partitioning will reduce inter-node communication cost. The goal of on-node vertex ordering is to increase locality of intra-node memory references, and thereby reduce intra-node computation time. RCM is a commonly-used vertex ordering strategy in sparse matrix and graph applications. We propose a BFS-based ordering (see Algorithm 2) which can be considered an approximation to RCM. It avoids the costly sorting step used in RCM where it tries to order the nodes with the same parent in terms of the degree. Recently, a similar ordering was proposed for improving the matrix-vector multiply time and bandwidth reduction [29]. The primary focus of that approach was to arrive at parallel orderings to improve the linear solver time. Our focus is to improve the graph computations' end-to-end timea. We randomly choose a minimal-degree vertex as the root r. A BFS from r adds vertices to a number of level sets L as they are visited, as with RCM. We avoid explicit sorting by assuming that each $L_{1\cdots l}$ is mostly sorted in the order of decreasing vertex degree, as there is a higher likelihood of encountering high-degree vertices sooner than later for most real world graphs. As we will show in the next section, this approach performs better than both random and RCM orderings in applications that have a high number of irregular memory accesses.

Algorithm 2 DGL BFS-based vertex ordering algorithm.

```
V_id ← DGL-order(G(V, E))
for all v ∈ V do
    V_id(v) ← v
l ← 0, r ← SelectRoot(), Q ← r
B(1 ··· n) ← 0
while Q ≠ ∅ do
    l ← l + 1
    for all v ∈ Q do
        Insert v into L_l
        for all u ∈ N(v) do
            if B(u) = 0 then
                B(u) ← 1
                Insert u into Q
m ← 0
for i = l ··· 1 do
    for j = 1 ··· |L_i| do
        V_id(L_i(j)) ← m
        m ← m + 1
```

With the five partitioning methods (random, METIS (single constraint and multi-constraint) and DGL (multiple constraint, multiple objective and multiple constraint and single objective) and three ordering methods (random, RCM and DGL) we evaluate all the combinations of *partitioning*×*ordering* pairs and demonstrate that the DGL layout with DGL partitioner and DGL based ordering performs the best in irregular graph computations.

3. PARALLEL GRAPH COMPUTATIONS

3.1 Subgraph Counting

Subgraph counting is a computationally challenging task, with the naïve approach scaling as $O(n^k)$, where n is the number of vertices in a graph and k the number of vertices in the subgraph being counted. The best known exact algorithm [23] improves the exponent by a factor of $\frac{\alpha}{3}$, where α is the exponent for fast matrix multiplication. Because of these extremely high running time bounds, recent work has focused on approximation algorithms. One such approach for counting *tree-structured* subgraphs utilizes the color-coding technique of Alon et al. [1].

Algorithm 3 Subgraph counting Fully Partitioned Counting Approach.

```
Partition subgraph S using single edge cuts
for it = 1 to Niter do
    Color G(V, E) with k colors
    for all S_i in reverse order of partitioning do
        Init Table_{i,r} for V_r (vertex partition on task r)
        for all v ∈ V_r do                    ▷ Thread-level parallelism
            for all c ∈ C_i do
                Compute all Count_{S_i,c,v}
        ⟨N, I, B⟩ ← Compress(Table_{i,r})
        Alltoallv exchange of ⟨N, I, B⟩
        Update Table_{i,r} based on information received
        for all d = 1 to NumTasks do
            N_d, I_d, B_d ← Compress(Table_{i,r})
            Send(d, N_d, I_d, B_d)
            Recv(d, N_r, I_r, B_r)
            Table_{i,r} ← N_r, I_r, B_r
    Count_r+ = Σ_v^{V_r} Σ_c^{C_T} Count_{T,c,v}
Count ← Reduce(Count_r)
Scale Count based on Niter and colorful embed prob.
```

Prior work used fast parallel implementation of color-coding subgraph counting in both shared-memory and distributed-memory environments [42, 43]. We use their approach as a

staring point and further improve on the distributed counting algorithm from [43] by fully partitioning and compressing the memory-intensive dynamic programming table and replacing MPI broadcasts with All-to-all exchanges. Fully partitioning the table decreases memory requirement across all tasks, and compressing the table during communication reduces total transfer volume along with and all-to-all exchange in lieu of broadcasts. This improves scaling and enables us to count subgraphs of 10 and 11 vertices on billion-edge networks in minutes on a modest 16 node cluster. An overview of the main subgraph counting algorithm as implemented here is given in Algorithm 3. For more detailed discussion of the stages and execution of the algorithm, please refer to [42, 43]. Our changes to the subgraph counting method are both described in Algorithm 3.

3.2 SSSP and BFS

We also assess the performance impact of layout on tuned implementations for parallel Breadth-First Search (BFS) and Single-source shortest paths (SSSP) computation in this paper. Our parallel BFS approach can take advantage of both 1D and 2D graph distributions [9–11]. We use a 1D distribution in this work, as it is easier to correlate communication time with edge cut after partitioning with a 1D distribution. Recent BFS and SSSP implementations use a 1D partitioning and direction-optimizing search [4] for work-efficient and highly scalable execution on Graph 500 test instances. For an overview of the current state-of-the-art in performance optimizations for these routines, we refer the reader to [15, 16].

We use an optimized parallel implementation [37] of the Δ-stepping algorithm [33] for parallel SSSP in this paper. Each BFS iteration and Δ-stepping phase comprise of three main steps, local discovery, all-to-all exchange, and local update. To aid adjacency queries, we use a distributed compressed sparse row representation for a graph. The distance array is also partitioned and distributed along with the distributed vertices (for Δ-stepping). In the local discovery step, both algorithms expand their frontiers by listing all corresponding adjacencies and their proposed distance based on vertices in a queue of recently-visited vertices (for BFS) or in a current bucket (for Δ-stepping). Note that BFS visits each reachable vertex only once while Δ-stepping may visit each reachable vertex multiple times before it is settled.

Once all vertices in the queue are processed or the current bucket is empty (with no more vertex reinsertions), all p tasks exchange vertices in these generated lists to make them local to the owner tasks. This step is the same for both BFS and Δ-stepping, and uses an all-to-all collective communication routine. At the end of each BFS iteration and Δ-stepping phase, each task locally updates the distance of its own vertices using the exchanged information. The update in BFS is only on unvisited vertices, while Δ-stepping updates all vertices whose distance can be decreased. Thus, the Δ-stepping algorithm performs more computation and has a higher communication complexity.

Since our goal is to analyze and evaluate the effect of graph partitioning and vertex reordering, we have not yet implemented all the optimizations in [15, 16]. However, our approach has three new optimizations: (i) A semi-sort of vertex adjacencies based on weights is used prior to execution of the algorithm. (ii) Memory-optimized queues are used to represent the bucket data structure. This decreases the algorithm memory requirement, while slightly increasing the running time. (iii) An array of all local unique adjacencies is created and locally used to track tentative distance of adjacencies. This array improves efficiency by filtering out unnecessary requests to be added in the new frontiers.

3.3 Distributed RDF Stores and SPARQL Query Processing

RDF [39] is a popular data format for storing web data sets. Informally, the RDF format specifies typed relationships between entities, and the basic record in an RDF data set is a *triple*. There are a growing number of publicly-available RDF data sets that contain billions of triples. Thus, database methodologies for storing these RDF data sets, also called triple stores [19, 40], are becoming popular. We have developed a distributed MPI-based implementation of an open-source triple store called RDF-3X [36]. Our distributed RDF store is called RDF3X-MPI.

An alternate approach to viewing an RDF data set is as a directed graph with edge types. RDF data sets can be queried using a language called SPARQL. We extend the distributed RDF store methodology of RDF-3X to the SPARQL querying phase as well. Thus our RDF3X-MPI tool has two phases, a load phase and a query phase. In the load phase, the given triple data set is partitioned into several independent files, one per task, and each task then constructs an index for helping answering SPARQL queries. It is possible to parallelize some query evaluation in a purely data-parallel manner (i.e., with no communication between tasks), provided there is sufficient replication of triples among partitions. Formally, if the triple partitions satisfy an *n-hop guarantee*, then SPARQL queries in which all pairs of join variables are at distance of less than n hops from each other can be solved without any inter-task communication [27]. So the role of graph partitioning in this application is to reorder vertices such that the number of triples that are replicated between tasks after applying an n-hop guarantee are minimized. If the number of triples that are replicated is reduced, then the database indexes are smaller, making them potentially faster to query. For this application, we study the impact of partitioning on the number of replicated triples. A smaller value of replication is desired, and further, smaller index sizes should translate to faster query times.

4. EXPERIMENTAL SETUP

We evaluate performance of our new partitioning and ordering strategy DGL and the graph analytics workload on a collection of nine large-scale low diameter graphs, listed in Table 1. LiveJournal, Orkut, and Twitter (follower network) are crawls of online social networks obtained from the SNAP Database and the Max Planck Institute for Software Systems [14, 45]. uk-2005 and sk-2005 are crawls of the United Kingdom (.uk) and Slovakian (.sk) domains performed in 2005 using UbiCrawler and downloaded from the University of Florida Sparse Matrix Collection [6, 7, 21]. WebBase is similarly a crawl obtained in 2001 by the Stanford WebBase crawler. We created the BSQM and LUBM graphs

from RDF data sets generated using the Berlin SPARQL benchmark [5] and Lehigh University Benchmark [26] generators. DBpedia was created from RDF triplies extracted from Wikipedia [35].

The Orkut graph is undirected and the remaining graphs are directed. For the web and social graphs, we preprocessed the graphs before executing BFS, SSSP, and subgraph counting. Specifically, we removed all degree-0 vertices, multi-edges, and extracted the largest (weakly) connected component. Further, edge directivity was ignored when partitioning and reordering the graphs using DGL and METIS. Table 1 lists the sizes of these nine graphs after preprocessing.

Table 1: Test graph characteristics *after* preprocessing. Graphs belong to three categories, OSN: Online social networks, WWW: Web crawl, RDF: graphs constructed from RDF data. # Vertices (n), # Edges (m), average ($davg$) and max ($dmax$) vertex degrees, and approximate diameter ($\widetilde{D}$) are listed. $B = \times 10^9$, $M = \times 10^6$, $K = \times 10^3$.

Network	Category	n	m	$davg$	$dmax$	$\widetilde{D}$	Source
LiveJournal	OSN	4.8 M	42 M	18	39 K	21	[32]
Orkut	OSN	3.1 M	117 M	76	33 K	9	[46]
Twitter	OSN	44 M	2.0 B	37	750 K	36	[14]
uk-2005	WWW	39 M	781 M	40	1.8 M	21	[7]
WebBase	WWW	113 M	844 M	15	816 K	376	[7]
sk-2005	WWW	44 M	1.6 B	73	15 M	308	[7]
BSBM	RDF	16 M	67 M	8.6	3.6 M	7	[5]
LUBM	RDF	33 M	133 M	8.1	11 M	6	[26]
DBpedia	RDF	62 M	190 M	6.1	7.3 M	7	[35]

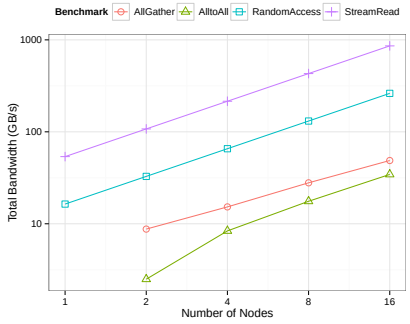

Figure 1: Bandwidth of Blue Waters for various memory and MPI benchmarks.

The scalability studies for subgraph counting, BFS, SSSP, and RDF query processing were done primarily on *Blue Waters*, a large petascale cluster at the National Center for Supercomputing Applications (NCSA). Each XE compute node of Blue Waters is a dual-socket system with 64 GB main memory and AMD 6276 Interlagos processors at 2.3 GHz. The system uses a Cray Gemini 3D torus interconnect. We built our programs with the GNU C++ compiler (version 4.8.2), using OpenMP for multithreading and the -O3 optimization parameter during compilation. For the pre-processing phases of DGL (partitioning and reordering) and some scalability runs, we utilized *Compton*, a testbed cluster. Compton has a dual socket setup with Intel Xeon E5-2670 (Sandy Bridge) CPUs at 2.60 GHz and 64 GB main memory. Due to the large memory requirements of partitioning with METIS, we also had to use the large memory nodes on *Carver* at NERSC for partitioning the larger networks (Twitter, uk-2005, Webbase, and sk-2005). Carver's large memory nodes have 1024 GB main memory and four Intel Xeon X7550 ("Nehalem-EX") CPUs at 2.00 GHz.

To give a relative sense of the intra-node data access and inter-node collective communication performance on Blue Waters, we present some memory and collective communication performance results in Figure 1 using micro benchmarks These benchmarks include AllGather and AlltoAllv MPI bandwidths and intra-node memory bandwidth sustained for regular stride-1 reads and random memory accesses, as a function of processing nodes.

5. RESULTS AND DISCUSSION

5.1 DGL Performance Evaluation

We evaluate our DGL label propagation-based partitioning methodology against METIS partitioning by examining total running time for generating 16 and 64 partitions. We consider two versions of both DGL and METIS. For DGL, we have an implementation that has *both* maximal vertex and edge balance constraints and minimizes *both* total edge cut and maximal per-part edge cut. We consider this our baseline implementation, and label it in figures as DGL-MOMC (DGL multi-objective multi-constraint). We also have a dual constraint version that only attempts to minimize the total edge cut, which we call DGL-MC. Similarly for METIS, the dual constraint single objective version is termed METIS-MC, while the single constraint (vertex balance) and single-objective version is termed simply as METIS. METIS-MC and DGL-MC are solving the same problem. However, DGL-MOMC uses the multi-constraint, multi-objective mode, instead of the single-constraint, single-objective mode of METIS. For our constraints, we fix the maximal vertex imbalance ratio at 1.10 and the edge imbalance ratio at 1.50. The results show multi-constraint, multi-objective mode is important for irregular graph computations.

Table 2 shows the partitioning time of DGL-MOMC running on Compton along with METIS-MC. Due to METIS's large memory requirements (close to 500GB for Twitter), only LiveJournal, Orkut, and the RDF graphs were partitioned on Compton. The larger web graphs and Twitter were all partitioned on Carver. We also report the relative speedup of DGL to METIS. From Table 2 we observe considerable speedup for DGL, with a geometric mean speedup of 12.4× for 16 parts and 10.1× for 64 parts.

The partitioning quality in terms of both vertex and edge balance constraints and edge cut and maximal per-part edge cut objectives for the different partitioners is shown in Table 3. In terms of the total edge cut (EC), the single-constraint, single-objective METIS does the best, but it performs poorly in the maximum per-part edge cut (EC_{max}) and edge balance (E_{max}). DGL also performs better than all the methods in the EC_{max} metric without sacrificing a lot in EC and still respecting the vertex balance and edge balance constraints. Also note the much larger E_{max} of single constraint METIS. As we will demonstrate, this can have a considerably impact of execution time for the applications in our benchmarks. Note that while METIS does better in $\#CC$, it does not affect the graph analytic appli-

Table 2: DGL-MOMC and METIS-MC partitioning time with 16-way and 64-way partitioning. DGL-MOMC uses multi-constraint multi-objective partitioning. METIS-MC uses multi-constraint single-objective partitioning.

Network	16-way partitioning			64-way partitioning		
	METIS time (s)	DGL time (s)	Speedup	METIS time (s)	DGL time (s)	Speedup
LiveJournal	75	7.4	10×	74	7.3	10×
Orkut	156	10	16×	197	13	15×
Twitter	12348	530	23×	12484	565	22×
uk-2005	255	15	17×	353	80	4.4×
WebBase	539	39	14×	551	42	13×
sk-2005	465	39	12×	514	65	7.9×
BSBM	348	28	12×	395	32	12×
LUBM	707	88	8.0×	966	123	7.9×
DBpedia	898	133	6.8×	1001	133	7.5×

Table 3: Average partitioning characteristics across all graphs. Geometric mean of vertex balance V_{max}, edge balance E_{max}, improvement over random partitioning for edge cut ratio EC and max per-part edge cut EC_{max}, and the mean improvement (decrease) in the average total number of connected components for all parts are shown. The best values for each of the last three columns are in bold font.

Partitioning	V_{max}	E_{max}	$EC(imp)$	$EC_{max}(imp)$	$\#CC(imp)$
Random	1.15	1.70	1.00	1.00	1.00
DGL-MC	1.10	1.50	5.50	2.10	72.0
DGL-MOMC	1.10	1.50	5.00	**3.18**	22.9
METIS-MC	1.10	1.50	4.40	2.16	62.1
METIS	1.10	3.88	**7.71**	2.39	**202**

cations. Traditional partitioners tend to look for fully connected components. In small-world graphs and applications that use them this does not necessarily translate into better performance.

Table 4: DGL distributed reordering time with 16-way and 64-way partitioning.

Network	16-way partitioning			64-way partitioning		
	RCM time (s)	DGL time (s)	Speedup	RCM time (s)	DGL time (s)	Speedup
LiveJournal	2.3	1.0	2.3×	2.3	1.0	2.3×
Orkut	3.9	1.9	2.1×	3.9	1.9	2.1×
Twitter	50	24	2.1×	61	29	2.1×
uk-2005	16	8.4	1.9×	17	7.6	2.2×
Webbase	33	13	2.5×	35	17	2.1×
sk-2005	24	11	2.2×	23	11	2.1×
BSBM	5.1	2.3	2.2×	4.7	2.3	2.0×
LUBM	5.7	1.7	3.4×	5.7	1.7	3.4×
DBpedia	16	6.1	2.6×	17	6.9	2.5×

We additionally compare our DGL vertex ordering strategy to RCM. Table 4 gives the average running times of both DGL and RCM in serial across all three partitioning strategies for reordering the vertices within each partition. DGL reordering results in a 2.3× average speedup compared to RCM for reordering both 16 and 64 parts. This reduction is due to the avoidance of explicit sorting required by RCM. There does not seem to be a large dependence of running times on the number of partitions, although with a greatly increased partition count for a fixed graph, it would be expected that running time decreases due to a lower diameter BFS search and overall increased cache utilization. Both these methods can be parallelized as DGL can use a parallel BFS and RCM can be implemented using the parallel version [29]. However, their timings are insignificant in the end-to-end performance of our primary subgraph counting benchmark.

5.2 Subgraph Counting Performance

We next compare the impact of various partitioning and ordering strategies with regards to the running times of our subgraph counting implementation. The performance of our subgraph counting implementation for counting a 10-vertex template and using 16 partitions across 16 nodes of Blue Waters with fixed random ordering and the five partitioning strategies, as well as the same code with DGL-MOMC partitioning and utilizing the three reordering strategies, is given in Figure 2. We also look at total end-to-end execution time for the five partitioning strategies with random ordering in terms of total time spent in the communication, computation, and partitioning steps. Note that the results with single constraint METIS for the uk-2005 and WebBase graphs are absent. This is due to execution times taking longer than 24 hours for these instances.

Several trends can be observed in Figure 2. The top subfigure gives the speedup of the communication phase of subgraph counting for each of the partitioning strategies relative to random partitioning. We note considerable speedup for all partitioners. We note our DGL methods give the best improvement for four out of the six tested graphs. Since subgraph counting is a communication-dominant application (the Twitter graph requires compression and transfer of several terabytes of data in total for the counts table exchanges between tasks), these results are significant in terms of total execution time.

The middle subfigure of Figure 2 gives the speedup relative to random ordering for the DGL and RCM reordering strategies with DGL-MOMC partitioning. We again note that DGL reordering demonstrates the highest speedup for four out of the six test instances. Note that DGL ordering can both be computed faster than RCM and can also result in better application performance. The reordering makes more noticeable impact on the larger graphs, where the importance of cache efficiency is higher, as is expected. Note that this improvement due to reordering only makes an impact when considering a pure MPI parallelization. With threading enabled, random partitioning performs optimally due to better work balance between threads. A good reordering strategy for an MPI+OpenMP parallelization model is future work.

Finally, the bottom subfigure of Figure 2 shows the total end-to-end execution times for initial partitioning plus running of the subgraph counting application. We further split subgraph counting into the sum of time spent in each of its computation and communication phases. We observe that our DGL partitioning strategies result in the fastest end-to-end running times for all test instances. The time spent for partitioning is considerable relative to execution time for METIS, as is the extra communication costs that result with random partitioning. However, we note that these results are application and runtime parameter-dependent.

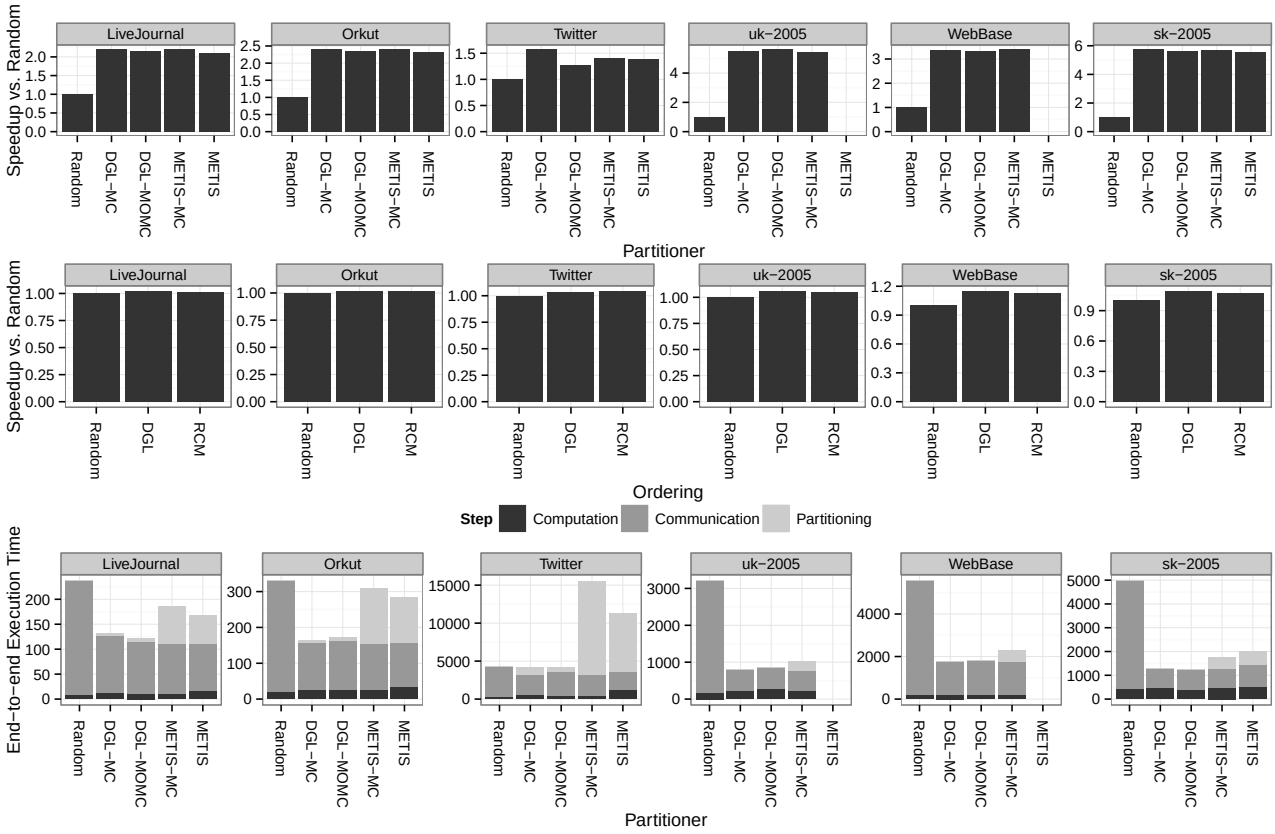

Figure 2: Speedups achieved with subgraph counting for total communication time of the various partitioning strategies relative to random partitioning, all with random ordering. Additionally, the speedups for the RCM and DGL orderings relative to random ordering with DGL multi objective partitioning. The bottom plot gives total end-to-end execution time in terms of the initial partitioning, total computation time, and total communication time.

To further visualize the performance of DGL on total execution time, we give an execution timeline in Figure 3 of a single run of counting a 10 vertex template on the LiveJournal graph. We used the Compton system for this test and random, single-constraint METIS, multi-constraint METIS, and multi-objective DGL partitioning (from left to right, respectively) with random ordering. We note first the two extreme cases. Random shows the lowest total computation times at a high cost of communication, while single objective METIS results in low communication times but high total times during the execution stages. This is due to unbalanced work among each task, which is directly proportional to the edge balance among each part. We observe that DGL-MOMC partitioning gives the best tradeoff in terms of work balance and communication requirements.

5.3 SSSP and BFS Performance

In this section, we analyze the performance of our SSSP and BFS implementation when using the different layouts. While the running time of distributed subgraph counting is dominated by large-scale data transfers during the communication phases, SSSP depends on intra-node computation for small number of tasks. Figure 4 shows the speedups for communication and computation for SSSP performance with 64 MPI tasks. The top subfigure of Figure 4 shows the communication speedups relative to random partitioning for all other partitioners with a random ordering. Due to the less complex and lower overall communication requirements for this SSSP implementation, we observe lower speedups relative to what was observed in Figure 2 with subgraph counting. However, we still observe speedups for computation times on the larger graphs with the various ordering strategies, with DGL ordering giving the best speedups on four out of the six graphs when used with DGL partitioner. The two subfigures of Figure 5 give the speedup in communication time with different partitioners and random ordering and speedups in computation time with different orderings and DGL partitioning, for the BFS implementation. We notice similar trends to SSSP in these plots.

5.4 SPARQL Query Processing

In this section, we study the impact of partitioning and ordering on the performance of RDF stores and SPARQL querying. In Table 5, we report replication ratios observed with 16-way and 64-way MPI tasking when an undirected 2-hop guarantee is enforced. The results compare DGL-MOMC with METIS-MC and random partitioning. Out of the 6 total graph-part count scenarios, DGL-MOMC approach shows the lowest replication ratio for half of them. Note that none of these partitioners are explicitly optimizing for this metric, so the performance of DGL in this instance is indirect.

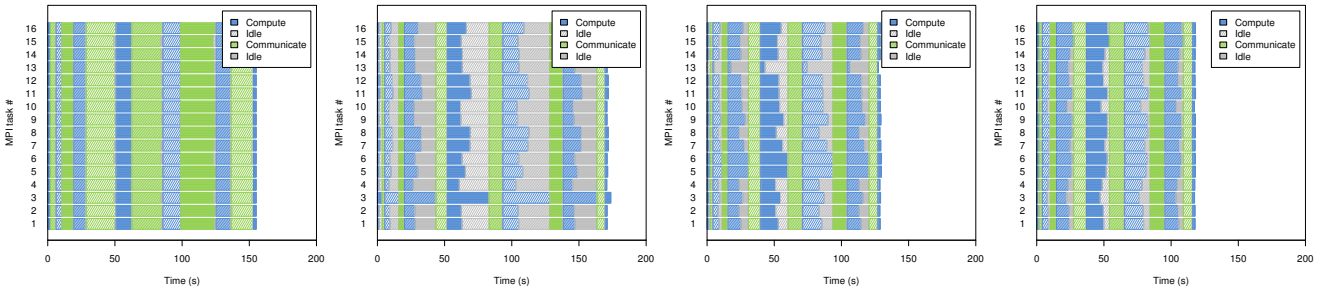

Figure 3: Subgraph counting execution timeline (single color-coding iteration with a 10-vertex template) on 16 tasks and 32 threads with (left to right) random, single and multi-constraint METIS, and DGL partitioning strategies. Random ordering was used in all cases.

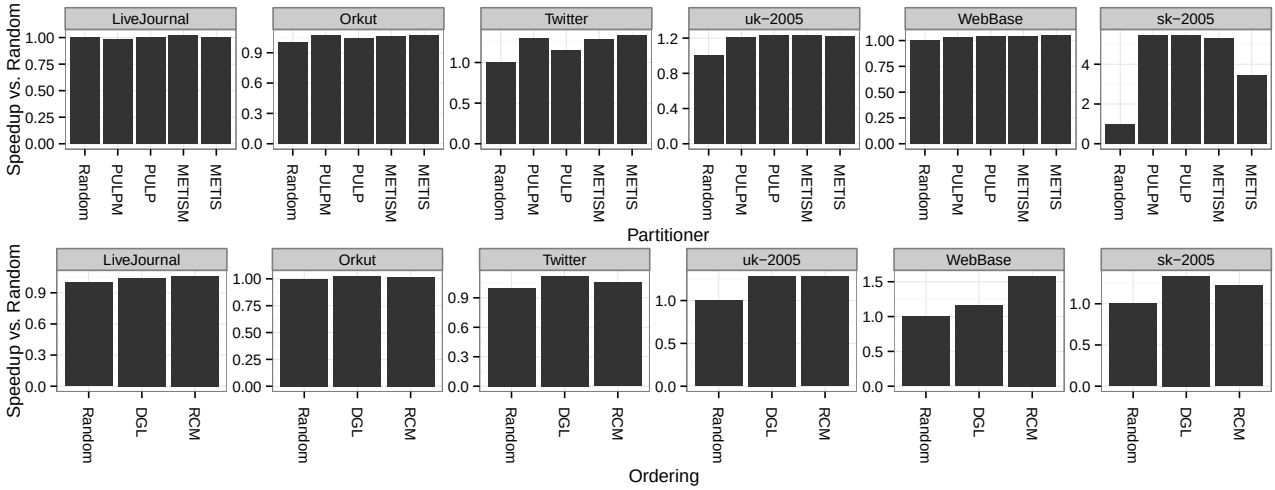

Figure 4: Communication time of SSSP implementation on 16 nodes with various partitioning options (top) and computation time of SSSP with various ordering strategies (bottom).

Table 5: Distributed RDF store replication ratios using various partitioning strategies. An undirected 2-hop guarantee is enforced. Lower values are better and best value for each graph and parts count is in bold.

Network	16-way			64-way		
	Random	DGL	METIS	Random	DGL	METIS
Berlin	5.847	**4.418**	4.795	15.482	**14.408**	19.445
LUBM	5.825	5.600	**5.041**	**16.036**	21.774	19.924
DBpedia	**2.603**	2.728	3.493	4.010	**3.320**	12.048

Table 6: Query time speedups relative to random partitioning-random ordering for the various partitioning and ordering strategies, averaged over all 3 graphs for 16 parts.

Partitioning	Ordering		
	Random	BFS	RCM
Random	1.00	0.696	0.717
DGL-MC	1.04	0.742	0.755
DGL	**1.05**	0.737	0.877
METIS-M	0.794	0.773	0.726
METIS	0.913	0.667	0.798

In Table 6, we report the speedup of query times of RDF3X-MPI averaged over the Berlin, LUBM, and DBpedia data set. The speedups are relative to the random partitioning, random ordering combination. We use the 16 part partitions for this test. DGL-MOMC partitioning with random ordering yields the best performance. We note that since DGL is faster and much more memory-efficient than METIS, this is a promising result. Future work can attempt to optimize DGL for the one and two hops replication ratio metrics, in order to further improve upon these results. Additionally, the lack of improvement with the ordering strategies is noted, and will be further investigated.

6. CONCLUSIONS

In this paper, we present DGL, a new methodology for distributed *graph layout* (partitioning, vertex ordering). The partitioning method in DGL is based on the label propagation community detection method that is scalable. The partitions produced are comparable in quality to the k-way multilevel partitioning scheme in METIS, but only take a fraction of the execution time. Our vertex layout strategies can also improve computational performance of graph computations that consist of a high proportion of irregular accesses. To conclude we answer the three questions we started with in Section 1.

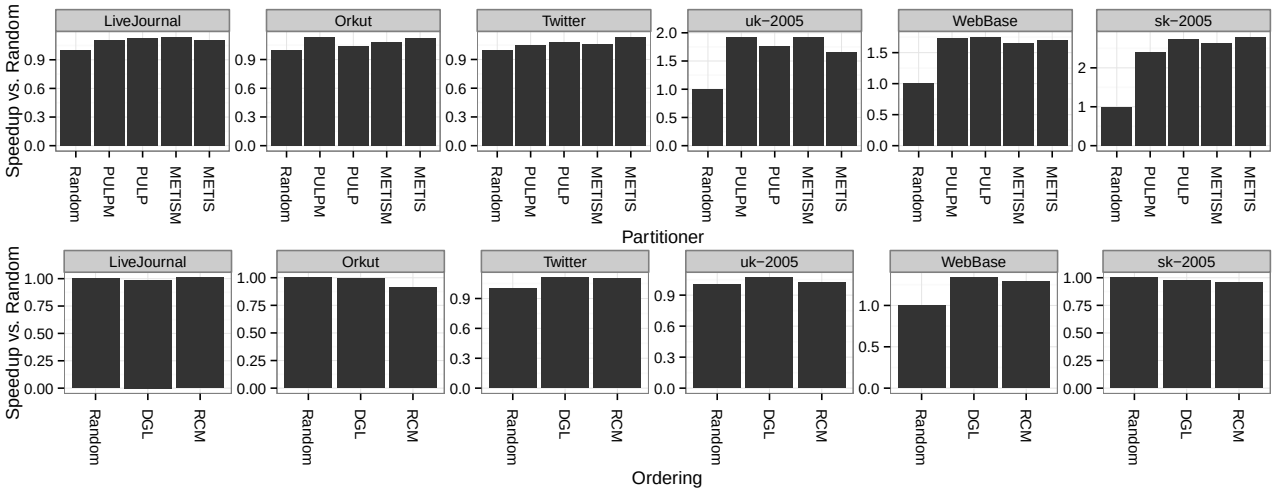

Figure 5: Communication time of BFS implementation on 16 nodes with various partitioning options (top) and computation time of BFS with various ordering strategies (bottom).

1. The layout of the graphs, both partitioning and ordering impact the performance of irregular, data-analytic frameworks considerably.
2. DGL can compute such a layout in fraction of time compared to traditional partitioners and improve end-to-end performance of graph analytics.
3. The graph computations that benefit the most are communication intensive. Simpler applications might exhibit this behavior at scale.

Acknowledgments

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work is also supported by NSF grants ACI-1253881, CCF-1439057, and the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

7. REFERENCES

[1] N. Alon, R. Yuster, and U. Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, an Approximate Minimum Degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):381–388, Sept. 2004.

[3] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. Graph partitioning and graph clustering, 10th DIMACS implementation challenge workshop. *Contemporary Mathematics*, 588, 2013.

[4] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proc. Supercomputing (SC)*, 2012.

[5] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[6] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[7] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[8] E. G. Boman, K. D. Devine, V. J. Leung, S. Rajamanickam, L. A. Riesen, M. Deveci, and U. Catalyurek. Zoltan2: Next-generation combinatorial toolkit. Technical report, Sandia National Laboratories, 2012.

[9] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 50. ACM, 2013.

[10] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[11] A. Buluç and K. Madduri. Graph partitioning for scalable distributed graph computations. In D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors, *Graph Partitioning and Graph Clustering*, chapter 6, pages 81–100. AMS, 2013.

[12] U. V. Catalyürek and C. Aykanat. Patoh: a multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara*, 6533, 1999.

[13] Ü. V. Catalyürek, M. Deveci, K. Kaya, and B. Uçar. Umpa: A multi-objective, multi-level partitioner for communication minimization. *Contemporary Mathematics*, 588, 2013.

[14] M. Cha, H. Haddadi, F. Benevenuto, and K. P.

Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Proc. Int'l. Conf. on Weblogs and Social Media (ICWSM)*, 2010.

[15] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[16] F. Checconi and F. Petrini. Traversing trillions of edges in real-time: Graph exploration on large-scale parallel machines. In *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[17] A. Ching and C. Kunz. Giraph: Large-scale graph processing infrastructure on Hadoop. *Hadoop Summit*, 6(29):2011, 2011.

[18] G. Cong and K. Makarychev. Optimizing large-scale graph analysis on multithreaded, multicore platforms. *Parallel and Distributed Processing Symposium, International*, 0:414–425, 2012.

[19] P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. T. Groth, A. Haque, A. Harth, F. L. Keppmann, D. P. Miranker, J. Sequeda, and M. Wylot. NoSQL databases for RDF: An empirical evaluation. In H. Alani, L. Kagal, A. Fokoue, P. T. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. F. Noy, C. Welty, and K. Janowicz, editors, *International Semantic Web Conference (2)*, volume 8219 of *Lecture Notes in Computer Science*, pages 310–325. Springer, 2013.

[20] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 1969 24th Nat'l. Conf.*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[21] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011.

[22] M. Deveci, S. Rajamanickam, K. Devine, and Ü. Çatalyürek. Multi-jagged: A scalable parallel spatial partitioning algorithm. *IEEE Transactions on Parallel and Distributed Systems (In revision)*, 2014.

[23] F. Eisenbrand and F. Grandoni. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science*, 326(1–3):57–67, 2004.

[24] M. Frasca, K. Madduri, and P. Raghavan. NUMA-aware graph mining techniques for performance and energy efficiency. In *Proc. Supercomputing (SC)*, 2012.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2012.

[26] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[27] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[28] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[29] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 921–932. IEEE Press, 2014.

[30] G. Karypis and V. Kumar. MeTis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. version 5.1.0. `http://glaros.dtc.umn.edu/gkhome/metis/metis/download`, last accessed Apr 2014.

[31] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[32] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[33] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algs.*, 49(1):114–152, 2003.

[34] H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning complex networks via size-constrained clustering. *CoRR*, abs/1402.3281, 2014.

[35] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. DBpedia SPARQL benchmark - performance assessment with real queries on real data. In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7031 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2011.

[36] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[37] T. Panitanarak and K. Madduri. Performance analysis of single-source shortest path algorithms on distributed-memory systems. In *Proc. SIAM Workshop on Combinatorial Scientific Computing (CSC)*, 2014.

[38] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[39] RDF primer, W3C recommendation, 2004. `http://www.w3.org/TR/rdf-primer`.

[40] S. Sakr, A. Liu, and A. G. Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM Comput. Surv.*, 46(1):11, 2013.

[41] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[42] G. Slota and K. Madduri. Fast approximate subgraph counting and enumeration. In *Proc. Int'l. Conf. on Parallel Processing (ICPP)*, 2013.

[43] G. Slota and K. Madduri. Complex network analysis using parallel approximate motif counting. In *Proc.*

IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS), 2014.

[44] G. M. Slota, K. Madduri, and S. Rajamanickam. Pulp: Scalable multi-objective multi-constraint partitioning for small-world networks. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 481–490. IEEE, 2014.

[45] Stanford large network dataset collection. `http://snap.stanford.edu/data/index.html`, last accessed Apr 2014.

[46] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proc. 12th IEEE Int'l. Conf. on Data Mining (ICDM)*, pages 745–754, 2012.