PuLP: Scalable Multi-Objective Multi-Constraint Partitioning for Small-World Networks

George M. Slota Kamesh Madduri
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA, USA
Email: {gms5016, madduri}@cse.psu.edu

Sivasankaran Rajamanickam
Scalable Algorithms Department
Sandia National Laboratories
Albuquerque, NM
Email: srajama@sandia.gov

***Abstract*—We present PuLP, a parallel and memory-efficient graph partitioning method specifically designed to partition low-diameter networks with skewed degree distributions. Graph partitioning is an important Big Data problem because it impacts the execution time and energy efficiency of graph analytics on distributed-memory platforms. Partitioning determines the in-memory layout of a graph, which affects locality, inter-task load balance, communication time, and overall memory utilization of graph analytics. A novel feature of our method PuLP (Partitioning using Label Propagation) is that it optimizes for multiple objective metrics simultaneously, while satisfying multiple partitioning constraints. Using our method, we are able to partition a web crawl with billions of edges on a single compute server in under a minute. For a collection of test graphs, we show that PuLP uses 8–$39\times$ less memory than state-of-the-art partitioners and is up to $14.5\times$ faster, on average, than alternate approaches (with 16-way parallelism). We also achieve better partitioning quality results for the multi-objective scenario.**

I. Introduction

Graph analytics deals with the computational analysis of real-world graph abstractions. There are now several online repositories that host representative real-world graphs with up to billions of vertices and edges (e.g., [19], [8], [30]). Also, new open-source and commercial distributed graph processing frameworks (e.g., PowerGraph [12], Giraph [7], Trinity [28], PEGASUS [14]) have emerged in the past few years. The primary goal of these frameworks is to permit in-memory or parallel analysis of massive web crawls and online social networking data. These networks are characterized by a low diameter and skewed vertex degree distributions, and are informally referred to as *small-world* or *power law* graphs. These graph processing frameworks use different I/O formats and programming models [25], [13], but all of them require an initial vertex and edge partitioning for scalability in a distributed-memory setting.

A key motivating question for this work is, how must one organize the data structures representing the graph on a cluster of multicore nodes, with each node having 32-64 GB memory? Fully replicating the data structures on each process is infeasible for massive graphs. A graph topology-agnostic partitioning will lead to severe load imbalances when processing graphs with skewed degree distributions. Two common topology-aware approaches to generate load-balanced partitions are (i) randomly permuting vertex and edge identifiers, and (ii) using a specialized graph partitioning tool. Random permutations ensure load balance, but hurt locality and inter-task communication. Graph partitioning methods attempt to maximize both locality and load balance, and optimize for aggregate measures after partitioning, such as edge cut, communication volume, and imbalance in partitions. There is a large collection of partitioning methods [1] that perform extremely well in practice for regular, structured networks. However, there are three issues that hinder use of existing graph partitioners for small-world network partitioning:

1) Traditional graph partitioners are heavyweight tools that are designed for improving performance of linear solvers. Most graph partitioning methods use multilevel approaches, and these are memory-intensive. Partitioning time is not a major consideration, as it is easy to amortize the cost of partitioning over multiple linear system solves.
2) The collection of complex network analysis routines is diverse and constantly evolving. There is no consensus on partitioning objective measures. Partitioning with multiple constraints and multiple objectives is not widely supported in the current partitioners.
3) Small-world graphs lack good vertex and edge separators [20]. This results in problems that are hard to partition the traditional way, resulting in even high-performing traditional partitioners taking hours to partition large small-world graphs.

This paper takes a fresh look at the problem of distributed graph layout and partitioning. We introduce a new partitioning method called PuLP (Partitioning using Label Propagation), and explore trade-offs in quality and partitioning overhead for a collection of real and synthetic small-world graphs. As the name suggests, PuLP is based on the label propagation community identification algorithm [26]. This algorithm belongs to the class of agglomerative clustering algorithms, generates reasonably good quality results for the community identification problem [1], is simple to implement and parallelize, and is extremely fast. One of the goals of any graph partitioning scheme is to reduce the number of inter-partition edges (or the edge cut), as it loosely correlates with the communication costs. Since communities are tightly connected vertices, co-locating vertices of a community in a partition will increase the proportion of intra-partition edges. In typical graph analytic algorithms, the number of vertices/edges in each partition represent the local work and the memory usage. In parallel graph analytics utilizing a bulk synchronous parallel (BSP) model, we also want to minimize the maximum communication (cut edges) incurred by any single partition. As a consequence, our approach also tries to impose vertex and edge balance constraints, with the goal to minimize both total edge cut and maximal per-partition edge cut.

To demonstrate the efficacy of our approach, we compare the quality of results obtained using PuLP to the multilevel k-way partitioning method in METIS [16], [15] and ParMETIS [17]. We use the multiple constraint version of both the codes [18], [27]. We also compare our code against the KaHIP [21] library, which uses label propagation within a multilevel framework. Our contributions in the paper are:

1) A fast, scalable, partitioner that is practical for partitioning small-world graphs.
2) A partitioner that handles the multiple objective and multiple constraints that are important for small-world graph analytics.
3) A performance study for a collection of seven large-scale small-world graphs (number of edges range from 42 million to 1.8 billion).

For the large networks and commonly-used quality measures (edge cut), our partitioning scheme is comparable to all the partitioners and better than them in additional objectives (maximum edge cut per partition) for a wide range of partition counts (2-128) and with fixed edge and vertex balance constraints. The main advantage of our approach is the relative efficiency improvement: for instance, to partition the 1.8 billion edge Slovakian domain (.sk) crawl [2], our approach uses 7.5× less memory and is 16× faster than METIS. PuLP takes less than a minute on a single compute node to generate 128 partitions of this graph, while satisfying the vertex and edge balance constraints.

Note that graph partitioning is frequently used as a preprocessing routine in distributed graph processing frameworks, so PuLP can be used to accelerate the execution time of graph algorithms in software such as Giraph and PowerGraph.

II. Preliminaries: Graph Distribution and Partitioning

We consider parallelizing analytics over large and sparse graphs: the numbers of vertices (n) is greater than 10 million, and the ratio of the number of edges (m) to number of vertices is less than 1000. The graph organization/layout in a distributed-memory system is characterized by the 'distribution, partitioning, ordering' triple. Given p processes or tasks, the most common distribution strategy, called 1D distribution, is to assign each task a p-way disjoint subset of vertices and their incident edges. If the p-way vertex/edge partitioning is load-balanced, the memory required per task would be $\frac{(2m+n)}{p}$ identifiers (integers). The advantages of the 1D scheme are its simplicity, memory efficiency and ease of parallel implementation of most graph computations using an 'owner computes' model. A disadvantage of 1D methods is that some collective communication routines could potentially require exchange of messages between all pairs of tasks (p^2), and this may become a bottleneck for large p.

In contrast, a *2D* distribution with $p = p_r \times p_c$ tasks results in each task being the owner of a subgraph of size $\frac{n}{p_r} \times \frac{n}{p_c}$ with disjoint edge assignments, but collective ownership of each vertex (by p_r or p_c tasks). The advantage is better load balance and collective communication phases with up to $max(p_r^2, p_c^2)$ pairwise exchanges. The disadvantages are increased complexity in algorithm design, primarily the communication steps, and increased memory usage for storing the graph (since both the row and column dimensions may be sparse).

The interaction of graph partitioning methods and 1D distributions is well-understood [1]. Recently, it has been shown that 1D graph partitioning used in a 2D distribution is effective for small-world graphs [3]. However, computing the 1D partition still remains expensive. Other 1D and 2D distributions, for instance, vertex degree-based, are also possible [23]. The current state-of-the-art in distributed-memory implementations is to adopt a graph distribution scheme, a specific partitioning method, and then organize inter-node communication around these choices. In this paper, we focus on the partitioning aspect of the aforementioned triple and use 1D distribution and natural ordering.

The partitioning problem we are interested in for graph analytic applications and is solved in PuLP can be formally described as below. Given an undirected graph $G = (V, E)$, partition V into p disjoint partitions. Let $\Pi = \{\pi_1, \ldots, \pi_p\}$ be a balanced partition such that $\forall i = 1 \ldots p$,

$$(1-\epsilon_l)\frac{|V|}{p} \quad \leq |V(\pi_i)| \quad \leq (1+\epsilon_u)\frac{|V|}{p} \tag{1}$$

$$|E(\pi_i)| \quad \leq (1+\eta_u)\frac{|E|}{p} \tag{2}$$

where ϵ_l and ϵ_u are the lower and upper vertex imbalance ratios, η_u is the upper edge imbalance ratio, $V(\pi_i)$ is the set of vertices in part π_i and $E(\pi_i)$ is the set of edges such that both its endpoints are in part π_i. We define the set of cut edges as

$$C(G,\Pi) = \{\{(u,v) \in E\} \mid \Pi(u) \neq \Pi(v)\} \tag{3}$$

$$C(G,\pi_k) = \{\{(u,v) \in C(G,\Pi)\} \mid (u \in \pi_k \vee v \in \pi_k)\} \tag{4}$$

Our partitioning problem is then to minimize the two metrics

$$EC(G,\Pi) = |C(G,\Pi)| \tag{5}$$

$$EC_{max}(G,\Pi) = \max_k |C(G,\pi_k)| \tag{6}$$

This can also be generalized for edge weights and vertex weights. PuLP can be extended to handle other metrics like the total communication volume and the maximum communication volume. In the past, multi-constraint graph partitioning with the EC objective has been implemented in METIS and ParMETIS [18], [27]. We will compare against both these methods in Section IV. Pinar et al. [24] suggested a framework for partitioning with complex objectives (but with a single constraint) that is similar to our iterative approach. More recently, there are multi-objective partitionings [31] and multi-constraint and multi-objective partitionings [4] for hypergraph partitioning. However, hypergraph methods are often much more compute-intensive than graph partitioning methods.

III. PuLP: Methodology and Algorithms

This section introduces PuLP, which is our methodology for utilizing label propagation to partition large-scale small-world graphs in a scalable manner. We will further detail how it is possible to create and vary label propagation weighting functions to create balanced partitions that minimize total edge cut (EC) and/or maximal per-partition edge cut (EC_{max}). This overall strategy can partition graphs under both single and multiple constraints as well as under single and multiple objectives. It is possible to extend this approach even further to include other objectives, e.g. communication volume, beyond those described below.

A. Label Propagation

Label propagation was originally proposed as a fast community detection algorithm. An overview of the baseline algorithm is given in Algorithm 1. We begin by randomly initializing the labels L for all vertices in the graph out of a possible p distinct labels. p can be chosen experimentally or based on some heuristic to maximize a community evaluation metric, such as modularity or conductance. Then, for a given vertex v in the set of all vertices V in a graph G, we examine for all of its neighbors u each of their labels $L(u)$. We keep track of the counts for each distinct label in C. After examining all neighbors, v updates its current label to whichever label has the maximal count in C.

Algorithm 1 Baseline label propagation algorithm.

```
procedure LABEL-PROP($G(V,E), p, I$)
    for all $v \in V$ do
        $L(v) \leftarrow$Rand($1 \cdots p$)
    $i \leftarrow 0$, $r \leftarrow 1$
    while $i < I$ and $r \neq 0$ do
        $r \leftarrow 0$
        for all $v \in V$ do
            $C(1 \cdots p) \leftarrow 0$
            for all $(v,u) \in E$ do
                $C(L(u)) \leftarrow C(L(u)) + 1$
            $x \leftarrow$ Max($C(1 \cdots p)$)
            if $x \neq L(v)$ then
                $L(v) \leftarrow x$
                $r \leftarrow r + 1$
        $i \leftarrow i + 1$
    return $L$
```

The algorithm proceeds to iterate over all V until some stopping criteria is met. This stopping criteria is usually some fixed number of iterations I, as we show, or until convergence is reached and no new updates and performed during a single iteration (number of updates r is zero). For large graphs, there is no guarantee that convergence will be reached quickly, if at all, so a fixed iteration count is usually preferred in practice. As with p, the maximal iteration count is usually determined experimentally. Since each iteration performs linear work, this results in an overall linear and efficient algorithm.

B. PuLP *Overview*

In general, label propagation methods are attractive for community detection due to their low computational overhead, low memory utilization, as well as the relative ease of parallelizability. In PuLP, we utilize weighted label propagation in three separate stages to partition an input graph. The first stage initializes data structures and creates an initial partitioning of vertices into communities or clusters. The communities serve as initial partitioning for our iterative approach in the second and third stages. In the second and the third stage, the algorithm iteratively alternates between a label-propagation based balancing step to minimize one of the objectives and a refinement step to further improve upon a given objective. Both of these stages ensure that constraints satisfied in previous steps are not violated.

The input parameters to PuLP are listed in Table I. Listed in brackets are the default values we used for partitioning the graphs for our experiments. The vertex and edge balance constraints were selected based on what might be chosen in practice for what a typical graph analytic code might use on a small-world graph. All of the iteration counts we show were determined experimentally, as they demonstrated the best trade-off between computation time and partitioning quality across our suite of test graphs for a wide range of tested values. A more in-depth quantitative study of the effects on partitioning quality with varying iteration counts for each of the stages is undoubtedly interesting, but is reserved for future work. Likewise, the sensitivity of PuLP to varying balance constraints is not examined in this paper, but will additionally make for interesting future work.

TABLE I
PuLP inputs, parameters, and subroutines.

$G(V,E)$	Input graph (undirected, unweighted)
$n = \|V\|$	Number of vertices in graph
$m = \|E\|$	Number of edges in graph
$P(1 \cdots n)$	Per-vertex partition mappings
p	Number of partitions
ϵ_l	Vertex lower balance constraint [0.75]
ϵ_u	Vertex upper balance constraint [0.1]
η_u	Edge upper balance constraint [0.5]
I_p	# of iterations in label propagation stage [3]
I_l	# of iterations in outer loop [3]
I_b	# of iterations in balanced propagation stage [5]
I_r	# of iterations in constrained refinement stage [10]
PuLP-X	PuLP subroutine for various stages

Algorithm 2 gives the overview of the three stages to create a vertex and edge-constrained partitioning that minimizes both edge cut and maximal per-part edge cut. We refer to this algorithm as PuLP Multi-Constraint Multi-Objective partitioning, or **PULP-MM** (Algorithm 2). After initializing a random partitioning, PULP-MM does an initial label propagation in PuLP-p (Algorithm 3) to get the initial community assignments. The communities are then used in an iterative stage that first balances the number of vertices in each part through weighted label propagation (PuLP-vp listed in Algorithm 4) while minimizing the edge cut and then improves the edge cut on the balanced partition through FM-refinement [9]. The next iterative stage further balances the number of edges per part while minimizing per-part edge cut through weighted label propagation (PuLP-cp listed in Algorithm 5) and then refines the achieved partitions through constrained FM-refinement (PuLP-cr as shown in Algorithm 6). More details of these stages are in the following subsections.

Algorithm 2 PuLP multi-constraint multi-objective algorithm.

```
procedure PULP-MM($G(V,E), p, I_p, I_l, I_b, I_r$)
    for all $v \in V$ do
        $P(v) \leftarrow$Rand($1 \cdots p$)
    $N(1 \cdots p) \leftarrow$ vertex counts in $P(1 \cdots p)$
    PuLP-p($G(V,E), p, P, N, I_p$)
    for $i = 1 \cdots I_l$ do
        PuLP-vp($G(V,E), p, P, N, I_b$)
        PuLP-vr($G(V,E), p, P, N, I_r$)
    $M(1 \cdots p) \leftarrow$ edge counts in $P(1 \cdots p)$
    $T(1 \cdots p) \leftarrow$ edge cut in $P(1 \cdots p)$
    $U \leftarrow$ current edge cut
    for $i = 1 \cdots I_l$ do
        PuLP-cp($G(V,E), p, P, N, M, T, U, I_b$)
        PuLP-cr($G(V,E), p, P, N, M, T, U, I_r$)
    return $P$
```

C. PuLP *Initialization and Label Propagation*

PuLP-p (Algorithm 3) randomly initializes partition assignments P for vertices and then uses this initial partitioning in label propagation. Here, in lieu of doing the standard label propagation approach of assigning to a given vertex v a label based on the maximal label count, Max($C(1 \cdots p)$), of all of its neighbors $(v, u) \in E$, we utilize an additional degree weighting by considering the size of the neighborhood of u ($|E(u)|$ in the Algorithm). A vertex v is therefore more likely to take u's label if u has a very large degree. This approach enables creation of dense clusters around the high degree vertices that are common in small world graphs. This ends up minimizing edge cut in practice by making it preferential for boundary vertices to be of smaller degree, as larger degree vertices will propagate their label to all of their neighbors in the subsequent iterations.

We use an additional minimal size constraint Min_v to prevent the size of a given part $|\pi_i|$ from becoming too small. Initial parts that are too small require a lot more iterations in the later stages to rebalance themselves. We otherwise allow clusters to naturally grow to any size and only perform vertex/edge balancing in the subsequent stages. We chose a small fixed iteration cutoff I_p in our current implementation, as we are not explicitly optimizing for a community detection measure such as modularity [10] and only want an initial partitioning of reasonable quality .

D. PuLP *Vertex Balancing and Total Edge Cut Minimization*

With the initial partitioning, the PuLP-vp (Algorithm 4) balances the vertex counts between parts to satisfy our original balance constraint. It follows the same basic outline of the initialization stage, in that it uses degree-weighted label propagation. However, there are two important changes.

Algorithm 3 PULP degree-weighted label propagation stage.

```
procedure PULP-P($G(V,E), p, P, N, I_p$)
    $Min_v \leftarrow (n/p) \times (1 - \epsilon_l)$
    $i \leftarrow 0$, $r \leftarrow 1$
    while $i < I_p$ and $r \neq 0$ do
        $r \leftarrow 0$
        for all $v \in V$ do
            $C(1 \cdots p) \leftarrow 0$
            for all $(v,u) \in E$ do
                $C(P(u)) \leftarrow C(P(u)) + |E(u)|$
            $x \leftarrow$ Max($C(1 \cdots p)$)
            if $x \neq P(v)$ and $N(P(v)) - 1 > Min_v$ then
                $P(v) \leftarrow x$
                $r \leftarrow r + 1$
        $i \leftarrow i + 1$
    return $P$
```

Algorithm 4 PULP single objective vertex-constrained label propagation stage.

```
$P \leftarrow$ PULP-vp($G(V,E), P, p, N, I_b$)
$i \leftarrow 0$, $r \leftarrow 1$
$Max_v \leftarrow (n/p) \times (1 + \epsilon_u)$
$W_v(1 \cdots p) \leftarrow$ Max($Max_v / N(1 \cdots p) - 1$,0)
while $i < I_b$ and $r \neq 0$ do
    $r \leftarrow 0$
    for all $v \in V$ do
        $C(1 \cdots p) \leftarrow 0$
        for all $(v,u) \in E$ do
            $C(P(u)) \leftarrow C(P(u)) + |E(u)|$
        for $j = 1 \cdots p$ do
            if Moving $v$ to $P_j$ violates $Max_v$ then
                $C(j) \leftarrow 0$
            else
                $C(j) \leftarrow C(j) \times W_v(j)$
        $x \leftarrow$ Max($C(1 \cdots p)$)
        if $x \neq P(v)$ then
            Update($N(P(v)), N(x)$)
            Update($W_v(P(v)), W_v(x)$)
            $P(v) \leftarrow x$
            $r \leftarrow r + 1$
    $i \leftarrow i + 1$
```

First, for any part that is overweight, i.e. the number of vertices in that current part π_q ($N(q)$ in the algorithm) is greater than our maximal Max_v, we do not allow that part to accept new vertices. Second, there is an additional weighting parameter $W_v(1 \cdots p)$ that is based on how underweight any part currently is. For a given part q, $W_v(q)$ will approach infinity as the size of that part approaches zero and will approach zero as the size of the part approaches Max_v. For part sizes above Max_v, we will consider the weight to be zero. This weighting forces larger parts to give vertices away with a preference towards the current most underweight parts. Due to the low diameter of small-world graphs, it is possible for any given vertex to quickly move between multiple parts. This stage is still degree-weighted and therefore minimizes the edge cut in the aforementioned indirect way, preferring small degree vertices on the boundary. When none of the parts are overweight and there is little difference in W_v values, this scheme will default to basic degree-weighted label propagation.

We further explicitly minimize edge cut with FM-refinement. The FM-refinement stage iteratively examines boundary vertices and passes them to a new part if it results in a lower edge cut without violating the vertex balance constraint. We do not show PULP-vr for space considerations, but it is essentially the same as Algorithm 6 (explained below). Except for PuLP-vr, we consider only Max_v as our single constraint. We perform I_l iterations of balancing and refining before moving on to the stages for other constraints and minimize other partitioning objectives. In order to create a vertex-constrained partitioning with the total edge cut minimized, the algorithm can stop after this stage. We call this PuLP Single-Constraint Single-Objective, or simply **PULP**. Also note that very simple changes to Algorithm 4 would allow us to constrain only edge balance instead of vertex balance.

E. PuLP *Edge Balancing and EC_{max} Minimization*

Once we have a vertex balanced partitioning that minimizes edge cuts, PuLP balances edges per part and minimizes per-part edge cut (Algorithm 5). The edge cut might increase because of the new objective, hence the algorithm uses a combination of both objectives with a dynamic weighting scheme to achieve a balance between the two objectives. The algorithm also ensures the vertex balance constraint will remain satisfied. The approach still uses weighted label propagation under given constraints. However, there are a number of nuances to make note of.

Initially, we do not use the edge balance constraint as it is given to us. Instead, a relaxed constraint based on the current maximal edge count across all parts Cur_{Max_e} is used to compute the edge balance weights ($W_e(1 \cdots p)$). This results in the possibility of all parts receiving more edges with the exception of the current largest, but no part will receive enough edges to become greater than Cur_{Max_e}. As the largest part can only give away vertices and edges, Cur_{Max_e} is iteratively tightened until the given edge balance constraint is met. Once we pass the threshold given by our input constraint, we fix Cur_{Max_e} to be equal to Max_e. To minimize the maximum edges cut per part, we employ a similar procedure with Cur_{Max_c} and the weightings for maximum cut edges ($W_v(1 \cdots p)$). We iteratively tighten this bound so that, although we have no assurance that the sum edge cut will decrease, we will always be decreasing the maximal edges cut per part.

We also introduce two additional dynamic weighting terms R_e and R_c that serve to shift the focus of the algorithm between hitting the Max_e constraint and minimizing Cur_{Max_c}. For every iteration of the algorithm that the Max_e constraint is not satisfied, R_e is increased by the ratio of which Cur_{Max_e} is greater than Max_e. This shifts the weighting function to give higher preference towards moving vertices to parts with low edge counts instead of attempting to minimize the edge

Algorithm 5 PULP multi-objective vertex and edge-constrained label propagation stage.

```
$P \leftarrow$ PULP-cp($G(V,E), P, p, N, M, T, U, I_b$)
$i \leftarrow 0$, $r \leftarrow 1$
$Max_v \leftarrow (n/p) \times (1+\epsilon_u)$
$Max_e \leftarrow (m/p) \times (1+\eta_u)$
$Cur_{Max_e} \leftarrow$ Max($M(1 \cdots p)$)
$Cur_{Max_c} \leftarrow$ Max($T(1 \cdots p)$)
$W_e(1 \cdots p) \leftarrow Cur_{Max_e}/M(1 \cdots p) - 1$
$W_c(1 \cdots p) \leftarrow Cur_{Max_c}/T(1 \cdots p) - 1$
$R_e \leftarrow 1$, $R_c \leftarrow 1$
while $i < I_p$ and $r \neq 0$ do
    $r \leftarrow 0$
    for all $v \in V$ do
        $C(1 \cdots p) \leftarrow 0$
        for all $(v,u) \in E$ do
            $C(P(u)) \leftarrow C(P(u)) + 1$
        for $j = 1 \cdots p$ do
            if Moving $v$ to $P_j$ violates $Max_v$, $Cur_{Max_e}$, $Cur_{Max_c}$ then
                $C(j) \leftarrow 0$
            else
                $C(j) \leftarrow C(j) \times (W_e(j) \times R_e + W_v(j) \times R_c)$
        $x \leftarrow$ Max($C(1 \cdots p)$)
        if $x \neq P(v)$ then
            $P(v) \leftarrow x$
            $r \leftarrow r + 1$
            Update($N(P(v)), N(x)$)
            Update($M(P(v)), M(x)$)
            Update($T(P(v)), T(x)$)
            Update($U$)
            Update($W_e(P(v)), W_e(x)$)
            Update($W_c(P(v)), W_c(x)$)
            Update($Cur_{Max_e}$,$Cur_{Max_c}$)
    if $Cur_{Max_e} < Max_e$ then
        $Cur_{Max_e} \leftarrow Max_e$
        $R_c \leftarrow R_c \times Cur_{Max_c}$
        $R_e \leftarrow 1$
    else
        $R_e \leftarrow R_e \times (Cur_{Max_e}/Max_e)$
        $R_c \leftarrow 1$
    $i \leftarrow i + 1$
```

cut balance. Likewise, when the edge balance constraint is satisfied, we reset R_e to one and iteratively increase R_c to now focus the algorithm on minimizing maximal per-part edge cut.

This iterative approach with different stages works much better in practice for multiple constraints, as employing two explicit constraints at the beginning is a very tough problem. The label propagation will often get stuck, unable to find any vertices that can be moved without violating either constraint. Note that we can very easily turn the problem in a multi-constraint single-objective problem by not including Cur_{Max_c} and W_c in our weighting function or constraint checks. We demonstrate this later in Section IV by running PuLP Multi-Constraint Single-Objective, or **PULP-M**. Additionally, we can instead turn the problem into a single-constraint three-objective problem by ignoring Max_e altogether and instead just attempt to further minimize both Cur_{Max_e} and Cur_{Max_c} along with total edge cut.

Algorithm 6 PULP multi-objective vertex and edge-constrained refinement stage.

```
$P \leftarrow$ PULP-cr($G(V,E), P, N, M, T, U, p$)
$i \leftarrow 0$, $r \leftarrow 1$
$Max_v \leftarrow (n/p) \times (1 + \epsilon_u)$
$Max_e \leftarrow (m/p) \times (1 + \eta_u)$
$Cur_{Max_e} \leftarrow$ Max(Max($M(1 \cdots p)$),$Max_e$)
$Cur_{Max_c} \leftarrow$ Max($T(1 \cdots p)$)
while $i < I_r$ and $r \neq 0$ do
    $r \leftarrow 0$
    for all $v \in V$ do
        $C(1 \cdots p) \leftarrow 0$
        for all $(v,u) \in E$ do
            $C(P(u)) \leftarrow C(P(u)) + 1$
        $x \leftarrow$ Max($C(1 \cdots p)$)
        if Moving $v$ to $P_x$ does not violate $Max_v$, $Cur_{Max_e}$, $Cur_{Max_c}$ then
            $P(v) \leftarrow x$
            $r \leftarrow r + 1$
            Update($N(P(v)), N(x)$)
            Update($M(P(v)), M(x)$)
            Update($T(P(v)), T(x)$)
            Update($U$)
    $i \leftarrow i + 1$
```

After the completion of Algorithm 5, we again perform a constrained FM-refinement, given by Algorithm 6. This algorithm uses the current maximal balance sizes of Max_v, Cur_{Max_e}, and Cur_{Max_c}, and we attempt to minimize the total edge cut without violating any of these current balances.

F. Algorithm Parallelization and Optimization

One of the strengths of using label propagation for partitioning is that its vertex-centric nature lends itself towards very straightforward and efficient parallelization. For all of our listed label propagation-based and refinement algorithms, we implement shared-memory parallelization over the primary outer loop of all $v \in V$. Max_v, Cur_{Max_e}, Cur_{Max_c}, R_e, and R_c as well as N, M, and T are all global values and arrays and are updated in a thread-safe manner. Each thread creates and updates its own C, W_v, W_e, and W_c arrays.

The algorithm also uses global and thread-owned queues as well as boolean *in queue* arrays to speed up label propagation through employing a queue-based approach similar to what can be used for *color propagation* [29]. This technique avoids having to examine all $v \in V$ in every iteration. Although it is possible, because of the dynamic weighting functions, that a vertex doesn't end up enqueued when it is desirable for it to change parts on a subsequent iteration, the effects of this are observed to be minimal in practice. We observe near identical quality between both our queue and non-queue implementations as well as our serial and parallel code.

IV. Results and Discussion

A. Experimental Setup

We evaluate performance of our new PuLP partitioning strategies on a collection of seven large-scale small-world graphs, listed in Table II. LiveJournal, Orkut, and Twitter (follower network) are crawls of online social networks from the SNAP Database and the Max Planck Institute for Software Systems [30], [5], [35]. sk-2005 is a crawl of the Slovakian (.sk) domain using UbiCrawler and was retrieved from the University of Florida Sparse Matrix Collection [2], [8]. WikiLinks is a crawl of links between articles within Wikipedia [19]. The DBpedia graph is a structured RDF graph generated from Wikipedia data [22]. The R-MAT graph is a randomly generated scale 24 R-MAT graph [6]. The Orkut graph is undirected and the remaining six graphs are directed. We preprocessed the graphs before running PuLP by removing directivity in edges, deleting all degree-0 vertices and multi-edges, and extracted the largest connected component. Ignoring I/O, this preprocessing required minimal computational time, only on the order of a few seconds in serial for our datasets. Table II lists the sizes and properties of these seven graphs after preprocessing.

TABLE II
TEST GRAPH CHARACTERISTICS *after* PREPROCESSING. GRAPHS BELONG TO FOUR CATEGORIES, OSN: ONLINE SOCIAL NETWORKS, WWW: WEB CRAWL, RDF: GRAPHS CONSTRUCTED FROM RDF DATA, SYN: GENERATED SYNTHETIC NETWORK. # 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	43 M	18	20 K	16	[20]
Orkut	OSN	3.1 M	117 M	76	33 K	9	[35]
R-MAT	SYN	7.7 M	133 M	35	260 K	8	[6]
DBpedia	RDF	62 M	190 M	6.1	7.3 M	8	[22]
WikiLinks	WWW	26 M	504 M	42	4.3 M	170	[19]
sk-2005	WWW	51 M	1.8 B	72	8.6 M	308	[2]
Twitter	OSN	53 M	1.6 B	61	3.5 M	19	[5]

Scalability and performance studies were done on three clusters: *Compton*, *Shannon*, and *Stampede*. Each node of Compton and Shannon is a dual-socket system with 64 GB or 128 GB main memory and Intel Xeon E5-2670 (Sandy Bridge) CPUs at 2.60 GHz and 20 MB last-level cache running RHEL 6.1. Stampede has two Intel Xeon E5-2680 CPUs and 1024 GB DDR3 on the large memory compute nodes running CentOS 6.3. In addition, we used *Carver* at the National Energy Research Scientific Computing Center for generating some partitions used in quality evaluations. The programs were built with the Intel C++ compiler (version 13) using OpenMP for multithreading and the `-O3` option, and we used Intel MPI (version 4.1) for MPI codes.

B. Performance Evaluation

We evaluate our PuLP partitioning methodology against both single and multi-constraint METIS and ParMETIS as well as KaFFPa from KaHIP. METIS runs used k-way partitioning with sorted heavy-edge matching and minimized edge cut. KaFFPa results use the *fastsocial* option (KaFFPa-FS), which does constrained label propagation during the initial graph contraction phase. We attempted the *ecosocial* and *strong-social* variants, but partitioning was unable to complete for

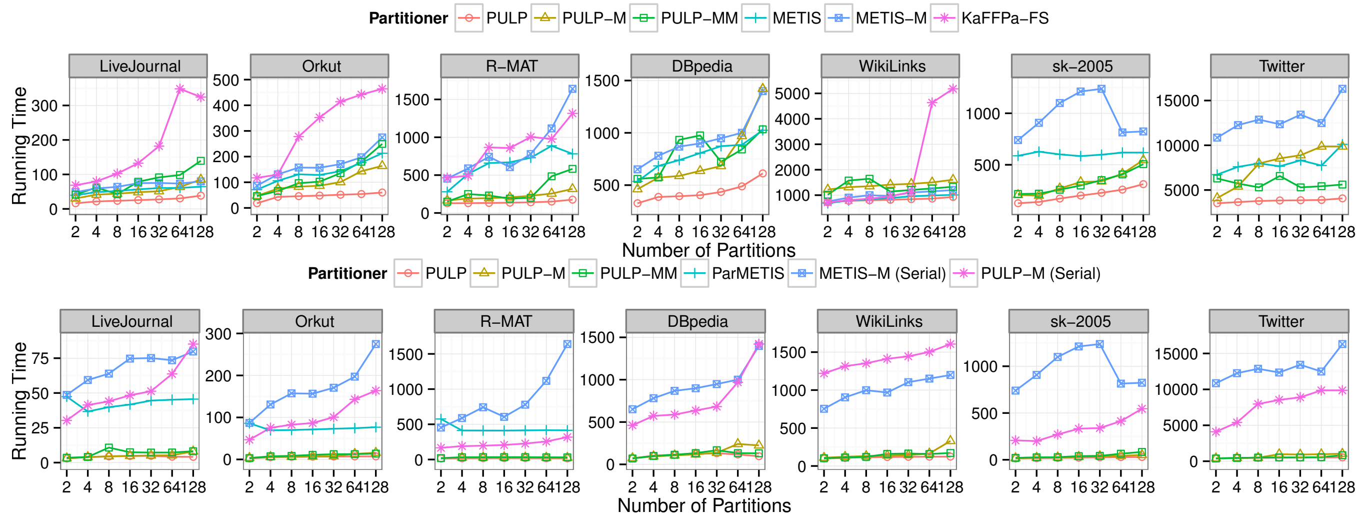

Fig. 1. Partitioning times with PuLP, METIS, and KaFFPa. Top: Comparison of **serial** running times. Bottom: Parallel and Serial running times.

TABLE III
COMPARISON OF EXECUTION TIME OF SERIAL AND PARALLEL (16 CORES) PULP-MM ALGORITHM WITH SERIAL METIS-M, KAFFPA, PARMETIS (BEST OF 1 TO 256 CORES), COMPUTING 32 PARTS. THE "ALL" SPEEDUP COMPARES PARALLEL PULP-MM TO THE BEST OF THE REST.

	Execution time (s)					**PuLP-MM Speedup**	
	Serial			Parallel		vs Best	All
Network	PuLP	METIS-M	KaFFPa	PuLP	ParMETIS	Serial	Parallel
LiveJournal	91	**75**	182	7	44	0.8×	6.1×
Orkut	**135**	170	413	12	73	1.2×	5.8×
R-MAT	**200**	778	1000	34	414	3.9×	12.2×
DBpedia	**723**	947	-	167	-	1.3×	5.6×
WikiLinks	1205	**1104**	1120	164	-	0.9×	6.7×
sk-2005	**351**	1237	-	42	-	3.5×	29.4×
Twitter	**5296**	13428	-	530	-	2.5×	25.3×

most test cases. DBpedia was not able to be partitioned in under 12 hours with any option. KaFFPa was also unable to process the sk-2005 or Twitter graphs due to a 32-bit `int` limitation. ParMETIS was only able to successfully complete on LiveJournal, Orkut, and the R-MAT graph.

We use the three aforementioned variants of PuLP for comparison, which are single-constraint single-objective (PuLP), multi-constraint single-objective (PuLP-M), and multi-constraint multi-objective (PuLP-MM). We do comparisons on the basis of edge cut, maximal per-part edge cut, execution time, and memory utilization. For all experiments, vertex imbalance ratio is set to 10%. For multi-constraint experiments, edge imbalance ratio is 50%. Due to the existance of a very high-degree vertex in DBpedia, we relax this constraint to 100% at 64 parts and 200% at 128 parts on this graph.

C. Execution Time and Memory Utilization

We first compare PuLP to METIS, ParMETIS, and KaFFPa in terms of partitioning times and memory utilizations. We compare all seven test graphs and from 2 to 128 parts. The top plots of Figure 1 give the serial execution times for all of the tested variants of PuLP, METIS, and KaFFPa on our Sandy Bridge test systems. We observe that the single-constraint single-objective variant of PuLP runs fastest in almost all test instances. We note that the serial running times of PuLP-M and PuLP-MM to also be faster than METIS and METIS with multi-constraints (METIS-M) in majority of tests.

The bottom plots of Figure 1 give the parallel execution times of the PuLP variants across 16 cores and 32 threads on our *Compton* system. We did multiple runs of ParMETIS using 1 task to 256 tasks (16 nodes) and present the **lowest runtime** over all the successful runs on LiveJournal, Orkut, and R-MAT. As a result, the speedups reported are very conservative. ParMETIS didn't complete for any of the other graphs. We also include the serial versions of PuLP-M and METIS-M for comparison. From Figure 1, we can see that the parallel execution times of all PuLP variants are minimal compared to the other partitioners. In order to better demonstrate the speedup, Table III shows the running time to compute 32 parts, comparing serial PuLP-MM with METIS-M and parallel PuLP-MM against ParMETIS, when possible, or the best of the serial methods otherwise. The parallel speedups range from about 6× to 29×.

We additionally note that the increase in running times versus number of parts increases for PuLP because it correspondingly increases the total number of vertex swaps between parts. For parallel runs, this also increases the number of required atomic updates to the global values and arrays. However, we also see a relative increase in parallel speedup for increasing part counts because, although the total number of updates increases, the contention between threads to atomically update any given value decreases when the number of threads is much less than the number of parts.

Table IV compares the maximal memory utilization of PuLP-MM, METIS and KaFFPa for computing 128 parts. Memory savings for PuLP versus the best of either METIS or KaFFPa are significant(39× for Twitter and 7.5× for sk-2005). These memory savings are primarily due to avoiding a multilevel approach. The only structures PuLP needs (in

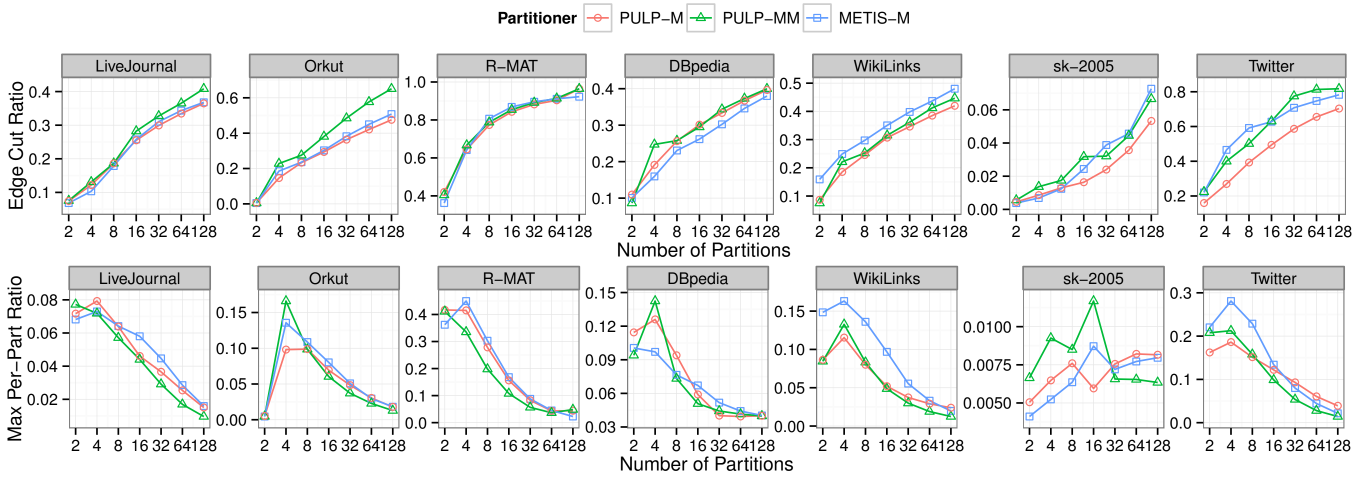

Fig. 2. Quality metrics of total cut edge ratio (top) and maximum per-part edge cut ratio (bottom) for PuLP-M, PuLP-MM and METIS-M.

TABLE IV
PuLP efficiency: Maximum memory utilization comparisons for generating 128 parts.

Network	Memory Utilization				Improv.
	METIS-M	KaFFPa	PuLP-MM	Graph Size	
LiveJournal	7.2 GB	5.0 GB	0.44 GB	0.33 GB	11×
Orkut	21 GB	13 GB	0.99 GB	0.88 GB	13×
R-MAT	42 GB	64 GB	1.2 GB	1.02 GB	35×
DBpedia	46 GB	-	2.8 GB	1.6 GB	16×
WikiLinks	103 GB	42 GB	5.3 GB	4.1 GB	8×
sk-2005	121 GB	-	16 GB	13.7 GB	8×
Twitter	487 GB	-	14 GB	12.2 GB	39×

TABLE V
Comparison of the two quality metrics, EC and EC_{max} for PuLP-MM and METIS-M when computing 32 parts. The % improvement shows relative improvement in quality for PuLP-MM with respect to METIS-M quality.

Network	PuLP-MM		METIS-M		% Improvement	
	EC	EC_{max}	EC	EC_{max}	EC	EC_{max}
LiveJournal	$14.0M$	$1.2M$	$13.3M$	$1.9M$	-5%	34%
Orkut	$56.8M$	$4.3M$	$44.8M$	$5.9M$	-26%	26%
RMAT	$118.7M$	$7.5M$	$119.3M$	$11.7M$	1/2%	56%
DBpedia	$65.2M$	$8.0M$	$57.3M$	$9.7M$	-13%	17%
WikiLinks	$196.3M$	$16.3M$	$216.1M$	$30.1M$	9%	45%
sk-2005	$58.0M$	$10.9M$	$70.2M$	$12.4M$	17%	12%
Twitter	$1250.0M$	$86.1M$	$1142.0M$	$129.1M$	-9%	33%

addition to graph storage) are the global array of length n to store the partition mappings; the vertex, edge count, and cut count arrays each of length p; and the thread-owned weight arrays also each of length p. The storage cost for all p length arrays is insignificant with a modest thread and part count. We additionally utilize a few more n length integer and boolean arrays as well as smaller thread-owned queues and arrays to speed up label propagation, as mentioned in Section III.

D. Edge Cut and Maximal Per-Part Edge Cut

Figure 2 compares the quality of the partitionings from PuLP and METIS with the seven test graph for 2 to 128 parts using multiple constraints for both programs and both the single and multiple objectives for PuLP. We report the median value obtained over 5 experiments for each part count and method. We report on partitions obtained by running PuLP in parallel, but report on partitions obtained by METIS running in serial, as ParMETIS could successfully run just three of the problems. We don't report results from KaFFPa, since the code does not currently support multiple constraints.

The top plots show the edge cuts (EC) obtained for multi-constraint METIS (METIS-M) as well as both multi-constraint (PuLP-M) and multi-constraint multi-objective PuLP (PuLP-MM). The bottom plots give the maximal per-part edge cut (EC_{max}) as a ratio of total edges. Taken together, these two plots demonstrate the tradeoff offered by PuLP to minimize either the total edge cut at a cost of maximal per-part edge cut or to minimize the maximal per-part edge cut at a cost of total edge cut.

We observe PuLP-M does better than METIS-M for Twitter and sk-2005; as good as METIS-M for WikiLinks, Livejournal, R-MAT, and Orkut; and worse on DBpedia in terms of total edge cuts. PuLP-MM does slightly worse than METIS for three graphs and comparable for four graphs, but results in much better partitions in terms of the maximal per-part edge cut. A likely reason as to why PuLP does not work as well as METIS on DBpedia is because DBpedia is a generated RDF graph, not having the same community structure inherent to the other test graphs. Therefore, it does not derive the same degree of benefit from utilizing a label-propagation-based approach. We also observe that our multi-objective PuLP occasionally out-performs our single-objective PuLP on some test instances. This can be explained by the additional dynamic weighting parameters which can more fully explore the search space and are therefore more likely to avoid local minima.

As mentioned, the bottom plots of Figure 2 demonstrate that multi-objective PuLP can be relatively effective at minimizing the maximal per-part edge cut on partitions derived from these graphs at a small cost to total edge cut. Table V shows this tradeoff for 32 parts. We compare the quality of both the metrics, EC and EC_{max}, and observe that PULP-MM improves EC_{max} substantially (12% – 56%) when compared with METIS-M at modest expense of the edge cut in some graphs.

Note that while we include METIS-M for comparison, it does not explicitly attempt to minimize edge cut balance. Also note that other experiments showed that tightening the edge balance in METIS will also inherently improve the maximal per-part edge cut. However, PULP-MM still demonstrated overall better performance in these experiments. The results of these tests are not shown for space considerations. In instances where PULP-M outperforms PULP-MM, examination of results show that while the overall edge cut balance has been improved by the approach, the higher total edge cut that results can offset the derived benefits for smaller part counts. The jump seen on select graphs between two and four parts can be explained by the fact that the bipartitioning problem on those instances is a relatively much easier problem than the 4-way partitioning.

V. Related Work

There are three other works known to us that use label propagation for the task of partitioning large-scale graphs. We compare our results with their published results as the codes are not available publicly. Vaquero et al. [33] implement vertex-balanced label propagation to partition dynamically changing graphs. Ugander and Backstrom [32] implement label propagation for vertex-balanced partitioning as an optimization problem. They report performance on a similarly pre-processed LiveJournal graph for generating 100 parts, with a serial running time of 88 minutes and resultant edge cut ratio of 0.49. By comparison, our multi-constraint and multi-objective serial code creates 128 parts of the LiveJournal graph in about two minutes and produces an edge cut ratio of about 0.41. Wang et al. [34] utilize label propagation in a manner similar to KaFFPa, which is a multilevel approach with label propagation during graph coarsening. At the coarsest level, METIS is used to partition the graph. They also implement non-multilevel partitioning via a label propagation step followed by a greedy balancing phase. Their multilevel single-constraint and single-objective approach to partition pre-processed LiveJournal has a serial running time of about 75 seconds, consumes about 1.5 GB memory, and has an edge cut about 25% greater than that produced by METIS alone. By comparison, our code consumes only 440 MB memory and produces cut quality comparable to or better than METIS on the same graph. Their non-multilevel approach runs in about half the time, but at a considerable cost to cut quality.

VI. Conclusions and Future Work

In this paper, we present PULP, a new fast multi-objective multi-constraint partitioner for scalable partitioning of small-world networks. The partitioning method in PULP is based on the label propagation community detection method. In a fraction of the execution time, while consuming an order of magnitude less memory, PULP can produce partitions comparable or better in terms of total edge cut to the k-way multilevel partitioning scheme in METIS. In addition, PULP produces partitions that are better in terms of maximal cut edges per-part. In future work, we will apply partitionings from PULP to a larger set of graph computations, as well as fully explore the input parameters and weighting functions governing PULP's partitioning phases. Additionally, exploring other community detection methods (e.g. [11]) for complex objective partitioning might make for promising future work.

Acknowledgment

This work is supported by NSF grant ACI-1253881, the DOE Office of Science through the FASTMath SciDAC Institute and by NSF grant ACI-1253881, and used instrumentation funded by the NSF grant OCI-0821527 as well as the National Energy Research Scientific Computing Center, supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. 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.

References

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

[2] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "UbiCrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[3] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2D graph partitioning," in *Proc. Int'l. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

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

[5] 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.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. Int'l. Conf. on Data Mining (SDM)*, 2004.

[7] A. Ching and C. Kunz, "Giraph: Large-scale graph processing infrastructure on Hadoop," *Hadoop Summit*, vol. 6, no. 29, p. 2011, 2011.

[8] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[9] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. Conf. on Design Automation*, 1982.

[10] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.

[11] D. F. Gleich and C. Seshadhri, "Vertex neighborhoods, low conductance cuts, and good seeds for local community methods," in *KDD*, 2012.

[12] 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.

[13] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "Towards benchmarking graph-processing platforms," in *Proc. Supercomputing (SC), poster*, 2013.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. IEEE Int'l. Conf. on Data Mining (ICDM)*, 2009.

[15] 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 July 2014.

[16] ——, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[17] ——, "Parallel multilevel K-way partitioning scheme for irregular graphs," in *Proc. ACM/IEEE Conference on Supercomputing (SC)*, 1996.

[18] ——, "Multilevel algorithms for multi-constraint graph partitioning," in *Proc. ACM/IEEE Conference on Supercomputing (SC)*, 1998.

[19] J. Kunegis, "KONECT - the Koblenz network collection," konect.uni-koblenz.de, last accessed July 2014.

[20] 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*, vol. 6, no. 1, pp. 29–123, 2009.

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

[22] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo, "DBpedia SPARQL benchmark - performance assessment with real queries on real data," in *Proc. Int'l. Semantic Web Conf. (ISWC)*, 2011.

[23] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2013.

[24] A. Pinar and B. Hendrickson, "Partitioning for complex objectives," in *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, ser. IPDPS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 121–.

[25] L. Quick, P. Wilkinson, and D. Hardcastle, "Using Pregel-like large scale graph processing frameworks for social network analysis," in *Proc. Int'l. Conf. on Advances in Social Networks Analysis and Mining (ASONAM)*, 2012.

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

[27] K. Schloegel, G. Karypis, and V. Kumar, "Parallel multilevel algorithms for multi-constraint graph partitioning," in *Proc. Euro-Par 2000 Parallel Processing*, 2000.

[28] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proc. ACM Int'l. Conf. on Management of Data (SIGMOD)*, 2013.

[29] G. M. Slota, S. Rajamanickam, and K. Madduri, "BFS and coloring-based parallel algorithms for strongly connected components and related problems," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[30] "Stanford large network dataset collection," http://snap.stanford.edu/data/index.html, last accessed July 2014.

[31] B. Uçar and C. Aykanat, "Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies," *SIAM Journal on Scientific Computing*, vol. 25, no. 6, pp. 1837–1859, 2004.

[32] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proc. Web Search and Data Mining (WSDM)*, 2013.

[33] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "xDGP: A dynamic graph processing system with adaptive partitioning," *CoRR*, vol. abs/1309.1049, 2013.

[34] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 568–579.

[35] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proc. IEEE Int'l. Conf. on Data Mining (ICDM)*, 2012, pp. 745–754.