SIAM J. SCI. COMPUT.
Vol. 38, No. 5, pp. S620–S645

COMPLEX NETWORK PARTITIONING USING LABEL PROPAGATION[*]

GEORGE M. SLOTA[†], KAMESH MADDURI[†], AND SIVASANKARAN RAJAMANICKAM[‡]

Abstract. We present PuLP (partitioning using label propagation), a parallel and memory-efficient graph partitioning method specifically designed to partition low-diameter networks with skewed degree distributions on shared-memory multicore platforms. Graph partitioning is an important problem in scientific computing because it impacts the execution time and energy efficiency of computations on distributed-memory platforms. Partitioning determines the in-memory layout of a graph, which affects locality, intertask load balance, communication time, and overall memory utilization. A novel feature of our PuLP method 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 up to 7.8× less memory than state-of-the-art partitioners and is 5.0× faster, on average, than alternate approaches (with 16-way parallelism). We also achieve better partitioning quality results for the multiobjective scenario.

Key words. partitioning, label propagation, parallel, graph algorithm, small-world

AMS subject classifications. 68W10, 05C85, 05C70

DOI. 10.1137/15M1026183

1. 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., [8, 18, 31]). Also, several open-source and commercial distributed graph processing frameworks (e.g., PowerGraph [11], Giraph [7], Trinity [28], PEGASUS [13]) 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 [12, 25], but all of them require an initial vertex and edge partitioning for scalability in a distributed-memory setting.

[*]Received by the editors June 16, 2015; accepted for publication (in revised form) May 25, 2016; published electronically October 27, 2016. 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 and CCF-1439057 and used instrumentation funded by 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 DE-AC02-05CH11231. This work was also suppoerted by the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000. Portions of this work previously appeared in *Proceedings of the IEEE International Conference on Big Data*, 2014.

http://www.siam.org/journals/sisc/38-5/M102618.html

[†]Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802 (gslota@psu.edu, madduri@cse.psu.edu).

[‡]Scalable Algorithms Department, Sandia National Laboratories, Albuquerque, NM 87123 (srajama@sandia.gov).

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 intertask 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 current partitioners.
3. Small-world graphs lack good vertex and edge separators [19]. 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 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 interpartition edges (or the edge cut), as it loosely correlates to internode communication costs. Since communities are tightly connected vertices, co-locating vertices of a community in a partition will increase the proportion of intrapartition edges. Other related work has used label propagation as part of a multilevel scheme for graph coarsening [21, 22, 35] or, similar to PuLP, as part of a single-level method [20, 33, 34]. However, to our knowledge, PuLP is the first label propagation–based partitioner that considers both multiple constraints and multiple objectives. It is also the first single-level partitioner that can produce cut quality comparable to state-of-the-art multilevel partitioners with multiple constraints.

In typical graph analytic algorithms, the number of vertices/edges in each partition represents the local work and the memory usage. In parallel graph analytics utilizing a bulk synchronous parallel 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-part 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 [14, 16] and ParMETIS [15]. We use the multiple constraint version of both the codes [17, 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 15 large-scale small-world graphs (number of edges range from 253,000 to 1.8 billion).

For the large networks and commonly used quality measures (edge cut), our partitioning scheme is comparable to METIS and is better than it in additional objectives (maximum per-part edge cut) for a wide range of partition counts (2–1024) 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], PuLP takes less than a minute on a single compute node to generate 32-way partitions of this graph while satisfying given vertex and edge balance constraints. Due to higher memory requirements than the 64 GB available on the test node, neither METIS, ParMETIS, nor KaFFPa was able to successfully partition this graph on our test system.

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

2. Preliminaries: Graph distribution and partitioning.

We consider parallelizing analytics over large and sparse graphs: the numbers of vertices (n) are on the order of at least tens of millions, and the numbers of edges (m) are much closer to order $O(n)$ instead of order $O(n^2)$. The graph organization/layout in a distributed-memory system is characterized by the "distribution, partitioning, ordering" triple. 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 one-dimensional (1D) distribution and natural ordering.

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. The advantages of the 1D scheme are its simplicity, memory efficiency, ease of parallel implementation of most graph computations using an "owner computes" model, and the fact that the interaction of graph partitioning methods and 1D distributions is well understood [1]. 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. However, we focus on 1D instead of 2D or hybrid partitioning methods for this work, due to the increased complexity of algorithm design, primarily the communication steps, and increased memory usage for storing the graph (since both the row and column dimensions may be sparse) that 2D and hybrid partitioning methods require. Potential future work might be to utilize the 1D partitions that we compute in a 2D distribution [5] or with other distribution schemes such as degree-based ones [23].

The specific partitioning problem we are interested in for graph analytic applications and which 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 nearly balanced partition such that $\forall i = 1 \ldots p$,

$$(1 - \epsilon_l)\,\frac{|V|}{p} \le |V(\pi_i)| \le (1 + \epsilon_u)\,\frac{|V|}{p}, \tag{2.1}$$

$$|E(\pi_i)| \le (1 + \eta_u)\,\frac{|E|}{p}, \tag{2.2}$$

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

$$C(G, \Pi) = \{\{(u, v) \in E\} \mid \Pi(u) \ne \Pi(v)\}\,, \tag{2.3}$$

$$C(G, \pi_k) = \{\{(u, v) \in C(G, \Pi)\} \mid (u \in \pi_k \vee v \in \pi_k)\}\,. \tag{2.4}$$

Our partitioning problem is then to minimize the two metrics

$$EC(G, \Pi) = |C(G, \Pi)|, \tag{2.5}$$

$$EC_{max}(G, \Pi) = \max_k |C(G, \pi_k)|. \tag{2.6}$$

This can also be generalized for graphs with edge weights and vertex weights. PuLP can be extended to handle other metrics like the total communication volume and the maximum per-part communication volume. The communication volume for a given part i can be simply defined as the total number of vertices that are neighbors to vertices in part i that are assigned to a different part. As this constitutes only one direction of communication, the total communication volume would be the sum communication volumes of all parts produced by the partitioning.

In the past, multiconstraint graph partitioning with the EC objective has been implemented in METIS and ParMETIS [17, 27]. We will compare PuLP with both these methods in section 4. Pinar and Hendrickson [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 multiobjective partitionings [32] and multiconstraint and multiobjective partitionings [6] for hypergraph partitioning. However, hypergraph methods are often much more compute-intensive than graph partitioning methods.

3. 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.

3.1. Label propagation.

Label propagation was originally proposed as a fast community detection algorithm [26]. 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 l distinct labels. The value of l is usually the number of vertices

Algorithm 1. Baseline label propagation algorithm.

```
procedure LABEL-PROP($G(V, E), l, Iter$)
    for all $v \in V$ do
        $L(v) \leftarrow$Vid($v$) or Rand($1 \cdots l$)
    $i \leftarrow 0$, $updates \leftarrow 1$
    while $i < Iter$ and $updates \neq 0$ do
        $updates \leftarrow 0$
        for all $v \in V$ do
            $Counts(1 \cdots l) \leftarrow 0$
            for all $\langle v, u \rangle \in E$ do
                $Counts(L(u)) \leftarrow Counts(L(u)) + 1$
            $x \leftarrow$ GetMax($Counts(1 \cdots l)$)
            if $x \neq L(v)$ then
                $L(v) \leftarrow x$
                $updates \leftarrow updates + 1$
        $i \leftarrow i + 1$
    return $L$
```

Algorithm 2. PuLP multiconstraint multiobjective algorithm.

```
procedure PuLP-MM($G(V, E), p, Iter_{loop}, Iter_{bal}, Iter_{ref}$)
    $P \leftarrow$ PuLP-Init($G(V, E), p, P$)
    $N_{vert}(1 \cdots p) \leftarrow$ vertex counts in $P(1 \cdots p)$
    for $i = 1 \cdots Iter_{loop}$ do
        $P \leftarrow$ PuLP-VertBal($G(V, E), p, P, N_{vert}, Iter_{bal}$)
        $P \leftarrow$ PuLP-VertRef($G(V, E), p, P, N_{vert}, Iter_{ref}$)
    $N_{edge}(1 \cdots p) \leftarrow$ edge counts in $P(1 \cdots p)$
    $N_{cut}(1 \cdots p) \leftarrow$ edge cuts in $P(1 \cdots p)$
    for $i = 1 \cdots Iter_{loop}$ do
        $P \leftarrow$ PuLP-CutBal($G(V, E), p, P, N_{vert}, N_{edge}, N_{cut}, Iter_{bal}$)
        $P \leftarrow$ PuLP-CutRef($G(V, E), p, P, N_{vert}, N_{edge}, N_{cut}, Iter_{ref}$)
    return $P$
```

in the graph (each vertex v gets a distinct label, usually its numeric vertex identifier, $Vid(v)$), but it can also 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 *Counts*. After examining all neighbors, v updates its current label to whichever label has the maximal count in *Counts* with ties broken randomly.

The algorithm proceeds to iterate over all V until some stopping criterion is met. This stopping criterion is usually some fixed number of iterations *Iter*, as we show, or convergence being achieved through no label assignments updating during a single iteration (number of *updates* is zero). For large graphs, there is no guarantee that convergence will be reached quickly, so a fixed iteration count is usually preferred in practice. As with l, the maximal iteration count is usually determined experimentally. Since each iteration performs linear work with regards to the size of the input graph, this results in an overall linear and efficient algorithm.

3.2. PuLP overview. In general, label propagation methods are attractive for community detection due to their low computational overhead and low memory utilization, as well as the relative ease of parallelization. In PuLP, we utilize weighted label propagation in several separate stages to partition an input graph. Algorithm 2

TABLE 1
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 parts
ϵ_u	Vertex upper balance constraint {0.1}
η_u	Edge upper balance constraint {0.1}
$Iter_{loop}$	# of iterations in outer loop {3}
$Iter_{bal}$	# of iterations in balanced propagation stage {5}
$Iter_{ref}$	# of iterations in constrained refinement stage {10}
PuLP-X	PuLP subroutine for various stages

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 multiconstraint multiobjective partitioning, or PuLP-MM. We first initialize the partition using a multisource breadth-first search (BFS) with p randomly selected initial roots (PuLP-Init in Algorithm 3). The initial (unbalanced) partitioning is then passed to an iterative stage that first balances the number of vertices in each part through weighted label propagation (PuLP-VertBal listed in Algorithm 4) while minimizing the edge cut and then improves the edge cut on the balanced partition through a refinement stage motivated by FM-refinement [9] (PuLP-VertRef). The next iterative stage further balances the number of edges per part while minimizing and balancing the per-part edge cut through weighted label propagation (PuLP-CutBal listed in Algorithm 6) and then refines the achieved partitions through constrained refinement (PuLP-CutRef as shown in Algorithm 7). More details of these stages are in the following subsections. We demonstrate a more quantitative analysis on the sensitivity of results to varying iteration counts and algorithmic changes in our results.

The input parameters to PuLP are listed in Table 1. Listed in the braces are the default values we used for partitioning the graphs during our experiments. The vertex and edge balance constraints (ϵ_u and η_u) are selected based on what might be reasonable in practice for a typical graph analytic code running on a small-world

Algorithm 3. PuLP BFS-based partition initialization procedure.

```
procedure PuLP-Init($G(V,E), p, P$)
    $Q \leftarrow \varnothing$,$Q_n \leftarrow \varnothing$
    $P(1 \cdots |V|) = none$
    for $i = 1$to$p$ do
        $v \leftarrow$UniqueRand($V$)
        $P(v) \leftarrow i$
        Add $v$ to $Q$
    while $Q \neq \varnothing$ do
        for all $v \in Q$ do
            for all $(v,u) \in E$ do
                if $P(u) = none$ then
                    $P(u) \leftarrow P(v)$
                    Add $u$ to $Q_n$
        $Q \leftarrow \varnothing$
        Swap($Q$, $Q_n$)
    return $P$
```

graph. The iteration counts we use ($Iter_{loop}$, $Iter_{bal,}$, $Iter_{ref}$) 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.

3.3. PuLP initialization. To first initialize our partitions, we use a randomly sourced BFS approach similar to graph growing as implemented in prior work [10, 16]. Our BFS initialization approach is demonstrated in Algorithm 3. To perform our initialization, we randomly select p initial roots (where p is the number of desired parts) and perform a parallel multisource level-synchronous BFS from these roots. We use a queue-based approach where Q contains the vertices to be explored on the current level, and Q_n is the queue where vertices are placed as they are discovered for exploration on the next level. The queues are swapped at the end of each level. Vertices discovered as descendants from one of these roots are marked as in the same initial part as that root. We impose no explicit constraints during this stage. We note that an initial (vertex/edge) imbalance is expected to occur, the severity of which is dependent on the randomly selected roots; however, in practice, this problem is observed to be minimal, as the subsequent iterative stages are capable of rebalancing the parts. This approach has resulted in the most consistent and highest quality end partitions across a wide range of tested variants, including ones that enforced loose balance constraints up to 100% imbalance.

Some of the tested variants have included the original random label propagation-based approach [29], doing a multisource BFS with a variety of loose balance constraints, iteratively performing single-source BFSs through a variety of methods to select the roots, and performing multisource BFSs with a variety of nonrandom methods to select the roots. Each of these tested variants had several additional subvariants; however, the approach given by Algorithm 3 resulted in the highest average partition quality in terms of edge cut and max per-part cut across the DIMACS and LAW collections of regular and irregular test graphs with part counts as presented later in our results. This unconstrained BFS method performed approximately $2\times$ better on average than the other implemented BFS methods and approximately $20\times$ better than the original label propagation-based approach [29] on the regular DIMACS problems. This large improvement against the original approach was the result of the label propagation initialization performing very poorly on the regular graphs and meshes that do not contain any intrinsic community structure, where the resulting partitions ended up comprising multiple small and disconnected components with high relative edge cuts. The selected method also benefits in terms of ease and scalability of parallelization.

3.4. PuLP vertex balancing and total edge cut minimization. With the initial partitioning, PuLP-VertBal (Algorithm 4) balances the vertex counts between parts to satisfy our original balance constraint. Here, we use degree-weighted label propagation. In lieu of doing the standard label propagation approach of assigning to a given vertex v a label based on the maximal label count, $\mathrm{Max}(C(1 \cdots p))$, of all of its neighbors $\langle v, u \rangle \in E$, we utilize an additional degree weighting by considering the size of the neighborhood of u ($|E(u)|$ in Algorithm 4). 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. Doing as such 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 can generalize this to alternative objectives by considering all $C(1 \cdots p)$ values as a benefit function

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

```
procedure PuLP-VertBal($G(V,E), P, p, N_{vert}, I_b$)
    $i \leftarrow 0$, $updates \leftarrow 1$
    $Max_{vert} \leftarrow (n/p) \times (1+\epsilon_u)$
    $W_{vert}(1\cdots p) \leftarrow$ GetMax($Max_{vert}/N_{vert}(1\cdots p) - 1$,0)
    while $i < Iter_{bal}$ and $updates \neq 0$ do
        $updates \leftarrow 0$
        for all $v \in V$ do
            $C(1\cdots p) \leftarrow 0$
            for all $\langle v,u\rangle \in E$ do
                $C(P(u)) \leftarrow C(P(u)) + |E(u)|$
            for $j = 1\cdots p$ do
                if Moving $v$ to $P_j$ violates $Max_{vert}$ then
                    $C(j) \leftarrow 0$
                else
                    $C(j) \leftarrow C(j) \times W_{vert}(j)$
            $x \leftarrow$ GetMax($C(1\cdots p)$)
            if $x \neq P(v)$ then
                Update($N_{vert}(P(v)), N_{vert}(x)$)
                Update($W_{vert}(P(v)), W_{vert}(x)$)
                $P(v) \leftarrow x$
                $updates \leftarrow updates + 1$
        $i \leftarrow i + 1$
    return $P$
```

for moving v to some partition $(1\cdots p)$. Considering, for example, communication volume, then $C(1\cdots p)$ would be the reduction in communication volume for moving v to each of $(1\cdots p)$.

There are two additional changes from baseline label propagation to make note of. First, for any part that is overweight, i.e., the number of vertices in that current part π_q ($N_{vert}(q)$ in the algorithm) is greater than our desired maximal Max_{vert}, we do not allow that part to accept new vertices. Max_{vert} is the maximum number of vertices allowed in a given part, depending on the balance constraint ϵ_u. Second, there is an additional weighting parameter $W_{vert}(1\cdots p)$ that is based on how underweight any part currently is. For a given part q, $W_{vert}(q)$ will approach infinity as the size of that part approaches zero and will approach zero as the size of the part approaches Max_{vert}. For part sizes above Max_{vert}, we will consider the weight to be zero. As shown in Algorithm 4, we calculate W_{vert} for any given part q as $W_{vert}(q) = Max_{vert}/N_{vert}(q)$, where $N_{vert}(q)$ is the current size of part q in terms of vertices. When $N_{vert}(q) > Max_{vert}$, then $W_{vert}(q) = 0$.

This weighting forces larger parts to give vertices away with a preference toward the current most underweight 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 is overweight and there is little difference in W_{vert} values, this scheme will default to basic degree-weighted label propagation. This weighting-based approach is similar to that employed in related work [20]. A possible drawback we will note is that the only new part assignments any vertex is able to assume are the current part assignments of its neighbors, which might limit the ability of the algorithm to achieve proper balance. However, due to the low diameter and general small-world structural characteristics of the graphs for which PuLP is designed and optimized, subdomain connectivity of partitions is very high (i.e., there usually exists multiple edges between any given pair of q_i,q_j parts).

Algorithm 5. PuLP single-objective vertex constrained refinement stage.

```
procedure PuLP-VertRef($G(V,E), P, p, N_{vert}, Iter_{ref}$)
    $i \leftarrow 0$, $updates \leftarrow 1$
    $Max_{vert} \leftarrow (n/p) \times (1 + \epsilon_u)$
    while $i < Iter_{ref}$ and $updates \neq 0$ do
        $updates \leftarrow 0$
        for all $v \in V$ do
            $C(1 \cdots p) \leftarrow 0$
            $x \leftarrow P(v)$
            for all $\langle v, u \rangle \in E$ do
                $C(P(u)) \leftarrow C(P(u)) + 1$
                if $C(P(u)) > C(x)$ then
                    $x \leftarrow P(u)$
            if Moving $v$ to $P_x$ does not violate $Max_{vert}$ then
                $P(v) \leftarrow x$
                Update($N_{vert}(P(v)), N_{vert}(x)$)
                $updates \leftarrow updates + 1$
        $i \leftarrow i + 1$
    return $P$
```

This allows vertices to move fluidly among all parts and vertex balance to be quickly achieved. Each outer loop of Algorithm 4 runs in $O(np+m)$, so it is linear in the size of the network for a fixed p. Here, p is the number of parts/labels and n and m and the numbers of vertices and edges in the graph, respectively.

We further explicitly minimize edge cut with a greedy refinement stage, as given in Algorithm 5. The 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 (Max_{vert}). We calculate possible refinements by examining the part assignments of all neighbors of a given vertex v, and if the number of neighbors in an adjacent part is greater than the number of neighbors in v's current part, moving v to the adjacent part will result in an overall lower edge cut. Similarly with an alternative objective such as communication volume, we would greedily move vertices to a new part if it improves upon the objective without violating a constraint. As with the balancing stages, each iteration of the refinement stages run in $O(np + m)$ time. Note that for alternative objectives, this running time bound might increase.

We perform $Iter_{loop}$ iterations of balancing (Algorithm 4) and refining (Algorithm 5) before moving on to the stages with other partitioning objectives and constraints. However, in order to create only 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. We note that very simple changes to Algorithms 4 and 5 would allow us to constrain edge balance instead of vertex balance.

3.5. PuLP edge balancing and EC_{max} minimization.

Once we have a vertex balanced partitioning that minimizes edge cut, PuLP balances edges per part and minimizes per-part edge cut (Algorithm 6). The total edge cut might increase because of the new objective, hence the algorithm uses a dynamic weighting scheme to achieve a balance between optimizing for the two different objectives while also refining just for the total edge cut objective. The algorithm also ensures the vertex and edge balance constraints will become satisfied if possible. And while the approach uses weighted label propagation under given constraints (similarly to Algorithm 4), there are a number of nuances to make note of.

Algorithm 6. PuLP multiobjective vertex and edge-constrained label propagation stage.

```
procedure PuLP-CutBal($G(V,E), P, p, N_{vert}, N_{edge}, N_{cut}, Iter_{bal}$)
    $i \leftarrow 0$, $r \leftarrow 1$
    $Max_{vert} \leftarrow (n/p) \times (1+\epsilon_u)$
    $Max_{edge} \leftarrow (m/p) \times (1+\eta_u)$
    $Cur_{Max_{edge}} \leftarrow$ GetMax($N_{edge}(1 \cdots p)$)
    $Cur_{Max_{cut}} \leftarrow$ GetMax($N_{cut}(1 \cdots p)$)
    $W_{edge}(1 \cdots p) \leftarrow Cur_{Max_{edge}} / N_{edge}(1 \cdots p) - 1$
    $W_{cut}(1 \cdots p) \leftarrow Cur_{Max_{cut}} / N_{cut}(1 \cdots p) - 1$
    $d_{edge} \leftarrow 1$, $d_{cut} \leftarrow 1$
    while $i < Iter_{bal}$ and $updates \neq 0$ do
        $updates \leftarrow 0$
        for all $v \in V$ do
            $C(1 \cdots p) \leftarrow 0$
            for all $\langle v,u \rangle \in E$ do
                $C(P(u)) \leftarrow C(P(u)) + 1$
            for $j = 1 \cdots p$ do
                if Moving $v$ to $P_j$ violates $Max_{vert}$, $Cur_{Max_{edge}}$, $Cur_{Max_{cut}}$ then
                    $C(j) \leftarrow 0$
                else
                    $C(j) \leftarrow C(j) \times (W_{edge}(j) \times d_{edge} + W_{vert}(j) \times d_{cut})$
            $x \leftarrow$ GetMax($C(1 \cdots p)$)
            if $x \neq P(v)$ then
                $P(v) \leftarrow x$
                Update($N_{vert}(P(v)), N_{vert}(x)$)
                Update($N_{edge}(P(v)), N_{edge}(x)$)
                Update($N_{cut}(P(v)), N_{cut}(x)$)
                Update($W_{edge}(P(v)), W_{edge}(x)$)
                Update($W_{cut}(P(v)), W_{cut}(x)$)
                Update($Cur_{Max_{edge}}$,$Cur_{Max_{cut}}$)
                $updates \leftarrow updates + 1$
        if $Cur_{Max_{edge}} < Max_{edge}$ then
            $Cur_{Max_{edge}} \leftarrow Max_{edge}$
            $d_{cut} \leftarrow d_{cut} \times Cur_{Max_{cut}}$
            $d_{edge} \leftarrow 1$
        else
            $d_{edge} \leftarrow d_{edge} \times (Cur_{Max_{edge}} / Max_{edge})$
            $d_{cut} \leftarrow 1$
        $i \leftarrow i + 1$
    return $P$
```

Initially, we do not use the given edge balance constraint explicitly. Instead, a relaxed constraint based on the current maximal edge count across all parts $Cur_{Max_{edge}}$ is used to compute the edge balance weights ($W_{edge}(1 \cdots p)$). The edge balance weights are similar to the vertex balance weights ($W_{vert}(1 \cdots q)$), in that these per-part weighting values increase inversely proportionally with how underweight each part is relative to the current maximum (calculated as $W_{edge}(q) = Cur_{Max_{edge}}/N_{edge}(q)$). This results in the possibility of all parts receiving more edges with the exception of which part is currently the largest, but no part will receive enough edges to become greater than $Cur_{Max_{edge}}$. As the largest part can only give away vertices and edges, $Cur_{Max_{edge}}$ is iteratively tightened until the given edge balance constraint is met. Once we pass the threshold given by our input constraint, we fix $Cur_{Max_{edge}}$ to be equal to Max_{edge}. To minimize the maximum edges cut per-part, we employ a similar procedure with $Cur_{Max_{cut}}$ and the weightings for maximum cut edges ($W_{cut}(1 \cdots p)$). We iteratively tighten this bound so that, although we have no assurance that the global edge cut will decrease, we will always be decreasing the maximal edges cut per part.

We also introduce two additional dynamic weighting terms d_{edge} and d_{cut} that serve to shift the focus of the algorithm between hitting the Max_{edge} constraint and minimizing $Cur_{Max_{cut}}$. For every iteration of the algorithm that the Max_{edge} constraint is not satisfied, d_{edge} is increased by the ratio of which $Cur_{Max_{edge}}$ is greater than Max_{edge}. This shifts the weighting function to give higher preference toward moving vertices to parts with low edge counts instead of attempting to minimize the edge cut balance. Likewise, when the edge balance constraint is satisfied, we reset d_{edge} to one and iteratively increase d_{cut} 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 multiconstraint single-objective problem by not including $Cur_{Max_{cut}}$ and W_{cut} in our weighting function or constraint checks. We demonstrate this later in section 4 by running PuLP Multiconstraint single-objective, or PuLP-M. Additionally, we can instead turn the problem into a single-constraint three-objective problem by ignoring Max_{edge} altogether and instead just attempt to further minimize both $Cur_{Max_{edge}}$ and $Cur_{Max_{cut}}$ along with total edge cut. Finally, we would be able to completely generalize this algorithm into an arbitrary number of q constraints. We would do this by calculating $Max_{1 \cdots q}$, $Cur_{Max_{1 \cdots q}}$, and $W_{1 \cdots q}(1 \cdots p)$ for each of the q constraints, progressively increasing one of $d_{1 \cdots q}$ while setting the others to 1, and then using these to calculate the additional weightings to $C(1 \cdots p)$ as $C(1 \cdots p) \leftarrow C(1 \cdots p) \times \sum_{k=1}^{q} W_k(1 \cdots p) * d_k$. We can also again adjust the objective by altering how we initialize the $C(1 \cdots p)$ array for each v. As with Algorithm 4, each iteration of Algorithm 6 also runs in $O(np + m)$.

After the completion of Algorithm 6, we again perform a constrained refinement, given by Algorithm 7. This algorithm uses the current maximal balance sizes of Max_{vert}, $Cur_{Max_{edge}}$, and $Cur_{Max_{cut}}$, and we again attempt to find a local minimum for the total edge cut without violating any of these current balances. Although we have a hard cutoff for the number of outer loop iterations $Iter_{loop}$ shown in Algorithm 2, in practice, we continue to iterate between Algorithm 6 and Algorithm 7 if our secondary constraint has not been achieved and progress is still being made toward reaching it.

3.6. Algorithm parallelization and optimization. One of the strengths of using label propagation for partitioning is that its vertex-centric nature lends itself to 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_{vert}, $Cur_{Max_{edge}}$, $Cur_{Max_{cut}}$, d_{edge}, and d_{cut} as well as N_{vert}, N_{edge}, and N_{cut} are all global values and arrays and are updated in a thread-safe manner. Each thread creates and updates its own C, W_{vert}, W_{edge}, and W_{cut} counting and weighting 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* [30]. This technique avoids

Algorithm 7. PuLP multiobjective vertex and edge-constrained refinement stage.

```
procedure PuLP-CutRef($G(V,E), P, p, N_{vert}, N_{edge}, N_{cut}, Iter_{ref}$)
    $i \leftarrow 0$, $updates \leftarrow 1$
    $Max_{vert} \leftarrow (n/p) \times (1+\epsilon_u)$
    $Max_{edge} \leftarrow (m/p) \times (1+\eta_u)$
    $Cur_{Max_{edge}} \leftarrow$ GetMax(GetMax($N_{edge}(1 \cdots p)$),$Max_{edge}$)
    $Cur_{Max_{cut}} \leftarrow$ GetMax($N_{cut}(1 \cdots p)$)
    while $i < Iter_{ref}$ and $updates \neq 0$ do
        $updates \leftarrow 0$
        for all $v \in V$ do
            $C(1 \cdots p) \leftarrow 0$
            $x \leftarrow P(v))$
            for all $\langle v,u \rangle \in E$ do
                $C(P(u)) \leftarrow C(P(u)) + 1$
                if $C(P(u)) > C(x)$ then
                    $x \leftarrow P(u)$
            if Moving $v$ to $P_x$ does not violate $Max_{vert}$, $Cur_{Max_{edge}}$, $Cur_{Max_{cut}}$ then
                $P(v) \leftarrow x$
                Update($N_{vert}(P(v)), N_{vert}(x)$)
                Update($N_{edge}(P(v)), N_{edge}(x)$)
                Update($N_{cut}(P(v)), N_{cut}(x)$)
                $updates \leftarrow updates + 1$
        $i \leftarrow i + 1$
    return $P$
```

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 nonqueue implementations as well as our serial and parallel code.

4. Results and discussion.

4.1. Experimental setup.

We evaluate performance of our new PuLP partitioning strategies on the small-world networks comprising the Laboratory for Web Algorithmics (LAW) group from the University of Florida Sparse Matrix Collection [2, 3, 4, 8]. Additional tests were performed on the graphs comprising the test suite from the 10th DIMACS Implementation Challenge [1]. The LAW graphs and their global properties after preprocessing are listed in Table 2. We omit listing the DIMACS graphs for brevity. We preprocessed the LAW graphs before partitioning by removing directivity in edges and deleting all degree-0 vertices and multiedges, and extracted the largest connected component. Ignoring I/O, this preprocessing required minimal computational time, only on the order of seconds in serial for each of the datasets. To determine approximate diameter, we perform iterative BFSs rooted at a randomly selected vertex from the farthest level of the previous search. We return the maximum number of levels as determined by the BFSs after it doesn't increase in 10 consecutive iterations.

Scalability and performance studies were done on the *Compton* testbed cluster at Sandia National Laboratories. Each node of Compton is a dual-socket 16 core system with 64 GB main memory and Intel Xeon E5-2670 (Sandy Bridge) CPUs at 2.60 GHz with 20 MB last-level cache running RHEL 6.1. The executables were built with the Intel C++ compiler (version 13) using OpenMP for multithreading and the `-O3` option.

Table 2

Test graph characteristics after *preprocessing. # Vertices (n), # Edges (m), average (davg) and max (dmax) vertex degrees, and approximate diameter ($\widetilde{D}$) are listed. The bottom* 10 *graphs are all web crawls, while the top* 5 *are of various types.* $B = \times 10^9$, $M = \times 10^6$, $K = \times 10^3$.

Network	n	m	$davg$	$dmax$	$\widetilde{D}$
enron	68 K	253 K	7.5	1.6 K	11
dblp	226 K	716 K	6.3	238	17
amazon	735 K	3.5 M	9.6	1.1 K	23
ljournal	5.4 M	50 M	18	19 K	28
hollywood	1.1 M	56 M	105	11 K	10
cnr	326 K	2.7 M	17	18 K	35
in	1.4 M	131 M	19	22 K	44
indochina	7.3 M	149 M	41	256 K	27
eu	862 K	161 M	37	69 K	22
uk-2002	18 M	261 M	28	195 K	28
arabic	23 M	552 M	49	576 K	48
uk-2005	39 M	781 M	40	1.8 M	21
webbase	113 M	845 M	15	816 K	375
it	41 M	1.0 B	50	1.3 M	26
sk	51 M	1.8 B	72	8.6 M	308

4.2. Performance evaluation. We evaluate our PuLP partitioning methodology against both single- and multiconstraint METIS (v5.1.0) and ParMETIS (v4.0.3) as well as KaFFPa from KaHIP (v0.71b). METIS runs used k-way partitioning with sorted heavy-edge matching and minimized edge cut. KaFFPa results use the *fast-social* option (KaFFPa-FS), which does constrained label propagation during the initial graph contraction phase. KaFFPa allows for constraints on either vertices or edges; however, using multiple constraints concurrently is not possible with the current version. METIS allows for multiconstraint partitioning (METIS-M in the results). KaFFPa was unable to process the sk graph due to a 32-bit `int` limitation. A number of experiments were unable to complete for METIS and ParMETIS due to memory limitations.

We use the three aforementioned variants of PuLP for comparison: single-constraint single-objective (PuLP), multiconstraint single-objective (PuLP-M), and multiconstraint multiobjective (PuLP-MM). We do comparisons on the basis of edge cut, maximal per-part edge cut, execution time, and memory utilization. For all experiments on the LAW networks, the vertex imbalance ratio is set to 10%. For multiconstraint experiments, the maximal number of edges perpart for each graph is set to the maximum of either 10% imbalance or 4× the number of edges of the highest degree vertex of the graph. The rationale for loosening the edge balance constraints is due to the existence of very high degree vertices in certain graphs, which make a fixed constraint impossible to achieve for higher part counts. We set a relatively high vertex imbalance due to how the work requirements for a lot of graph computations are partially dependent on per-part edges, so achieving a better edge balance at a small cost to vertex imbalance can result in better overall work balance in practice.

4.3. Execution time and memory utilization. We first compare PuLP-MM to METIS-M, ParMETIS, and KaFFPa-FS in terms of partitioning times and memory utilization. Table 3 gives the serial execution times for computing 32 parts with PuLP-MM, METIS-M, and KaFFPa-FS, as well as the parallel execution times for PuLP-MM and ParMETIS. We also give the speedup of serial and parallel PuLP-

Table 3

*Comparison of execution time of serial and parallel (*16 *cores)* PuLP*-MM algorithm with serial METIS-M, KaFFPa-FS, ParMETIS (best of* 1 *to* 256 *cores), computing* 32 *parts. The "All" speedup compares parallel* PuLP*-MM to the best of the rest.*

Network	**Execution time (s)**					**PuLP-MM Speedup**	
	Serial			Parallel		vs. Best	All
	PuLP-MM	METIS-M	KaFFPa-FS	PuLP-MM	ParMETIS	Serial	Parallel
enron	0.38	0.72	1.18	0.22	0.32	1.89×	1.45×
dblp	0.67	1.23	3.18	0.23	0.34	1.84×	1.48×
amazon	2.70	3.97	11.7	0.54	0.77	1.47×	1.43×
hollywood	41.5	47.9	189	3.88	22.0	1.15×	5.67×
ljournal	119	116	206	12.2	47.6	0.97×	3.90×
cnr	1.56	1.63	2.15	0.44	3.11	1.04×	3.73×
in	6.17	2.9	7.03	0.89	2.6	0.47×	2.92×
indochina	75.7	60.2	52.3	9.74	120	0.69×	5.37×
eu	8.22	6.82	12.0	1.23	9.93	0.83×	5.54×
uk-2002	82.9	67.6	128	7.65	77.1	0.82×	8.84×
arabic	147	129	185	13.4		0.88×	9.63×
uk-2005	336		439	34.9		1.31×	12.6 ×
webbase	521		831	45.2		1.60×	18.4 ×
it	364		409	28.6		1.12×	14.3 ×
sk	644			54.6			

MM relative to the fastest serial code as well as the fastest code overall. Note that the execution time given for ParMETIS for each graph was selected as the fastest from 1 to 16 nodes on Compton with 1 to 16 tasks per node (up to 256 total cores). In effect, the speedups we give to PuLP-MM relative to ParMETIS are extremely conservative. Also note that we are comparing against one of the "fast" variants in KaFFPa.

Overall, we observe a geometric mean speedup of 1.07× relative to the next fastest of METIS-M and KaFFPa-FS for our serial code across the entire set of LAW graphs. This relatively modest speedup in serial comes from the fact that although all PuLP variants run in $O(np + m)$, that order of work is performed on a per-iteration basis for up to the 90 iterations of PuLP-MM. This gives a large initial constant factor for execution time. The primary benefits for a single level label propagation-based approach come from the ease of and efficiency of parallelization as well as the low memory overhead. For our parallel code, we note a mean speedup of 5.0×. Parallel speedups are also observed to increase with increasing graph size, likely a result of the improved scalability of label propagation versus the multilevel approaches.

Table 4 compares the maximal memory utilization of PuLP-MM, METIS-M, and KaFFPa-FS for computing 32 parts. Memory savings for PuLP-MM versus the best of either METIS-M or KaFFPa-FS are significant (3.0× geometric mean). We note an increase in memory savings with increasing graph size. These memory savings are primarily due to avoiding a multilevel approach. The only structures that the PuLP variants need (in 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 3.

We also plot the scalability of our PuLP codes versus METIS-M, ParMETIS, and KaFFPa-FS relative to increasing the parallelization and increasing the number

Table 4
PuLP *efficiency: Maximum memory utilization comparisons for generating* 32 *parts.*

Network	Memory Utilization				**Improv.**
	METIS-M	KaFFPa-FS	PuLP-MM	Graph Size	
enron	50 MB	33 MB	66 MB	2.5 MB	0.5×
dblp	91 MB	83 MB	67 MB	7.4 MB	1.2×
amazon	482 MB	385 MB	106 MB	33 MB	3.6×
ljournal	10 GB	4.8 GB	616 MB	429 MB	7.8×
hollywood	7.5 GB	3.8 GB	534 MB	448 MB	7.0×
cnr	285 MB	143 MB	105 MB	24 MB	1.4×
in	1.2 GB	537 MB	208 MB	113 MB	2.6×
indochina	20 GB	5.3 GB	1.4 GB	1.2 GB	3.8×
eu	1.6 GB	786 MB	217 MB	133 MB	3.6×
uk-2002	23 GB	9.8 GB	2.6 GB	2.2 GB	3.8×
arabic	33 GB	19 GB	5.1 GB	4.5 GB	3.7×
uk-2005	-	30 GB	7.4 GB	6.4 GB	4.1×
webbase	-	38 GB	9.8 GB	7.5 GB	3.9×
it	-	35 GB	9.4 GB	8.3 GB	3.7×
sk	-	-	16 GB	15 GB	-

of parts being computed. We analyze the effects of each on execution time and memory consumption on four selected test graphs (enron, hollywood, ljournal, and uk-2002), as given by Figure 1. The top two plots give the effect on execution time and memory consumption when going from 1- to 16-way parallelism. For this test, ParMETIS was run with 16 tasks on either 1 node (when possible) or 16 nodes, with the lower execution time being reported. The times and memory for serial METIS-M and KaFFPa-FS are plotted as flat lines for comparison. The bottom two plots of Figure 1 show the effect on execution time and memory consumption when increasing the number of parts being computed from 2 to 1024 when running in serial for METIS-M and KaFFPa-FS and with 16-way parallelism with the PuLP codes and ParMETIS.

We observe consistent strong scaling across all of our implementations in Figure 1, with a geometric mean speedup of 5.7× for 16 cores across all graphs and the three PuLP algorithms. We note our code strong scales better on these instances than ParMETIS, which has a maximum speedup of less than 2×. We also note that our memory requirements increase minimally with an increasing numbers of threads/cores, while the memory load of ParMETIS increases almost linearly with number of tasks. Throughout these tests, we also noted that there is no edge cut cost for increasing parallelism, at least up to 16 cores. This is because all per-vertex update decisions being made by each thread use globally synchronized information.

When computing an increasing number of parts, we note that our execution time plots are generally flat from 2 parts up to about 128 parts, where times begin to increase. We note a rather large increase for all codes when partitioning enron. Analyzing the computation of the PuLP algorithms, it appears the likely cause is the increasing difficulty to create balanced partitions with larger part counts, as enron is one of the smallest test cases. The bottom plot of Figure 1 gives the memory requirements versus part count for the various partitioners. For the PuLP and METIS variants, the curves are close to flat. For KaFFPa, we note a slight dependence of memory requirement on part count.

4.4. Edge cut and maximal per-part edge cut. Figure 2 (top) compares the quality of the computed partitions from PuLP-M and PuLP-MM to METIS with

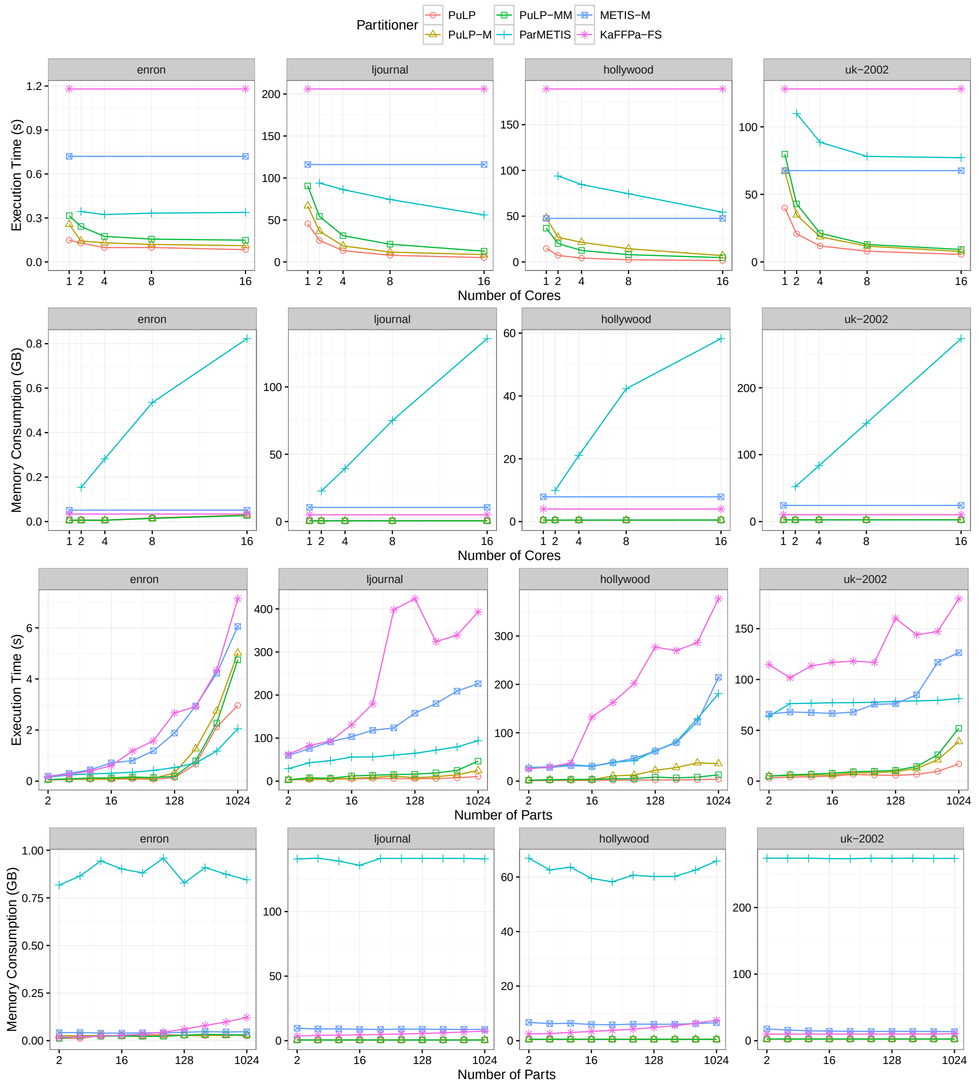

FIG. 1. *Scaling for each partitioner in terms of execution time versus number of cores (top), total memory utilization versus number of cores (second from top), execution time versus number of computed parts (third from top), and memory utilization versus number of computer parts (bottom).*

the LAW test graphs for 2 to 1024 parts using multiple constraints for both programs and minimizing the total edge cut. We report the median value obtained over 5 experiments for each part count and method. We omit comparison to KaFFPa-FS as it is unable to generate partitions that satisfy the multiple concurrent constraints.

The top plots of Figure 2 show the edge cuts (EC) obtained for multiconstraint METIS (METIS-M) as well as both multiconstraint (PuLP-M) and multiconstraint multiobjective PuLP (PuLP-MM) with varying numbers of parts. Figure 2 (bottom) gives the maximal per-part edge cut (EC_{max}) as a ratio of total edges scaled by the number of parts (scaling was done for visualization purposes). For all plots, a lower value indicates a higher-quality partitioning. We additionally show Table 5, where we quantify a performance metric in terms of edge cut (and max per-part

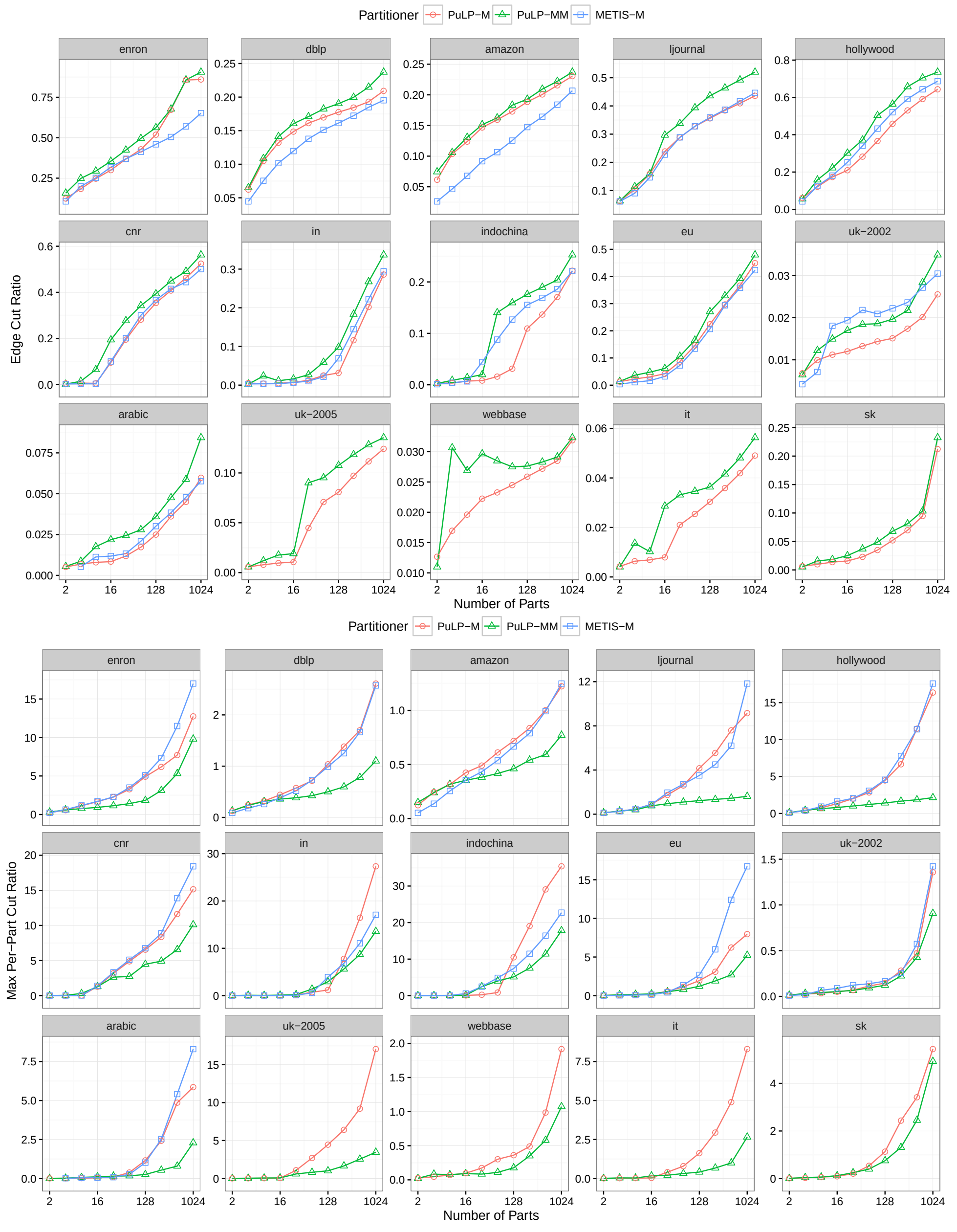

FIG. 2. *Quality metrics of total cut edge ratio (top) and scaled maximum per-part edge cut ratio (bottom) for* PuLP*-M,* PuLP*-MM, and METIS-M.*

cut, in the three right columns) divided by the best edge cut (and max per-part cut) of all partitioners for each "network-number of parts" combination. For each partitioner, we report the geometric mean for each graph across all numbers of parts. We also report an overall mean across all graph and number of parts combinations (All), as well as the overall mean for only the smaller graphs that METIS-M was able to successfully partition (Small). Again for this metric, lower is better, with

TABLE 5

Performance for each partitioner and graph as the geometric mean of the ratio of produced edge cut (EC) and max per-part cut (EC_{max}) relative to the best for across each network and number of generated parts.

Network	Edge Cut			Max Per-Part Cut		
	PuLP-M	PuLP-MM	METIS-M	PuLP-M	PuLP-MM	METIS-M
enron	1.08	1.26	**1.02**	1.72	**1.06**	1.59
dblp	1.18	1.27	**1.00**	1.68	**1.09**	1.48
amazon	1.52	1.59	**1.00**	1.54	**1.20**	1.23
ljournal	1.03	1.17	**1.00**	2.05	**1.01**	2.15
hollywood	**1.04**	1.28	1.11	2.44	**1.05**	2.56
cnr	1.28	1.86	**1.07**	1.63	1.55	**1.46**
in	1.17	2.22	**1.15**	1.46	2.06	**1.23**
indochina	**1.22**	2.29	1.76	**1.55**	2.05	2.26
eu	1.39	1.72	**1.00**	1.73	**1.47**	1.68
uk-2002	**1.08**	1.40	1.35	1.21	**1.12**	1.41
arabic	**1.03**	1.62	1.16	1.97	**1.35**	2.31
uk-2005	**1.00**	1.39	-	2.12	**1.15**	-
webbase	**1.02**	1.20	-	1.48	**1.08**	-
it	**1.00**	1.46	-	2.05	**1.31**	-
sk	**1.02**	1.30	-	1.22	**1.09**	-
Small	1.17	1.57	**1.13**	1.71	**1.32**	1.71
All	**1.12**	1.50	-	1.70	**1.27**	-

a minimum score of 1.0 indicating that the partitioner with that score produced the best partitions for all test cases. Taken together, the top and bottom plots of Figure 2 and the left and right three columns of Table 5 demonstrate the trade-off offered by PuLP-M and PuLP-MM 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.

Overall, we observe in Figure 2 (top) and Table 5 that PuLP-M does better than METIS-M for hollywood, indochina, uk-2002, and arabic; almost as good as METIS-M for enron, ljournal, in; and worse on dblp, amazon, eu, and cnr in terms of total edge cuts. Over all graphs that METIS-M was able to partition, PuLP-M has a slightly worse edge cut on average. PuLP-MM does worse than METIS-M and PuLP-M in most instances for the edge cut metric but results in much better partitions in terms of the maximal per-part edge cut on all but three test graphs (cnr, in, and indochina), as shown in Figure 2 (bottom) and the right three columns of Table 5. When the single objective partitioners (METIS-M and PuLP-M) outperform multiobjective PuLP-MM, it can be explained by the fact that the lower total edge cut for the single-objective partitioners effectively results in a lower max cut, even if the cuts aren't as relatively balanced as they are with PuLP-MM. We also observe that the benefit of max per-part minimization increases with an increasing number of parts. A consistent improvement over METIS-M and PuLP-M is noted with part counts greater than about 16.

As mentioned, Figure 2 and Table 5 demonstrate that multiobjective PuLP-MM can be relatively effective at minimizing the maximal per-part edge cut on partitions derived from these graphs at a nominal cost to total edge cut. Table 6 shows this trade-off explicitly between PuLP-MM and METIS-M when partitioning each graph into 512 parts. We compare the quality of both the metrics, EC and EC_{max}, and observe that PuLP-MM improves EC_{max} substantially (up to almost 600% improvement) when compared with METIS-M. This is at a cost of only a 4%–34% increase in total

TABLE 6

Comparison of the two quality metrics, EC and EC_{max} for PuLP*-MM and METIS-M when computing* 512 *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}
enron	217 K	2.6 K	144 K	5.7 K	-34%	116%
dblp	154 K	1.1 K	132 K	2.3 K	-14%	113%
amazon	783 K	4.1 K	648 K	6.8 K	-17%	68%
ljournal	24 M	140 K	21 M	600 M	-15%	328%
hollywood	40 M	205 K	36 M	1.3 M	-9%	513%
cnr	1.3 M	35 K	1.2 M	74 K	-10%	112%
in	3.5 M	223 K	2.9 M	283 K	-17%	27%
indochina	30 M	3.3 M	28 M	4.8 M	-9%	44%
eu	6.3 M	85 K	5.8 M	391 K	-9%	361%
uk-2002	7.4 M	218 K	7.1 M	293 M	-4%	34%
arabic	32 M	860 K	26 M	5.8 M	-19%	579%
uk-2005	100 M	3.9 M	-	-	-	-
webbase	25 M	957 K	-	-	-	-
it	49 M	2.0 M	-	-	-	-
sk	187 M	8.7 M	-	-	-	-

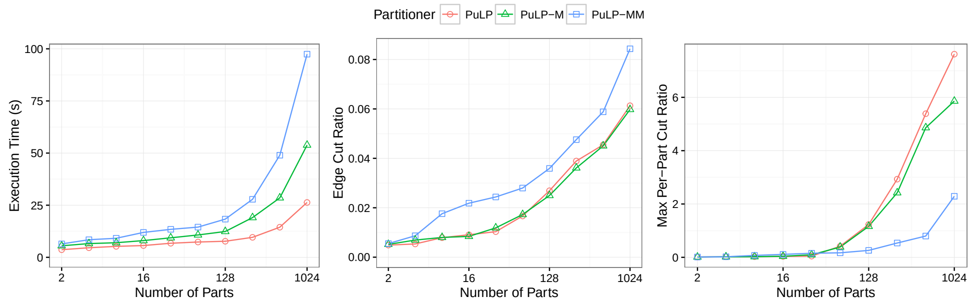

FIG. 3. *Comparison of* PuLP*,* PuLP*-M, and* PuLP*-MM with regards to execution time, edge cut, and max per-part cut to demonstrate the effects of more complex constraints and objectives on execution.*

edge cut. Note that while we have been using METIS-M for comparison, it does not explicitly attempt to minimize edge cut balance.

Additionally, we present Figure 3, which shows for the ljournal graph the differences in execution time (left), edge cut (middle), and max per-part cut (right) for computing 2 to 1024 parts with PuLP, PuLP-M, and PuLP-MM. These plots demonstrate the effects on performance and quality with the increasing complexity of objectives and constraints on a representative test instance. In general, the multiconstraint PuLP-M and PuLP-MM run at least 2× slower than PuLP, as they perform twice as many total label propagation iterations due to the secondary constraint balancing and refinement stages. Although PuLP-MM performs the same number of total iterations as PuLP-M, we see another relative doubling of execution time. This is a result of the increasing difficulty involved with the secondary (max per-part cut) objective, which consistently results in more work as a result of more active boundary vertices in the queue on each iteration.

For the edge cut objective, we note a similar difference between PuLP-M and PuLP-MM as observed before. PuLP-MM has a slightly worse global edge cut as

TABLE 7
Comparison of the multiple variants of algorithmic choices on quality in terms of edge cut (top) and max per-part cut (bottom) relative to PuLP-*MM.*

Percent improvement for edge cut

Network	# Parts	BigData	NoRefine	NoDegWgt	FewerIter	MoreIter
enron	16	-1%	-33%	-2%	-7%	3%
amazon	32	-38%	-37%	-3%	-5%	6%
uk-2002	8	-19%	-214%	-46%	-50%	5%
it	256	-1%	-51%	-15%	-3%	3%
Global	2-1024	-6%	-64%	-17%	-8%	3%

Percent improvement for max per-part cut

Network	# Parts	BigData	NoRefine	NoDegWgt	FewerIter	MoreIter
enron	16	3%	-7%	8%	-8%	6%
amazon	32	-20%	-21%	-3%	-6%	4%
uk-2002	8	-5%	-180%	-43%	-104%	7%
it	256	0%	-1%	-1%	-7%	4%
Global	2-1024	-1%	-37%	-13%	-15%	5%

a result of the trade-off involved with additionally optimizing the max per-part cut. PuLP and PuLP-M generally show equivalent edge cuts, as they're both only optimizing for this metric, and therefore both asymptotically approach what can be considered a relative "lower bound" for the general approach. As we'll show in the next subsection, performing considerably more iterations can improve the objective, but only by a few percent at most. In terms of the max per-part cut as given by the right plot, PuLP-MM shows considerably better quality, as expected, with the relative performance improving with increasing part count. PuLP-M shows moderately better quality for PuLP with this objective, most likely due to a slight increase in cut balance incidentally resulting from balancing for the secondary (edges per part) constraint.

4.5. Justification for algorithmic choices. There were several small algorithmic details noted in the earlier description of the three PuLP algorithms. We ran a set of tests examining the impact of these details and variations on partition quality. Table 7 gives the percentage improvement in quality relative to PuLP-MM with these variations. Tested variants include using the original BigData 2014 [29] random label propagation-based initialization procedure (*BigData*), not running any refinement (*NoRefine*), not using degree weighted label propagation during the vertex balance stage (*NoDegWgt*), running only a single iteration for the outer loops instead of three (*FewerIter*), and running 10 iterations of the outer loops (*MoreIter*). We look at the global averages across all LAW networks and part counts as well as a few select representative instances. A negative value indicates lower quality was produced with the given variant. We ran each "number of parts-network" test case five times and report the geometric mean.

From Table 7, we note that running refinement has the largest and most consistent impact on partition quality out of the tested variants. On average, not performing refinement resulted in 64% higher edge cuts and 37% higher max per-part cuts. Degree-weighted label propagation for the vertex balancing stage is especially important for the irregular web crawls (uk-2002 and it) but shows little quality benefit on the other network types (enron communication network and amazon co-purchasing network). As mentioned, this is because the benefits of weighted label propagation are partially dependent on the existence of a skewed degree distribution and an intrinsic community structure.

Table 7 also gives insight into our selected number of outer loop iterations. We observed that three iterations is approximately the point at which diminishing returns on partition quality become realized for most test instances. This can be noted by comparing the edge cut and max per-part cut differences to baseline in the *FewerIter* and *MoreIter* columns. While the difference between a single iteration and the baseline three iterations (*FewerIter*) can result in a quality improvement of over 100% on certain test instances, running up to 10 iterations instead of just three iterations (*MoreIter*) improves quality only by a few additional percent age points at most. Although Table 7 shows that running more total outer loop iterations improves partition quality for our test set by 3% and 5% on average for edge cut and max per-part cut, respectively, the considerably higher computation cost doesn't necessarily justify such a minor improvement.

To further demonstrate the effect of iterations on various metrics, we also plot how the edge cut, max per-part cut, vertex imbalance, and edge imbalance changes on a per-iteration basis for a run of PuLP-MM computing 64 parts on the amazon network. We give these plots in Figure 4. We use the baseline iteration counts from Table 1, so there are 3 iterations for each outer loop with each balancing stage running for 5 iterations and each refinement stage running for 10 iterations, resulting in 90 total label propagation iterations.

The thin vertical lines indicate a switch from the balancing to refinement stages, and the thicker vertical lines indicate the beginning of both the vertex balancing/refinement outer loop and the edge and max per-part cut balancing/refinement outer loop as given by Algorithm 2. The horizontal lines on the bottom two plots indicate the given 10% imbalance constraints for vertices and edges.

We consider the plots in Figure 4 to be a representative instance for our test networks. In general, we observe that the vertex balance is achieved in few iterations on most networks. Edge balance is slower to achieve, although this is more highly dependent on the given imbalance ratio and the topology of the network. We note a modest trade-off in total edge cut and max per-part cut as the edge balance constraint is reached. As was observed by examining the sum affects of greater and fewer iterations with the Global row in Table 7, we observe the edge cut and max per-part cut are within a modest fraction of their minimal values after a single balancing and refinement stage. We also note consistent improvements in edge cut through the refinement iterations, although these improvements are slight enough to not be easily visible with the given scale of the plots. With a looser constraint, we observe correspondingly lower edge cuts and max per-part cuts.

4.6. Rebalancing single-constraint single-objective partitions. Another benefit to using a single-level label propagation-based partitioner is the ease with which it can be used on a given input partition to rebalance it for additional objectives and constraints. To demonstrate, we used KaFFPa-FS to compute high-quality single-objective and single-constraint partitions for all of the LAW graphs across 2–1024 parts. We then used the computed partitions as inputs to PuLP-MM's second loop, the edge balancing and max per-part cut minimization stage. Figure 5 gives plots of amazon (top) and webbase (bottom) for edge cut (left) and max per-part cut (right) versus number of parts.

From Figure 5, we can observe considerable improvement in terms of the edge cut metric upon our baseline PuLP-MM partition when first using a high-quality single-

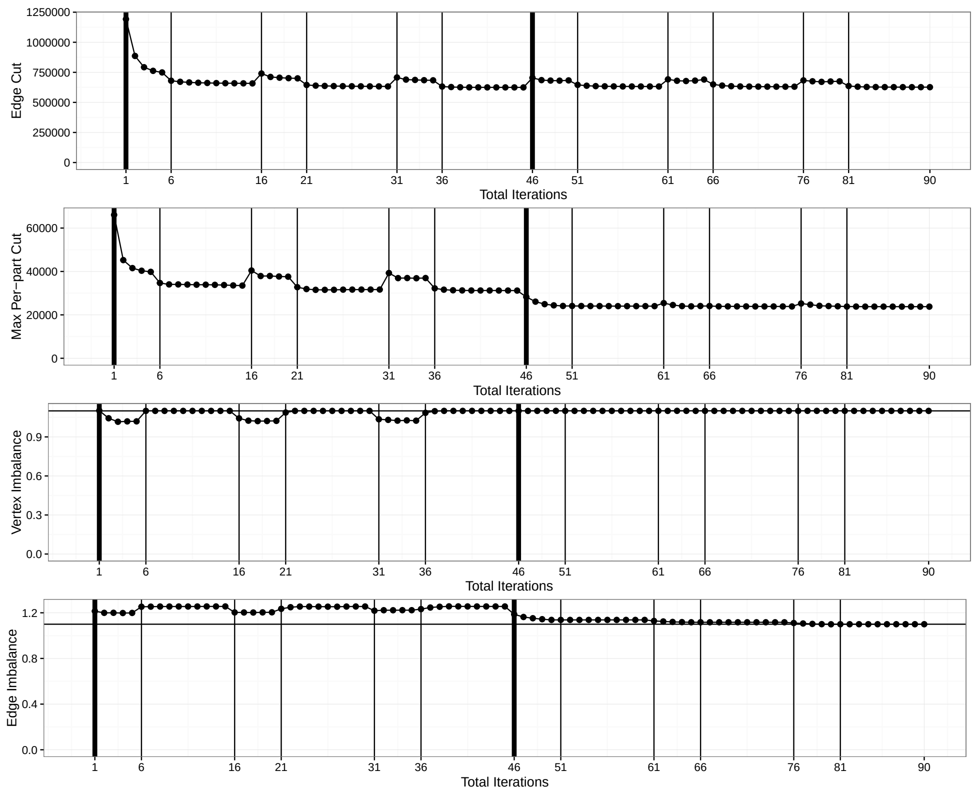

FIG. 4. *Per-iteration performance of* PuLP*-MM in terms of total edge cut, max per-part edge cut, vertex imbalance, and edge imbalance when computing* 64 *parts of amazon. The left side of the figures shows the vertex balancing stage, while the right side of the figures shows the edge balancing phase.*

objective and single-constraint partitioning. Additional improvements are noted with the max per-part cut objective as well. Note that while the edge cut is higher than just KaFFPa-FS, the KaFFPa-FS partitions violate the given balance constraints. Overall, we calculated a geometric mean improvement relative to just running PuLP-MM of 22% for edge cut and 17% for max per-part cut across all tested graphs and numbers of parts. Although we don't explicitly perform such an analysis, this approach is also directly applicable to using one of the PuLP variants to repartition/rebalance a dynamically changing graph.

4.7. DIMACS 10th Implementation Challenge comparison. While PuLP-M and PuLP-MM were specifically designed for irregular small-world graphs and the multiobjective multiconstraint scenario, we also observe relatively good performance for PuLP computing single-objective single-constraint partitioning of regular networks. To examine the performance of PuLP relative to the state of the art, we used PuLP to partition all 90 instances for the DIMACS10 test suite and compared results to both KaFFPa, with both *fast* (KaFFPa-F) and *fastsocial* (KaFFPa-FS) variants, and METIS. We only run the initialization and vertex balancing/refinement stages with PuLP. We again calculated a performance ratio as reported for the LAW graphs in Table 5, and we report the overall geometric mean across the 90 DIMACS10 test instances. Our tests showed METIS to be the highest scoring partitioner with a ratio

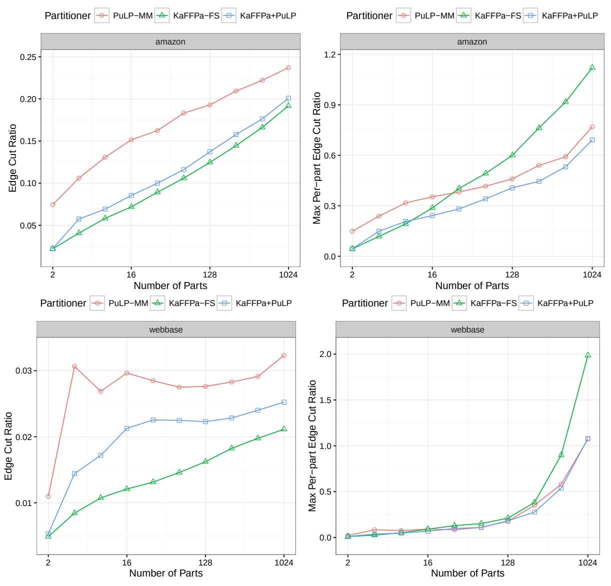

FIG. 5. *Using* PuLP-*MM to repartition single-objective single-constraint partitions computed with KaFFPa-FS. Shown are graphs amazon (top) and webbase (bottom) plotted with edge cut (left) and max per-part cut (right) versus number of parts.*

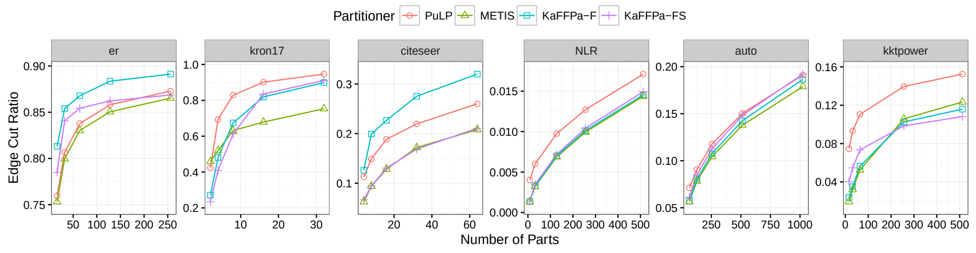

FIG. 6. *Edge cut versus number of parts for a few select representative instances from the* 10th *DIMACS Implementation Challenge for* PuLP, *METIS, KaFFPa-fast, and KaFFPa-fastsocial.*

of 1.005. KaFFPa-FS had a ratio of 1.053, KaFFPa-F had a ratio of 1.14, and PuLP had a ratio of 1.43. This indicates that although PuLP was designed for very different partitioning circumstances, the general approach is adaptable enough to still perform within a small factor of the state of the art for other scenarios.

Figure 6 gives the edge cut ratio versus number of parts for PuLP, METIS, KaFFPa-F, and KaFFPa-FS on a few test instances from the DIMACS Implementation Challenge. The number of parts chosen for each graph instance were those used in the DIMACS challenge and are select multiples of 2 from 2 to 1024 as can be discerned in the plotted results. We observe that PuLP is usually within a small factor of the best partitioner and is sometimes better than at least one of the other partitioners. Note that the test graphs here are mostly regular or semiregular and lack inherent community structure. So despite the fact that PuLP was designed for a drastically different graph structure, the surprising quality of the results shown here demonstrate again just how versatile the general PuLP algorithmic approach can be for graph partitioning.

5. Related work. There are a few 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 publicly available. Most recently, Meyerhenke, Sanders, and Sehulz [22] utilized their sized-constrained label propagation technique in a multilevel implementation for both graph coarsening and local optimization [21] in a distributed setting. This approach was combined with an evolutionary partitioning algorithm to produce fast and high-quality single-objective and single-constraint partitions. On a 32 core machine with 512 GB memory, they report execution times and edge cuts for bipartitioning several graphs from the DIMACS10 and LAW collections. We directly compare to the largest instances from each, sk-2005 and uk-2007. On sk-2005, [22] reports execution times and edge cuts of 471 seconds with 3.2 M cut edges and 1688 seconds with 2.9 M cut edges for their *Fast* and *Eco* variants, respectively. On uk-2007, they report 170 seconds with 1.0 M cut edges and 723 seconds with 981 K cut edges, respectively. Running single-objective single-constraint PuLP on these graphs with the same balance constraint (3%), we return a cut of 6.3 M in 20 seconds on sk-2005 and a cut of 1.5 M in 30 seconds on uk-2007 on a 16 core and 64 GB node of *Compton*. Note, however, that PuLP generally demonstrates better relative performance with increasing part counts and in the multiple constraint scenario.

Vaquero et al. [34] implement vertex-balanced label propagation to partition dynamically changing graphs. Martella et al. improve upon this general approach with Spinner [20]. They report that Spinner produces a 0.45 edge cut ratio for an edge-balanced 64-way partitioning of the SNAP LiveJournal graph [19, 31]. By comparison, PuLP produces an edge cut ratio of about 0.35 under an equivalent constraint.

Ugander and Backstrom [33] implement label propagation for vertex-balanced partitioning as an optimization problem. They report performance on the 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 multiconstraint and multiobjective serial code creates 128 parts of the LiveJournal graph in about 2 minutes and produces an edge cut ratio of about 0.41.

Wang et al. [35] utilize label propagation in a manner similar to KaFFPa-FS, 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 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 nonmultilevel approach runs in about half the time, but at a considerable cost to cut quality.

6. Conclusions and future work. In this paper, we presented PuLP, an approach for scalable multiobjective multiconstraint partitioning of irregular small-world networks. The partitioning method in PuLP is based on the label propagation community detection algorithm. Our PuLP algorithms all run in a fraction of the execution time while consuming an order of magnitude less memory than the k-way multilevel partitioning scheme in METIS. On irregular small-world networks, our single-objective PuLP-M implementation can produce partition quality comparable to METIS in terms of total edge cut, while our multiobjective PuLP-MM implementation shows consistent relative improvement against METIS in the secondary max per-part cut objective. In future work, we will demonstrate the effectiveness of partitionings produced from PuLP on a broad set of graph computations, as well as implement PuLP in a distributed setting to partition graphs too large to fit in the main memory of a single compute node.

REFERENCES

[1] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner, *Benchmarking for graph clustering and partitioning*, in Encyclopedia of Social Network Analysis and Mining, Springer, NY, 2014, pp. 73–82.

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

[3] P. Boldi, M. Rosa, M. Santini, and S. Vigna, *Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks*, in Proceedings of the 20th International Conference on World Wide Web, ACM Press, New York, 2011.

[4] P. Boldi and S. Vigna, *The WebGraph framework I: Compression techniques*, in Proceedings of the 13th International World Wide Web Conference (WWW 2004), ACM Press, New York, 2004. pp. 595–601.

[5] E. G. Boman, K. D. Devine, and S. Rajamanickam, *Scalable matrix computations on large scale-free graphs using 2D graph partitioning*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013.

[6] Ü. V. Catalyürek, M. Deveci, K. Kaya, and B. Uçar, *UMPa: A multi-objective, multi-level partitioner for communication minimization*, Contemp. Math., 588 (2013).

[7] A. Ching and C. Kunz, *Giraph: Large-scale graph processing infrastructure on Hadoop*, in Proceedings of the Hadoop Summit, 2011, p. 2011.

[8] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1–25.

[9] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in Proceedings of the Conference on Design Automation, 1982.

[10] A. George and J. W. Liu, *Computer Solutions of Large Sparse Positive Definite* Systems, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, *PowerGraph: Distributed graph-parallel computation on natural graphs*, in Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI), 2012.

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

[13] U. Kang, C. E. Tsourakakis, and C. Faloutsos, *PEGASUS: A peta-scale graph mining system implementation and observations*, in Proceedings of the IEEE International Conference on Data Mining (ICDM), 2009.

[14] 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 (2014).

[15] G. Karypis and V. Kumar, *Parallel multilevel K-way partitioning scheme for irregular graphs*, in Proceedings of the ACM/IEEE Conference on Supercomputing, 1996.

[16] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[17] G. Karypis and V. Kumar, *Multilevel algorithms for multi-constraint graph partitioning*, in Proceedings of the ACM/IEEE Conference on Supercomputing, 1998.

[18] J. Kunegis, *KONECT — The Koblenz Network Collection*, konect.uni-koblenz.de (2014).

[19] 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 Math., 6 (2009), pp. 29–123.

[20] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, *Spinner: Scalable Graph Partitioning in the Cloud*, preprint, arXiv:1404.3861, 2014.

[21] H. Meyerhenke, P. Sanders, and C. Schulz, *Partitioning complex networks via size-constrained clustering*, in Proceedings of Experimental Algorithms — 13th International Symposium, Copenhagen, Denmark, 2014, pp. 351–363.

[22] H. Meyerhenke, P. Sanders, and C. Schulz, *Parallel graph partitioning for complex networks*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, 2015, pp. 1055–1064.

[23] R. Pearce, M. Gokhale, and N. M. Amato, *Scaling techniques for massive scale-free graphs in distributed (external) memory*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2013.

[24] A. Pinar and B. Hendrickson, *Partitioning for complex objectives*, in Proceedings of the 15th International Parallel and Distributed Processing Symposium, IPDPS '01, Washington, DC, 2001.

[25] L. Quick, P. Wilkinson, and D. Hardcastle, *Using Pregel-like large scale graph processing frameworks for social network analysis*, in Proceedings of the International Conference 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*, Phys. Rev. E, 76 (2007), 036106.

[27] K. Schloegel, G. Karypis, and V. Kumar, *Parallel multilevel algorithms for multi-constraint graph partitioning*, in Proceedings of Euro-Par 2000, International European Conference on Parallel and Distributed Computing, 2000.

[28] B. Shao, H. Wang, and Y. Li, *Trinity: A distributed graph engine on a memory cloud*, in Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2013.

[29] G. M. Slota, K. Madduri, and S. Rajamanickam, *PULP: Scalable multi-objective multi-constraint partitioning for small-world networks*, in Proceedings of the IEEE Conference on Big Data (BigData 2014), 2014.

[30] G. M. Slota, S. Rajamanickam, and K. Madduri, *BFS and coloring-based parallel algorithms for strongly connected components and related problems*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposiom (IPDPS), 2014.

[31] *Stanford Large Network Dataset Collection*, http://snap.stanford.edu/data/index.html (2014).

[32] B. Uçar and C. Aykanat, *Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies*, SIAM J. Sci. Comput., 25 (2004), pp. 1837–1859.

[33] J. Ugander and L. Backstrom, *Balanced label propagation for partitioning massive graphs*, in Proceedings of the Web Search and Data Mining (WSDM), 2013.

[34] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, *xDGP: A Dynamic Graph Processing System with Adaptive Partitioning*, CoRR abs/1309.1049, 2013.

[35] L. Wang, Y. Xiao, B. Shao, and H. Wang, *How to partition a billion-node graph*, in Proceedings of the IEEE 30th International Conference on Data Engineering, 2014, pp. 568–579.