Scalable, Multi-Constraint, Complex-Objective Graph Partitioning

George M. Slota, Cameron Root, Karen Devine, Kamesh Madduri, Sivasankaran Rajamanickam

Abstract—We introduce XTRAPULP, a distributed-memory graph partitioner designed to process irregular trillion-edge graphs. XTRAPULP is based on the scalable label propagation community detection technique, which has been demonstrated in various prior works as a viable means to produce high quality partitions of *skewed* and *small-world* graphs with minimal computation time. Our XTRAPULP implementation can also be generalized to compute partitions with an arbitrary number of constraints, and it can compute partitions with balanced communication load across all parts. On a collection of large sparse graphs, we show that XTRAPULP partitioning is considerably faster than state-of-the-art partitioning methods, while also demonstrating that XTRAPULP can produce partitions of real-world graphs with billion+ vertices and over a hundred billion edges in minutes. Additionally, we demonstrate XTRAPULP on a variety of applications, including large-scale graph analytics and sparse matrix-vector multiplication.

Index Terms—graph partitioning, load balancing, label propagation, graph analysis.

1 INTRODUCTION

XTRAPULP is our new graph partitioner exploiting MPI+OpenMP parallelism to efficiently partition extreme-scale real-world graphs. It can be considered a significant extension to our prior shared-memory-only partitioner, PULP [1]. We specifically target skewed and small-world graphs, though our methods are also directly applicable to more regular graphs such as meshes. Graph partitioning is an essential preprocessing step to ensure load-balanced computation and to reduce inter-node communication in parallel applications [2], [3]. With the sizes of online social networks, web graphs, and other non-traditional graph data (e.g., brain graphs) growing at an exponential pace, scalable and efficient algorithms are necessary to partition and analyze them. These networks are typically characterized by highly skewed vertex degree distributions and low average path lengths. Some of these graphs can be modeled using the "small-world" graph model [4], [5], and others are referred to as "power-law" graphs [6], [7].

For highly parallel distributed-memory graph analytics on billion+ vertex or trillion+ edge small-world graphs, any computational and communication imbalance can result in a significant performance loss. Thus, graph partitioning can be used to improve balance. While some (correctly) argue that shared-memory graph processing can be superior to distributed processing for simple analytics on graphs that fit in RAM or disc of a single machine [8], [9], extreme-scale networks [10], memory-intensive dynamic programming algorithms [11], and other computationally-complex analytical algorithms [12] still necessitate distributed processing models in a number of circumstances.

Traditional tools for partitioning graphs (e.g., ParMETIS [13], Scotch [14]), which were generally designed for scientific computing applications on regular meshes, are limited either in the size of the graphs they can partition or the partitioning objective metrics that they support. Further, as many analytics themselves are often quite fast in comparison to scientific computing applications, the time and scalability requirements for the effective use of a graph partitioner with graph processing applications is much stricter. In essence, partitioning methods targeting emerging graph analytics should be significantly faster than the current state-of-the-art, support multiple objective metrics, and scale better on large-scale irregularly structured inputs. We also desire the method to (1) be more memory-efficient than traditional partitioning methods (used in scientific computing); (2) have good strong-scaling performance, since we may work with fixed-size problems; (3) be relatively simple to implement, and (4) require little tuning.

There has been some progress made in the recent past towards such partitioning methods. There are methods using random or edge-based distributions [15], label propagation-based graph partitioning methods [16], [17], [18], adaptations of traditional partitioning methodologies to small-world graph instances [19], and moderate-scale methods for distributed graph frameworks [20]. Graph problems at the trillion edge scale have been attempted only recently [8], [10], [21], [22], with very recent work implementing edge-based partitioning at the trillion edge scale [23].

1.1 Contributions

We describe the implementation details for XTRAPULP, which enable us to solve the multiple constraint and multiple objective graph partition problem on tera-scale graphs.

- *G. M. Slota and C. Root are with the Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, 12180. E-mail: slotag@rpi.edu*
- *K. Devine and S. Rajamanickam are with the Scalable Algorithms Department, Sandia National Laboratories, Albuquerque, NM, 87123. E-mail: kddevin@sandia.gov, srajama@sandia.gov*
- *K. Madduri is with the Electrical Engineering and Computer Science Department, The Pennsylvania State University, University Park, PA, 16810. E-mail: madduri@cse.psu.edu*

Manuscript received June 10, 2020

We explicitly demonstrate this scalability by running on hundreds of thousands of cores to partition graph instances with 17 billion vertices and 1.1 trillion edges. When optimizing for the single constraint and single objective partitioning problem at the small scale, we compute partitions an order-of-magnitude faster than ParMETIS or ParHIP on average while maintaining an edge cut within a factor of 2× of these partitioners on average. When computing the multiple constraint and objective problem, we also consistently compute partitions faster than ParMETIS while outputting edge cuts within a factor of 5× on average.

We finally note that portions of this work have appeared in prior publication[1]. New contributions include a generalization of our algorithm to an arbitrary number of constraints and the associated experiments, updated scaling experiments, and a more comprehensive experimental analysis of using XTRAPULP to accelerate sparse matrix-vector multiplication. The code used in this paper is publicly available in our repository on GitHub[2].

1.2 Supplemental Material

Included with this article is a supplementary document. This document contains explicit algorithm listings containing low-level implementation details; execution time complexity discussion; additional experiments comparing XTRAPULP to ParMETIS, ParHIP, and Distributed Neighbor Expansion in regards to the single objective and single constraint partitioning problem; and experimental results that helped guide parametric tuning of our iteration count and balance control variables.

2 BACKGROUND

2.1 Partitioning Formulation

In this work, we specifically consider two formulations of the graph partitioning problem. The first instance considers a variant that has been specifically developed and empirically validated for complex distributed graph analytics [24]. We further demonstrate its efficacy in our results by accelerating distributed 2-dimensional sparse matrix-vector computation. In prior work and throughout this paper, we refer to this definition as the PULP-MM variant for distributed graph analytics. In addition, we also consider a generalized *horizontal* formulation for an arbitrary number of constraints. The formal definitions for these two formulations follow.

Distributed Graph Analytics Formulation: Given an undirected and unweighted graph $G = (V, E)$ and target max imbalance ratios Rat_v and Rat_e as a maximum allowed percentage imbalance of vertices and edges per part, our "distributed graph analytics" partitioning problem can be formally described as a vertex-disjoint partitioning of V into p parts. Suppose $\Pi = \{\pi_1, \ldots, \pi_p\}$ is some such partition. We wish to constrain Π such that $\forall\ i = 1 \ldots p$:

$$|V(\pi_i)| \leq (1 + Rat_v)\frac{|V|}{p}$$
$$|E(\pi_i)| \leq (1 + Rat_e)\frac{|E|}{p},$$

where $V(\pi_i)$ is the set of vertices in part π_i and $E(\pi_i)$ is the set of edges such that at least one of its endpoints are in part π_i. We define the global set of cut edges and the local set of cut edges relative to a part k as:

$$C(G, \Pi) = \{|\langle u, v\rangle \in E| \ :\ \Pi(u) \neq \Pi(v)\}$$
$$C(G, \pi_k) = \{|\langle u, v\rangle \in E(\pi_k)| \ :\ (u \notin \pi_k \vee v \notin \pi_k)\}.$$

Our partitioning problem is then to minimize the two metrics $|C(G, \Pi)|$ and $\max_{k \in 1 \ldots p} |C(G, \pi_k)|$ subject to the imbalance constraints given above.

Generalized Horizontal Multi-Weight Formulation: We also consider the *horizontal* multi-weight formulation [25], where each vertex has some number w of associated vertex weights. Here, we are given a weighted graph $G = (V, E, V_W, E_W)$, where V_W is a set of vertex weights and E_W is a set of edge weights. We specifically consider that V_W will contain exactly w weights for all $v \in V$ and E_W will contain a single weight for all $e \in E$. V_W and E_W can be considered as $|V| \times w$ dimensional matrices, where index (i, j) refers to the j^{th} weight of vertex i.

We are also given target max imbalances $Rat_j : \forall j = 1 \ldots w$, and we constrain Π such that $\forall\ i = 1 \ldots p$ and $\forall\ j = 1 \ldots w$:

$$|V_{W_j}(\pi_i)| \leq (1 + Rat_j)\frac{|V_{W_j}|}{p},$$

where $|V_{W_j}|$ is the sum of all vertex weights over all $v \in V$ for weight j. I.e., we have a specific target maximum imbalance for each individual weight that we wish to achieve on each part $\pi_i \in \Pi$. Our weighted cut and max per-part cut are:

$$C(G, \Pi) = \{\sum_{\langle u,v\rangle \in E} E_W(\langle u, v\rangle) \ :\ \Pi(u) \neq \Pi(v)\}$$
$$C(G, \pi_k) = \{\sum_{\langle u,v\rangle \in E(\pi_k)} E_W(\langle u, v\rangle) \ :\ (u \notin \pi_k \vee v \notin \pi_k)\},$$

where $E_W(\langle u, v\rangle)$ is the weight of edge $\langle u, v\rangle$. Our optimization goal is to minimize the weighted cut and max per-part cut subject to achieving our imbalance tolerance for every weight for every part as given above.

2.2 Label Propagation

The label propagation community detection algorithm [26] is a fast and scalable method for detecting communities in large networks. The algorithm works as follows: all vertices in a graph are initialized into their own communities, and on each iteration a vertex places itself in the community with a plurality of its neighbors, with ties broken randomly. Since it is fairly easy to parallelize and its optimization goal is similar to that of a balanced edge cut, label propagation has seen widespread adoption as an effective means to

1. ©2017 IEEE. Reprinted, with permission, from Slota et al., Partitioning Trillion-Edge Graphs in Minutes, IPDPS 2017.
2. https://github.com/HPCGraphAnalysis/PuLP/

find high quality partitions of small-world and irregular networks, such as social networks and web crawls. There are two primary approaches for using label propagation in partitioners.

The first approach uses label propagation as part of a multilevel framework, where label propagation is used in the coarsening stage. Partitioners that utilize these techniques include ParHIP [19] and Wang et al. [16]. Wang et al. demonstrated a case study of how label propagation might be used as part of a multilevel partitioner, by first coarsening the graph in parallel and then running METIS [27] at the coarsest level. Meyerhenke et al. (ParHIP) improved upon this approach in terms of partition quality and execution time by running an optimized implementation of distributed label propagation and then parallel runs of the evolutionary algorithm-based state-of-the-art KaFFPaE partitioner at the coarsest level. The biggest drawbacks to multilevel methods are the high memory requirements that result from having to store copies of the graph at the multiple levels of coarsening, the coarsening and uncoarsening processing overheads, and the scalability of the partitioning methods at the coarsest level.

The second approach uses label propagation directly to compute the partitions. Early efforts utilizing this approach include Ugander et al. [17] and Vaquero et al. [28]. Wang et al. [16] additionally used a variant of their coarsening scheme to compute balanced partitions, although at a non-negligible cost to cut quality. In our recent prior work, we introduced PuLP [1], [18], which uses weighted label propagation variants for various stages of a multi-constraint and multi-objective shared-memory parallel partitioning algorithm. Buurlage [29] extended our initial work with HYPER-PuLP, which modified the general PuLP scheme to the distributed partitioning of hypergraphs. Note that hypergraph partitioning requires a significantly different approach than graph partitioning, and it often is not as scalable. The graphs we consider are several orders of magnitude larger than those partitioned with HYPER-PuLP.

3 XtraPuLP Implementation Details

This section provides implementation details of XtraPuLP. For explicit algorithm listings, please see the supplementary document. Our primary contributions in this current work are both technical and algorithmic. We provide a discussion of the technical necessities to scale the prior PuLP algorithms to process graphs of several orders-of-magnitude larger and on several orders-of-magnitude more cores than the prior PuLP implementation is capable. We also demonstrate algorithmic improvements to our initialization approach and a novel algorithm for partitioning with an arbitrary number of vertex weights. We generally build upon distributed graph processing techniques and optimizations discussed in other prior work [30], but the three specific extensions needed for the distributed implementation of PuLP are:

- The graph and its vertices' part assignments and other associated data must be distributed in a memory-scalable way across processors. Only the necessary local per-rank information should be stored to reduce memory overhead. Access to rank-specific information should also be as efficient as possible for computational scalability in a large cluster. We develop and optimize our implementation to achieve these objectives.
- MPI-based communication is needed to update boundary part-assignment information and compute global quantities required by our weighting functions. We implement optimized communication routines to achieve scaling to thousands of nodes.
- The update pattern of part assignments must be finely controlled to prevent wild oscillations of part assignments as processes independently label their vertices. We develop a method for controlling part stability and demonstrate its effects on partition quality and balance.

3.1 XtraPuLP-MM Overview

We focus our initial discussion on the XtraPuLP-MM algorithm for multiple objective (minimizing the global cut and maximal cut edges of any part) and multiple constraint (vertex and edge balanced) partitioning. We specifically focus on this formulation as it can accelerate large-scale graph analytic computations [24] and even more regular computations such as matrix-vector multiplications (Sections 4.5 and 4.6).

Algorithm 1 XtraPuLP Overview:
$parts \leftarrow$ PuLP-MM$(G_g(V_g, E_g), p, I_{\text{out}}, I_{\text{bal}}, I_{\text{ref}}, Imb_v, Imb_e)$

```
G(V,E) ← distributeGraph(G_g(V_g,E_g))      ▷ read and distribute input graph to MPI ranks
I_tot ← I_out × (I_bal + I_ref)             ▷ total per-phase iteration count
iter_tot ← 0                                ▷ running per-phase iteration count
parts ← XtraPuLP-Init(...)
for i = 1 ... I_out do
    parts ← XtraPuLP-VertBalance(...)
    parts ← XtraPuLP-VertRefine(...)
iter_tot ← 0                                ▷ reset for next phase
for i = 1 ... I_out do
    parts ← XtraPuLP-EdgeBalance(...)
    parts ← XtraPuLP-EdgeRefine(...)
```

We give the XtraPuLP-MM algorithm in Algorithm 1. As the primary input for this algorithm, we have the unrealized global view of the graph as $G_g(V_g, E_g)$. Each MPI rank gets assigned a local graph representation of $G(V, E)$ which includes some disjoint partition of vertices and their one-hop neighborhood of edges (discussed later in Section 3.6). Additional inputs to this algorithm include the number of parts (p), and the target imbalance constraints and iteration counts, discussed below. This and the following algorithmic descriptions are given from the perspective of a single rank.

There are three stages to the algorithm. The first stage is a fast initialization strategy which allows some imbalance among parts (XtraPuLP-Init). It can be considered a hybrid between the two shared-memory PuLP initialization strategies of unconstrained label propagation [1] and breadth-first search-based graph growing [18], [31], [32]. Specifically, we perform a level-synchronized breadth-first search from a given set of p seeds, and any contention between parts for a vertex v is resolved through a random draw weighted by the number of neighbors v has in each part. The algorithm is described in further detail in [33] and the supplemental information for this paper.

The second stage balances the number of vertices for each part while minimizing the global number of cut edges (XTRAPULP-VertBalance, XTRAPULP-VertRefine). We defined Imb_v and Imb_e as the target maximum part sizes in terms of vertices and edges, respectively (e.g., $Imb_v = (1 + Rat_v) \times \frac{|V|}{p}$). The third stage balances vertices *and* edges, and minimizes the global edge cut *and* maximal edges cut on any part (XTRAPULP-EdgeBalance, XTRAPULP-EdgeRefine). In prior work [1], [18], we have noted solving the easier vertex balance problem with a low cut before attempting to solve the harder edge balance problem tends to result in a better final solution quality in terms of cut than processing in the other direction. However, first balancing edges and then vertices can result in better final imbalance for large part counts on networks with a skewed degree distribution where the former approach can't effectively re-balance parts. Note that we inherently can not guarantee the imbalance constraints will be achieved with either approach.

For our experiments, we alternate between balance and refinement for 3 outer iterations during each of the latter two stages (I_{out}). The balance algorithms run for 5 iterations (I_{bal} below) and the refinement algorithms run for 10 iterations. These iteration counts were selected empirically to provide a reasonable tradeoff between computation time and partition quality (we give experimental justification in the supplemental information for the paper). For certain applications, these iterations can be modified; we include the ability to do so as parameters within our stand-alone executable as well as library interface. Label propagation is typically run for fixed iteration counts, as we have observed convergence is not quickly guaranteed and convergence doesn't necessarily guarantee a higher solution quality [34]. We will describe the generalization of this algorithm to an arbitrary number of constraints in Section 3.5. For space, time, and work complexity discussion, see the supplementary document.

3.2 XTRAPULP-MM Vertex Balancing Phase

We allow considerable imbalance during our graph growing initialization phase to ensure most vertices are reached by at least one part; vertices that otherwise aren't reached by a part are randomly assigned to one. However, even with large imbalance this subsequent balancing stage can effectively rebalance the initial parts. The vertex balancing stage of XTRAPULP utilizes label propagation with a weighting function W_v to attempt to achieve the balance objective. For each possible part i that a vertex v might move to, the count of each number of neighbors in part i ($counts(i)$) is multiplied by W_v. Vertex v then assigns itself to the part i with the maximum such *gain* of:

$$\forall i \in p : gain(i) = W_v(i) \times counts(i).$$

W_v for part i is roughly proportional to the target part size Imb_v divided by the known current part size $S_v(i)$ plus an estimate to the number of non-local updates to part i:

$$W_v(i) = \max\left(\frac{Imb_v}{S_v(i) + mult \times C_v(i)} - 1, 0\right).$$

As can be inferred, we do not explicitly synchronize the current sizes of each part i ($S_v(i)$) in each iteration of the algorithm. Instead, we calculate the number of vertices gained or lost ($C_v(i)$) in each part i in the current iteration. When updating the weights applied to each rank i ($W_v(i)$), we find an approximate size of each part based on its size at the end of the previous iteration, the number of changes during this current iteration, and a dynamic multiplier $mult$. The approximate size for part i is calculated as:

$$S_{v_{new}}(i) = S_v(i) + mult \times C_v(i).$$

This multiplier allows fine-tuned control of imbalance when running on thousands of processors in distributed-memory, without necessitating a large number of synchronizations. This was not an issue in previous shared-memory work. The multiplier is computed as:

$$mult = nprocs \times \left((X - Y)(\frac{iter_{\text{tot}}}{I_{\text{tot}}}) + Y\right).$$

where $nprocs$ is the number of MPI ranks, $iter_{\text{tot}}$ is a counter of total iterations performed across the outer loops during the balance-refinement stages, I_{tot} is the maximum number of total iterations, and X and Y are parameters.

The basic idea is to use the multiplier to limit how many new vertices a single rank can add to some part i. This prevents all ranks from calculating a high W_v value for a presently underweight part and reassigning a large number of new vertices to that part (as we wish to avoid explicit communication while the assignments are processed). As the iterations progress, we linearly tighten the limit on how many updates a rank can do to each part, until a final iteration, where each rank can provide only up to a share of $\frac{1}{nprocs}(Imb_v - S_v(i))$ additional new vertex assignments to part i. This prevents the imbalance constraint from being violated for any currently balanced part.

When variables $nprocs, X, Y,$ and I_{tot} are held fixed (as they are during actual processing), the function $mult$ is a linear function on the number of iterations with y intercept (iteration 0) of ($nprocs \times Y$) and a final value (iteration I_{tot}) of ($nprocs \times X$). We use values of $Y = 0.25$ and $X = 1.0$, which correspond to each rank being allowed to add up to $4\times$ its "share" of updates to a rank at an initial iteration and just its "share" at the final iteration. As for our determination of the iteration parameters, we also discuss the selection of these X and Y parameters in our supplemental information.

3.3 XTRAPULP-MM Vertex Refinement Phase

This XTRAPULP refinement phase greedily minimizes the global number of cut edges without exceeding the vertex target part sizes Imb_v (if the constraint has been satisfied during the balancing phase) or without increasing the size of any part greater than the current most imbalanced part (Max_v). This refinement algorithm can be considered a variant of FM-refinement [35] or a constrained variant of baseline label propagation. More specifically, it is similar to the balancing algorithm, except that the $counts$ array is not weighted by some $W_v(i)$ for part i. Instead, the new part of vertex v will be the part assigned to most of its neighbors (similar to label propagation), with the restriction that moving v to some part i will not increase the part i's size (or estimated size with the multiplier) to larger than Max_v. We enforce this restriction by checking if:

$$S_v(i) + mult \times C_v(i) + 1 < Max_v.$$

We eliminate part i as a possible choice for vertex movement if the condition doesn't hold; e.g., the imbalance is expected to increase or exceed Imb_v. In effect, we directly move vertices to the part in their neighborhood that will minimize the global cut, without *increasing* the current estimated global imbalance or exceeding our imbalance criteria.

3.4 XTRAPULP-MM Edge Balancing Phase

After the outer number of iterations of the vertex balance-refinement stages (I_{out}), the edge balance-refinement stages begin. We balance by considering both the target number of edges (Imb_e) and vertices (Imb_v) per rank. The goal is to balance the number of edges per rank while not creating vertex imbalance. The vertex weighting terms $W_v(1 \dots p)$ are replaced by edge and cut imbalance weighting terms $W_e(1 \dots p)$ and $W_c(1 \dots p)$ using the current global maximum edge size per part Max_e and maximum cut size per part Max_c. Similar to the vertex weighting term, we again consider W_e for part i and vertex v with degree $d(v)$ as:

$$W_e(i) = \max\left(\frac{Max_e}{S_e(i) + mult \times C_e(i)} - d(v), 0\right).$$

In the above, $S_e(i)$ is the currently known size in terms of edges for part i and $C_e(i)$ is correspondingly the number of local changes. In the above Max_e is replaced with Imb_e once the target imbalance criteria has been achieved. W_c is calculated equivalently with the current known per-part cut and per-part cut changes, except there is no target imbalance, so we only use Max_c. W_e and W_c are then used to weight parts that are currently underweight both in terms of the number of edges and cut edges. We consider the *gain* of moving a vertex to part i with the equation:

$$gain(i) = counts(i) \times \Big(R_e W_e(i) + R_c W_c(i)\Big).$$

R_e and R_c dynamically bias the $gain$ towards balancing either edges or per-part cuts. Specifically, as long as the target edge imbalance has not been achieved on the prior iteration, we increase R_e by

$$R_e = R_e \times \frac{Max_e}{Imb_e},$$

and we set $R_c = 1$. Likewise, once (or *if*) we achieve the target edge imbalance, we increase R_c as

$$R_c = R_c \times Max_c,$$

and we set $R_e = 1$. In effect, this controls the relative influence of $W_e(i)$ and $W_c(i)$ on the calculated $gain(i)$ magnitude, which corresponds to greater movement of vertices to parts either underweight in edge or per-part cut, respectively.

Using our multiplier $mult$ for these distributed-memory updating procedures, we restrict the number of edges and cut edges transferred between any parts per iteration; we use the same X and Y constants as before. Vertices (C_v), edges (C_e) and cut edges (C_c) changed per part are all therefore synchronized among ranks. Correspondingly, part sizes are updated in terms of vertices (S_v), edges (S_e), and cut edges (S_c), and are used to update the W_e and W_c weights (in addition to R_e and R_c) as S_v updated W_v. At the conclusion of I_{bal} edge balancing iterations, a refinement phase similar to the one run during the vertex balancing stage is used. The only change is that we calculate Max_v, Max_e, and Max_c and restrict movement of a vertex to any part that would increase the global maximum imbalance in terms of vertices, edges, or cut size.

3.5 Generalized Multi-weight Partitioning

For some graph analytics as well as many scientific computing applications, it is desired to balance some arbitrary number of vertex weights within each part. Our baseline PULP-MM algorithm can be considered as balancing two weights, a unary weight for each vertex to balance the number of vertices per part and a weight representing the degree of each vertex to balance the number of edges per part. Similarly, in many applications there is some non-unitary weighting to each edge, representing non-uniform communication across these edges. As such, we developed a generalized approach for balancing some arbitrary number of vertex weights while minimizing the weighted global edge cut and maximum per-part cut. The overview of this approach is similar to PULP-MM, and is given in Algorithm 2.

Algorithm 2 XTRAPULP weighted generalization overview: $parts \leftarrow$ XTRAPULP-Weighted($G_g(V_g, E_g, V_{W_g}, E_{W_g}), Imb$)

```
G(V, E, W_V, W_E) ← distributeGraph(G_g(V_g, E_g, V_{W_g}), E_{W_g})   ▷ distribute input graph to MPI ranks
I_tot ← I_out × (I_bal + I_ref)                ▷ total per-phase iteration count
iter_tot ← 0                                   ▷ running per-phase iteration count
parts ← XTRAPULP-Init(...)
for i = 1 ... I_out do
    parts ← XTRAPULP-Balance(...)
    parts ← XTRAPULP-Refine(...)
```

While this approach is similar to PULP-MM, there exist a number of noteworthy changes. Here, we consider the input graph as having some additional set of edge weights E_W and some number of vertex weights V_W. We scale the weights to be within $[0 \dots 1]$. In our current implementation, we assume there is a single weight per edge but an arbitrary number of weights per vertex. Our target part imbalance Imb is now also an array, one index per vertex weight, as determined by from the horizontal multi-weight formulation in Section 2.1 (i.e., $Imb_j = Rat_j \times \frac{|V_{W_j}|}{p}$). We still utilize a balance-refine approach, where we iterate between balancing for weights and refining for cut within the outer loop for some number of iterations, as determined by two times the number of per-vertex weights. We retain the number of balance and refinement iterations as from PULP-MM. We note again that these iteration counts are empirically selected and can be easily adjusted for a given application.

As we will discuss, our generalized algorithm only uses a single balance function and refine function that each optimize for all vertex weights at once. Experimentally, we have observed that separating balance phases only gives

a tangible benefit if you solve the balance problems from "easiest" to "hardest". On unweighted real-world graphs, this is trivially "balance vertices" then "balance edges". For the arbitrary weights run with on actual applications, such an order is often indeterminate, and a separation of balance phases using what is effectively an "arbitrary" order can result in poor results in terms of cut quality.

3.5.1 Generalized Multi-weight Balancing

We note here the primary modifications versus the XTRAPULP-MM Edge Balance approach described earlier. First, the current part weights and bias arrays, W and R, are now specified for each vertex weight j as W_j and R_j; similarly, the Max values are specified for weight j as Max_j. Functionally, they serve the same purpose as within the vertex and edge balancing phases of PULP-MM. As a part become underweight for weight j, its Max_j value increases proportionally to the ratio of under-weightedness versus the per-weight target imbalance Imb_j. The bias terms R_j again are used globally; however, in this context we bias towards the most currently under-balanced weight. For instance, to compute W for weight j of part i, we have:

$$W_j(i) = \frac{Max_j}{S_w(i) + mult \times C_w(i)} - V_{W_j}(v).$$

Note that we do not have a Max() function here, and we allow weights to become negative. This is useful for when we compute the final *gain* for moving a vertex to a part i. E.g., it might slightly hurt our balance in one weight to move a vertex to a certain part, but if it's beneficial enough in other weights, we will still want to allow that move. For our bias terms R, we update all terms equivalently as

$$R_j \leftarrow R_j \times (Max_j/Imb_j),$$

for a given weight j. Once we achieve a target imbalance for some j, we set $R_j = 1$ and hold it there as long as the balance target is retained. To compute the $counts$ array, we now additionally consider the sum of edge weights from v to part i as

$$counts(i) = \sum E_{W_j}(e), \forall (e = (v,u) \in E(v) : parts(u) = i),$$

which is the weighted sum of all edges from v to the vertices u in part i in the neighborhood of v.

The second major modification is how we calculate the final *gain* function to determine vertex v's new part assignment. We consider the $gain(i)$ for moving vertex v to part i with current part assignment x as:

$$gain(i) = counts(i) \times \sum_{j \in W_V} (W_j(i) - W_j(x)) \times (V_{W_j} R_j).$$

The $gain$ for moving v from part x to part i is the sum over all vertex weights of the difference in current weight imbalances between the parts, times the vertex weight of v, times the current bias, multiplied by the edge-weighted connection v has to part i. We select the maximum $gain$ over all parts for v's new (or unchanged) assignment, with ties broken randomly. Our communication and synchronization routines for this generalized method is unchanged relative to PULP-MM, except for that we now track changes and sizes for each vertex weight j as C_j and S_j, and we correspondingly exchange these updates on each iteration.

3.5.2 Generalized Multi-weight Refinement

The refinement approach for our generalized method is again similar to our methods within PULP-MM for both vertex and edge refinement. We explicitly track the maximum imbalance for each vertex weight and refine the minimize edge cut such that we do not exceed this maximum imbalance over all parts.

3.6 Graph Representation and Parallelism

We base our implementation off of our HPCGRAPH Distributed Graph Analysis Framework [30] (`v0.1`). We use MPI+OpenMP for parallelization. Every subroutine and method is fully parallelized, except for updates to the weighting and bias variables, which is the primary enabler of our scalability. For the graph representation, we use a distributed one-dimensional compressed sparse row-like structure, where each rank owns a subset of vertices and their incident edges. When distributing the graph for the partitioner, we utilize either random or block distributions of the vertices. We observe random distributions are more scalable in practice for irregular networks, but block distributions might benefit partition quality on well-ordered inputs; in general and in our results, we opt for random distributions for the sake of consistency and scalability.

For memory and computation efficiency, we relabel global vertex identifiers to local per-rank identifiers. Each vertex's global identifier is mapped to their rank-local identifier using a hash map; we implement our own efficient hash map for this purpose. Local to global translation uses values stored in a flat array. Each rank stores part labels for both its owned vertices as well as its ghost vertices (vertices in its one hop neighborhood that are owned by another rank). When computing a partition, a rank will calculate updates only for its owned vertices and communicate the updates so the rank's neighbors update assignments for their ghosts. See the supplementary document for a more detailed discussion into the space complexity of our approach.

4 RESULTS

4.1 Experimental Setup

We evaluate XTRAPULP performance at modest scale on several small-world publicly available graphs, representing social networks, information networks, and web crawls. While XTRAPULP is not designed for regular high-diameter graphs, we do evaluate performance on several mesh and mesh-like graphs, such as scientific computing meshes used internally at Sandia National Labs (*InternalMesh*{*1,2,3,4*}). We give the test instances in Table 1. These graphs vary in scale from 69 million edges (*LiveJournal*) to 4.3 billion edges (*RMAT-28*).

We perform large-scale evaluations on the 2012 Web Data Commons hyperlink graph[3], which is created from

3. http://webdatacommons.org/hyperlinkgraph/

TABLE 1
Graph properties in terms of number of vertices (n), number of edges (m), average degree (d_{avg}), maximum degree (d_{max}), and in-memory size in gigabytes (GB)

Graph	n	m	d_{avg}	d_{max}	GB	Source
LiveJournal	5.4 M	69 M	26	20 K	0.60	[36]
Orkut	3.1 M	117 M	76	33 K	0.96	[37]
DBPedia	67 M	258 M	8	7.2 M	2.5	[38]
WikiLinks	26 M	601 M	46	39 K	5.0	[38]
Friendster	66 M	1.8 B	54	5.2 K	15	[37]
Twitter	53 M	1.9 B	72	3.7 M	16	[39]
indochina	7.3 M	149 M	41	256 K	1.3	[40], [41], [42]
arabic	23 M	552 M	49	576 K	4.6	[40], [41], [42]
it-2004	41 M	1.2 B	50	1.3 M	9.9	[40], [41], [42]
sk-2005	51 M	1.9 B	72	8.6 M	16	[40], [41], [42]
uk-2002	1.8 M	298 M	28	195 K	2.4	[40], [41], [42]
uk-2005	39 M	781 M	40	1.8 M	6.6	[40], [41], [42]
uk-2007	106 M	3.3 B	31	975 K	27	[41], [42], [43]
WDC12-pay	39 M	623 M	16	3.8 M	5.3	[3], [4]
WDC12-host	89 M	2.0 B	22	3.4 M	17	[3], [4]
WDC12	3.6 B	129 B	36	95 M	1100	[3], [4]
RMAT-22	2.1 M	68 M	32	1.7 K	0.56	[44], [45]
RMAT-24	17 M	268 M	32	3.3 K	2.3	[44], [45]
RMAT-26	67 M	1.1 B	32	6.7 K	9.3	[44], [45]
RMAT-28	268 M	4.3 B	32	13 K	37	[44], [45]
InternalMesh1	272 K	3.5 M	26	26	0.03	
InternalMesh2	2.2 M	28 M	26	26	0.24	
InternalMesh3	18 M	220 M	26	26	1.9	
InternalMesh4	140 M	1.8 B	26	26	16	
nlpkkt160	8.3 M	111 M	27	27	0.95	[46]
nlpkkt200	16 M	220 M	27	27	1.9	[46]
nlpkkt240	28 M	373 M	27	27	3.2	[46]

the Common Crawl web corpus[4]. This graph contains 3.56 billion vertices and 128 billion edges, and is the largest publicly available real-world graph to date. We use the pay and host level domain graphs from this crawl as well (*WDC12-pay* and *WDC12-host*). For performance and scaling comparisons, we also use *R-MAT* (labeled *RMAT-X*) and Erdös-Rényi (labeled *RandER*) random graphs. Additionally, we generate random graphs with a high diameter (labeled *RandHD*) by adding edges using the following procedure: for a vertex with identifier k, $0 \leq k < n$, we add d_{avg} edges connecting it to vertices chosen uniform randomly from the interval $(k - d_{\text{avg}}, k + d_{\text{avg}})$.

We use three compute platforms for evaluations. *Compton* is a 16 node cluster; each node has two eight-core 2.6 GHz Intel Xeon E5-2670 (Sandy Bridge) CPUs and 64 GB main memory. *Blake* is a 40 node cluster; each node has two 24-core Intel Xeon Platinum 8160 (Skylake) CPUs and 192 GB main memory. We also used the NCSA *Blue Waters* supercomputer for large-scale runs. *Blue Waters* is a Cray XE6/XK7 system with 22 640 XE6 compute nodes and 4228 XK7 compute nodes. We used only the XE6 nodes. Each node has two eight-core 2.45 GHz AMD Opteron 6276 (Interlagos) CPUs and 64 GB memory. Our experiments used up to 8192 nodes of *Blue Waters*, which is about 36% of the XE6 total capacity.

We compare XTRAPULP (version 0.3) against ParMETIS version 4.0.3 [13], PULP version 0.2 [1]. We used the default settings of ParMETIS and PULP for all experiments. The build settings (C++ compiler, optimization flags, MPI library) for all the codes were similar on *Blue Waters*, *Compton*, and *Blake*. Unless otherwise specified, we use one MPI rank per compute node for multi-node parallel runs of XTRAPULP, and set the number of OpenMP threads to the number of shared-memory threads.

We perform several scaling tests of the multi-constraint multi-objective XTRAPULP PULP-MM variant on the *Blue Waters* supercomputer and *Blake* testbed cluster. For these and subsequent tests, we optimize for edge cut and max per-part cut and place a 10% imbalance constraint on both vertices and edges, unless otherwise specified. We also test the generalized multi-weight variant on *Blake*, with a fixed 10% imbalance for all weights. We note the selected imbalance constraint is relatively higher than the commonly used 3-5% constraints in many similar research papers; however, the skewed graphs and multi-constraint problems we consider here pose a significantly greater challenge to balance than the meshes or more structured networks that many partitioners are optimized for. As we'll show, even with the loosened constraints, achieving the balance tolerances exactly is still quite difficult in the distributed setting for highly irregular networks.

4.2 Large-Scale Performance

4.2.1 *Partitioning of the* WDC12 *Graph*

We first analyze XTRAPULP performance in its intended use-case: running in a massively parallel setting on multi-billion vertex networks. To perform these tests, we use the *Blue Waters* supercomputer to partition the real-world Web Data Commons hyperlink graph (*WDC12*). Figure 1 gives the execution time for partitioning the *WDC12* graph and three generated graphs (*RMAT*, *RandER*, *RandHD*) of the same scale (3.56 billion vertices and 128 billion edges); for RMAT, we select *RMAT* parameters to approximately match the degree distribution (specifically, the maximum degree and number of low-degree vertices). We run on 256-4096 nodes of *Blue Waters* (4096-65536 cores) and compute 256 parts.

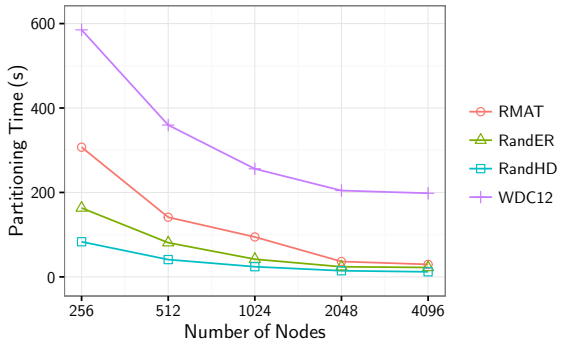


Fig. 1. XTRAPULP parallel performance (*strong scaling*) results on *Blue Waters* for computing 256 parts of various test graphs. We show absolute execution times.

As shown in Figure 1, XTRAPULP exhibits *strong scaling* up to 4096 nodes on all tested graphs. The speedups achieved are 3.0×, 10.4×, 7.3×, and 7.0× for *WDC12*, RMAT, *RandER*, and *RandHD* graphs, respectively, when going from 256 to 4096 nodes (16× increase in parallelism). As expected, we see better speedups for the synthetic graphs due to

4. http://commoncrawl.org

better computational and communication load balance. The partitioning time for the *RandHD* network on 256 nodes is nearly $\frac{1}{8}$ the partitioning time for *WDC12*, even though the graphs are the same size. This is due to significantly different degree distributions, which affect the efficiency of intra-node parallelism and inter-node communication.

We compare quality of XTRAPULP to the practical competing methods of Vertex Block, Edge Block, and Random partitioning in Table 2. We know of no other openly-available parallel partitioners that run at this scale. Edge Block partitioning stores a contiguous set of vertices and all their adjacencies on each rank such that each rank has approximately the same number of edges. Vertex Block partitioning stores roughly the same number of vertices and all their adjacencies on each rank. Random partitioning assigns vertices to ranks randomly. We again consider 256 parts, and we report the quality results from the XTRAPULP runs on 256 nodes of *Blue Waters*. We compare edge cut as percentage of edges cut, vertex and edge imbalance, and maximum per-part cut as "Cut Imbalance" (maximum per-part cut over total edges cut). The times to compute the block and random partitions are insignificant and therefore not listed.

TABLE 2
Partitioning quality comparison on the *WDC12* graph when comparing XTRAPULP to the competing practical method of vertex block, edge block, and random partitioning.

Method	Edge Cut	Vert. Imb.	Edge Imb.	Cut Imb.
XTRAPULP	5%	10%	15%	37%
Vertex Block	16%	0%	85%	39%
Edge Block	18%	105%	1%	42%
Random	99%	1%	2%	1%

We observe that XTRAPULP improves upon the next-best edge cut by a factor of over 3×, while only exceeding the edge balance constraint by 5%. Random partitioning is notably the most balanced of all of the methods, but this comes at a massive cost in cut quality. As we will later show in Section 4.5, the improvements in cut quality offered by XTRAPULP can have a noted practical benefit, even when considering the additional time needed to compute the partitions.

4.2.2 Weak Scaling

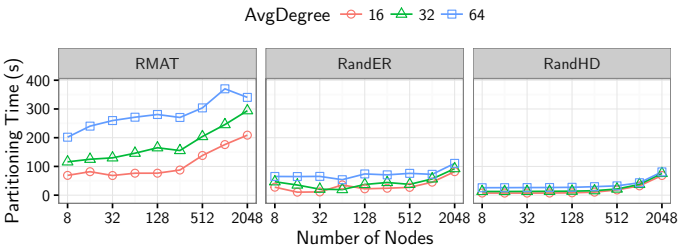


Fig. 2. XTRAPULP (*weak scaling*) results on *Blue Waters* for various *RMAT*, *RandER*, and *RandHD* graphs. The number of graph vertices per node is $\approx 2^{22}$. The number of parts computed is set to number of nodes.

Next, we perform *weak scaling* experiments on *Blue Waters*, using 8 to 2048 compute nodes. We generate *RMAT*, *RandER*, and *RandHD* graphs of different sizes, and double the number of vertices as the node count doubles. The 8-node runs use graphs with 2^{25} vertices, whereas the 2048-node runs are for graphs with 2^{33} vertices. We also vary the average vertex degree, using $d_{avg} = 16$, 32, and 64. The number of parts computed is set to the number of nodes being used for the run; thus, the computational cost changes as more parts are computed when the number of nodes increases. Figure 2 shows these results. We see that partitioning time is lowest for *RandHD* and highest for *RMAT*, similar to the strong scaling results. *RMAT* graphs appear to be the most sensitive to average degree (or edge count) variation. For 2048-node runs, when increasing the average degree (and thereby, the number of edges) by 4× (16 to 64), the running times of *RMAT*, *RandER*, and *RandHD* graphs increase by 1.63×, 1.35×, and 1.18×, respectively. Finally, we note that overall weak scaling performance is dependent on the graph structure. For the regular *RandHD* graphs, we see almost flat running times up to 1024 nodes, but for *RMAT* graphs, we observe a noted rise in times beyond 256 nodes. As graph size increases in *RMAT* graphs, so does the maximum degree, and as observed before, this might lead to computation imbalance with the one-dimensional graph distribution used in XTRAPULP.

4.2.3 Trillion Edge Runs

We ran an additional set of experiments on 8192 nodes, or 131,072 cores of *Blue Waters* with synthetically generated graphs with up to 17 billion vertices and 1.1 trillion edges. These tests use over a third of the available compute nodes on *Blue Waters*. At this scale, communication time tends to dominate the overall running time, and network traffic can have a considerable impact on total execution time. We were able to partition 17 billion (2^{34}) vertex, 1.1 trillion (2^{40}) edge *RandER* and *RandHD* graphs in 380 seconds and 357 seconds, respectively, on 8192 nodes. We successfully partitioned a 17 billion (2^{34}) vertex, 550 billion (2^{39}) edge *RMAT* graph in 608 seconds. Load imbalance and resulting out-of-memory issues prevented us from scaling *RMAT* to the full trillion edges, and addressing these problems is a good avenue for future work. Otherwise, in regards to performance, we state that the XTRAPULP strong and weak scaling results on *Blue Waters* demonstrate that there are no performance-crippling bottlenecks at scale in our implementation.

4.3 Small-scale Performance

We extensively test XTRAPULP at a smaller scale (16 nodes of *Blake*), for direct performance comparisons to ParMETIS and PULP. For MPI-only ParMETIS, we run 96, 48, 12, and 1 ranks per node and report the best time and quality in order to provide an extremely conservative comparison. OpenMP-only PULP results are with full 96-way threading on a single node. Note that XTRAPULP is explicitly designed for much larger-scale processing, so we perform this small-scale analysis here only to give relative performance comparisons to the current state-of-the-art.

4.3.1 Performance Comparison

We present 16-node performance via several performance plots in Figure 3. We partition all graphs from the first para-

graph of Section 4.1 into all powers-of-2 parts between 2 and 512, while constraining 10% balance for vertices and edges. ParMETIS was unable to complete for all graphs larger than *arabic*, as well as the smaller but more skewed graphs of *DBPedia* and *WikiLinks*. The error messages indicated failure of a halo exchange communication subroutine. PuLP completed on all tests except for 256 and 512 parts on *RMAT-28*, which failed due to out-of-memory errors. The performance plots for edge cut and execution time were created by taking the cut/time of a partitioner on a given test (graph-#parts) and dividing it by the best cut/time among all three partitioners. For balance, the excess imbalance is reported as the sum of vertex imbalance and edge imbalance over the 10% tolerance. We then sort these performance ratios in increasing order; the lower the points are on the plots indicate better relative performance. We also separate each plot into the tests in which ParMETIS completed and those that ParMETIS failed on. In effect, the smaller-scale results are plotted on the left and the larger-scale results are plotted on the right.

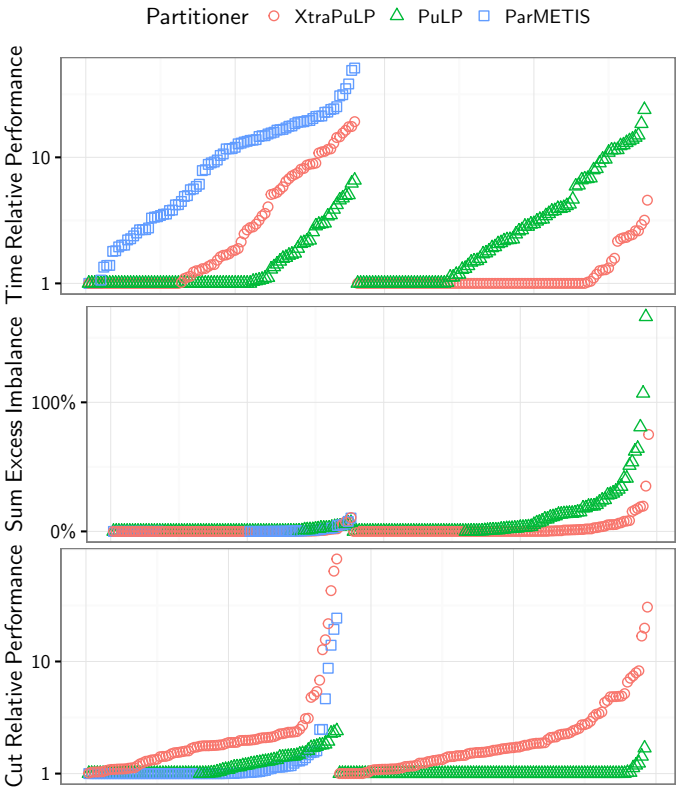


Fig. 3. Partitioning comparison between XtraPuLP, PuLP, and ParMETIS in terms of time (top), part imbalance (middle), edgecut (bottom) as performance plots. Lower points indicate better relative performance.

We make several observations about the results displayed in Figure 3. We note that XtraPuLP and PuLP demonstrate considerably faster execution time than ParMETIS on almost all tests. PuLP is generally faster than XtraPuLP at the smaller scale despite similar algorithms; this is due to the parallel overheads introduced by the distributed processing framework in XtraPuLP designed for tera-scale graphs. For larger graph scales and higher part counts, XtraPuLP's performance dominates. We make note of the fact that the per-node performance difference in the most recent `graph500.org` results between the fastest single node and fastest multi-node entries is several orders-of-magnitude. Relative to this metric, XtraPuLP's distributed framework is rather efficient.

Overall, XtraPuLP produces the most balanced partitions, though at the smaller scale this difference can be considered negligible. At the larger scale, XtraPuLP's partitions were considerably less imbalanced than PuLP's. We credit this performance to our per-iteration update and communication strategy detailed in our methods. Also at the larger scale, the graphs become more difficult to compute balanced multi-constraint partitions on, and the results here demonstrate as much. Generally, we observe an inverse relationship between performance on the balance metric and edge cut. At the smaller scale, most partitioners produce balanced parts and ParMETIS and PuLP are most competitive. As the scale increases, we can observe that this inverse relationship becomes apparent between XtraPuLP and PuLP. We explicitly note that on the tests in which both XtraPuLP and PuLP achieve the balance constraint, the difference in edge cut is considerably less than for tests in which PuLP greatly exceeds the imbalance tolerance.

4.3.2 *Strong Scaling*

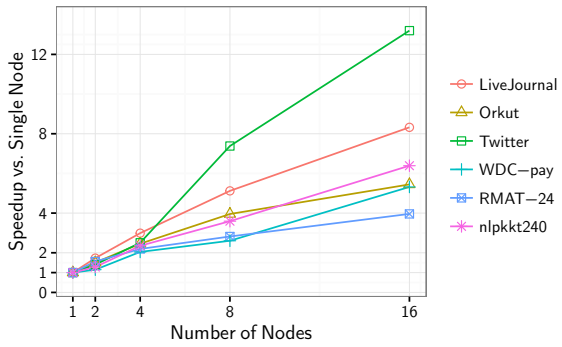


Fig. 4. XtraPuLP parallel performance (*strong scaling*) results on *Blake* for computing 16 parts of various test graphs. We plot speedups versus the single node time.

For these smaller-scale tests, we also include strong scaling results for XtraPuLP in Figure 4. We give strong scaling for six representative graphs from 1 to 16 nodes of *Blake*. Note that graph sizes vary significantly, ranging from the 69 million-edge *LiveJournal* graph to the 2 billion-edge *Twitter* graph. We observe a range of relative speedups, attributable to the graph structure. The geometric mean speedup across all graphs is 5.4×. While this is lower than ideal strong scaling, we again note that the inherent irregularity of complex networks makes distributed processing difficult in general to scale. Our strong scaling performance is in line with related work [23].

4.4 Generalized Multi-constraint Scientific Computing Comparisons

In Figure 5, we observe that our generalization of XtraPuLP's balance routine performs relatively comparable to ParMETIS in a k-way multi-constraint setting. As many scientific computations rely on multi-weight partitioning

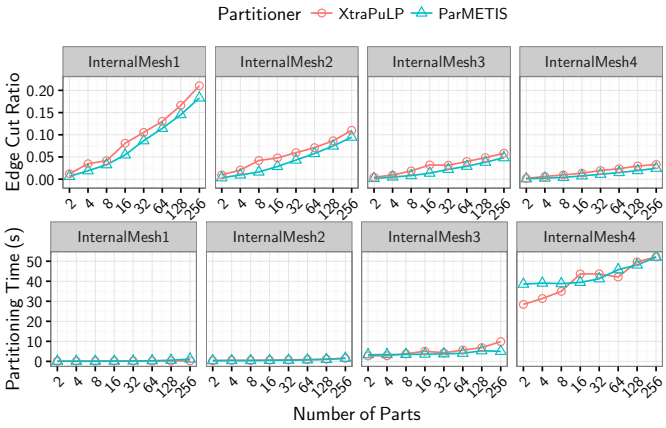


Fig. 5. Partitioning quality in terms of cut (top) and summed maximum imbalance (middle) along with execution time (bottom) for XTRAPULP and ParMETIS solving the single objective multi-constraint partitioning problem.

to achieve load balance, we consider our internal application meshes (*InternalMesh1-4*) in these experiments. We consider three vertex weights as representative of compute load for many scientific and graph analytic computations. These weights consist of a single unary weight along with weights representing the 1-hop neighborhood and 2-hop neighborhood sizes for each vertex. We set the load imbalance constraint for each weight to 5% for these experiments. We compute 2-256 parts while running on *Blake*. Again to provide a generous comparison to ParMETIS, we run with 8 MPI ranks for XTRAPULP and again consider the best time for ParMETIS over some n ranks running on 8 nodes.

We observe similar scaling behavior between the two partitioners in terms of the quality and partitioning time versus the number of parts being computed. In general in these tests, ParMETIS produces a better cut on average. Both partitioners are about equal in execution time, and both partitioners achieve the imbalance constraints for each individual weight (we do not plot these results as the lines are mostly flat). Overall, these results indicate that our generalization of XTRAPULP to horizontal multi-weight partitioning produces cuts comparable to ParMETIS on our internal application meshes. Using our generalized approach for multi-weight partitioning of irregular networks shows similar relative performance differences, although we do not show full results for brevity.

4.5 Large-scale Graph Analytics

We next demonstrate that XTRAPULP can significantly improve performance of real-world analytics. We consider analytics scalable enough to run on the 128 billion edge *WDC12* from the `HPCGraph` framework [47]. Without a partitioner that can process graphs of this scale, the common approaches to running analytics are to use simple balanced vertex and edge assignment strategies that do not optimize for edge cut. In Figure 6, we give the execution times of six analytics on *WDC12* with the four partitioning strategies previously discussed (XTRAPULP, Vertex Block, Edge Block, and Random).

The six analytics considered are a Harmonic Centrality (HC) computation of the 100 max-degree vertices, an approximate K-core decomposition (KC), 10 iterations of the Label Propagation-based community detection algorithm (LP), 10 iterations of PageRank (PR), a strongly connected components decomposition (SCC), a weakly connected components decomposition (WCC), and the partitioning time (Part); only XTRAPULP has a non-zero time for "Part". We also include the total time (Total) to run all analytics plus partitioning for all methods.

Using balanced XTRAPULP partitions reduces end-to-end execution time (including partitioning) by 30%, from 1229 seconds with an edge block partitioning to 867 seconds with XTRAPULP. We see a substantial reduction in analytics where inter-node communication time is directly proportional to total edge cut (e.g PR, LP, and WCC) even when including the XTRAPULP partitioning time. Not all analytic execution times appear to improve with XTRAPULP (e.g. K-core, SCC). We suspect that this can be explained by the way vertex block and edge block retain the original vertex ordering of the *WDC12* graph. The native ordering is highly clustered as a result of the crawling methodology, and it has been demonstrated in the literature that a clustered order has a beneficial impact on the number of iterations and total time required to complete graph analytics [24], [48].

4.6 Sparse Matrix-Vector Multiplication

In Table 3, we present results that show partitioning impact on sparse matrix vector multiplication (SpMV). We use the Epetra package of the Trilinos scientific computing library [49] to perform 100 SpMV operations, using matrices constructed from the six select test graphs previously shown. Note that we replace *Twitter* with *Friendster*, as *Twitter* failed to successfully complete in these tests due to out-of-memory errors; Friendster also failed on the 16-rank runs. Parallel SpMV is a key computation in eigensolvers and iterative methods for liner systems. We use several partitioning strategies, including one dimensional vertex block (1D-Block), random (1D-Rand), ParMETIS (1D-PM), and XTRAPULP (1D-XTRAPULP). For ParMETIS and XTRAPULP, we compute the multi-constraint problem as in Section 2.1 (with 10% imbalance for rows and nonzeros per rank), where with ParMETIS we consider minimizing edge cut and with XTRAPULP we minimize edge cut and maximum per part cut. We consider unit weights for edges, as edges are unweighted in the original datasets.

We also utilize 2D distributions with vertex block (2D-Block) and random partitions (2D-Rand). Additionally, using a strategy for mapping 1D partitions into 2D distributions [15], we run with 2D distributions produced from our 1D ParMETIS (2D-PM) and XTRAPULP (2D-XTRAPULP) partitions. We run these tests on 1, 8 and 16 nodes of *Compton* with 16, 128, and 256 MPI ranks, respectively. We observe that using XTRAPULP partitioning can often accelerate the SpMV operation time in these tests. We observe a 2.77× (geometric mean) reduction in execution time when using 2D XTRAPULP-based distributions instead of 1D-Rand for 256-way parallel code on the five irregular graphs, and 2D-XTRAPULP results in the best performance in **60**% of the total tests. 2D-XTRAPULP also demonstrates speedup versus 2D ParMETIS across all datasets except for the *nlpkkt240* mesh, which is the class of networks that

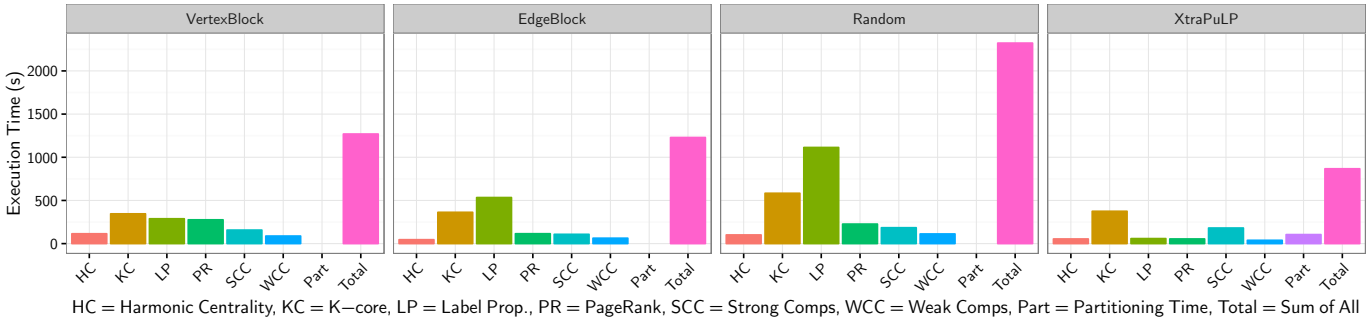


Fig. 6. The parallel performance results of various parallel graph analytics (HC, KC, LP, PR, SCC, WCC) on 256 nodes of *Blue Waters*, executed on the *WDC12* graph with different graph partitioning strategies.

TABLE 3
The performance results of parallel sparse matrix vector multiplication (SpMV) on 1 node (16 MPI ranks) to 16 nodes (256 MPI ranks) of *Compton*, with different graph partitioning strategies. PM: ParMETIS. We report time for 100 SpMVs. The fastest times per row (graph-#ranks) are highlighted.

Graph	MPI ranks	Execution time for 100 SpMVs (s)								Speedup	
		1D partitioning				2D partitioning				2D XtraPuLP	
		Block	Rand	PM	XtraPuLP	Block	Rand	PM	XtraPuLP	over 1D Rand	over 2D PM
LiveJournal	16	19.87	12.24	10.90	9.81	15.74	10.09	9.26	8.32	1.47×	1.11×
	128	6.31	4.29	3.41	3.05	4.04	1.92	2.07	1.85	2.32×	1.12×
	256	4.73	2.96	2.65	2.41	2.32	1.26	1.28	1.12	2.64×	1.14×
Orkut	16	28.80	26.62	23.31	23.82	19.88	17.29	17.86	16.51	1.61×	1.08×
	128	10.05	10.43	7.77	8.35	4.20	3.07	3.56	3.42	3.05×	1.04×
	256	6.55	6.27	5.63	5.25	2.25	1.94	1.82	1.91	3.28×	0.95×
Friendster	128	240.07	137.21		141.72	161.45	81.11		83.33	1.65×	
	256	154.43	79.67		89.98	79.56	46.85		46.69	1.71×	
WDC12-pay	16	614.82	175.25		155.79	395.94	85.79		76.99	2.28×	
	128	224.84	40.91		41.35	96.94	25.79		24.27	1.69×	
	256	153.51	43.14		27.61	59.18	14.65		14.51	2.97×	
RMAT-24	16	138.76	144.51	107.19	99.87	107.28	108.22	87.83	83.07	1.74×	1.06×
	128	44.20	50.49	27.69	27.28	25.57	25.31	15.93	15.52	3.25×	1.03×
	256	32.22	37.37	19.63	19.76	14.67	15.93	9.97	10.02	3.73×	1.00×
nlpkkt240	16	30.96	26.67	19.86	17.55	16.08	40.96	16.96	18.03	1.48×	0.94×
	128	2.59	26.73	2.14	2.57	2.71	14.55	2.24	2.56	10.44×	0.88×
	256	1.62	22.61	1.04	1.37	1.81	10.97	1.12	1.56	14.50×	0.72×

ParMETIS is optimized for. In addition, regular meshes such as *nlpkkt240* often do not directly benefit from a 2D distribution, as they have good edge separators and sparse subdomain connectivity.

4.7 Other Comparisons

The only other partitioner we know of that runs at the trillion-edge scale is *Distributed Neighbor Expansion* (DNE) by Hanai et al. [23]. This work solves the edge-based partitioning problem, where edges are balanced per-rank and vertices are cut. This vertex cut, also called the *replication factor*, is minimized. That work performed a comparison against XtraPuLP in their experimentation. To summarize their findings, Distributed Neighbor Expansion used less memory than XtraPuLP, quality was comparable on some networks (uk-2007) but better on others (*Twitter*, *Friendster*), while XtraPuLP was on average as fast or slightly faster. We note that XtraPuLP is not tuned to solve the edge partitioning problem, and its outputs had to be converted to an appropriate format by randomly assigning cut edges to one neighboring part. We duplicate their experiments but convert vertex cuts from DNE to edge cuts by assigning a vertex solely to one part on which it has a replica. We note on a selection of benchmark graphs and computing 2-256 parts that XtraPuLP is on average 33% faster and gives cuts of 14% the size of DNE. We only assign a single rank per part being computed for XtraPuLP to enable a one-to-one comparison on performance, as DNE requires the number of ranks and parts computed to be equal. In practical applications, we'd expect XtraPuLP to be able to further strong scale. We were unable to run DNE on the WDC12 graph, due to limitations of its I/O and pre-processing routines. Regardless, these results aren't directly comparable, due to the large differences in the vertex and edge cut optimization problems. Please see the supplemental document for more information related to this experiment. In addition, we also perform single constraint (balance vertices) single objective (minimize edge cut) comparisons against ParMETIS and ParHIP. We note that XtraPuLP generally runs considerably faster but at a cost of cut quality in these experiments.

We also consider comparisons to other recent work that processes at a comparably large scale [10], [21], [22]. Ching et al. [10] reports 2 iterations of label propagation on a 48 billion edge graph running on 200 nodes to take 19 minutes. A full run of XtraPuLP requires 90 iterations of label propagation, and we can partition the ~2.5× larger WDC12

graph on 256 nodes in less than 10 minutes. Checconi et al. [21] report about 4 trillion edges traversed per second (TEPS) for breadth-first search of an *R-MAT* graph on 8K nodes. Considering our largest instances on 8K nodes, we report approximately 0.5 trillion edges processed per second (PEPS) for the *RandHD* and *RandER* and 0.34 trillion edges processed per second for the *R-MAT* graph; the algorithmic complexity differences between graph partitioning and simple traversal explain the observed 8-13× performance difference. We make one final comparison to Lin et al. [22]. They report for a PageRank computation of the *WDC12* graph running on 256 nodes of the *TaihuLight* supercomputer a performance of 17 billion PEPS. We report a performance of 39 billion PEPS for partitioning the *WDC12* on 256 nodes of *Blue Waters*.

5 Discussion

As graph instances grow in scale and distributed graph applications become more widely utilized, partitioning is an invaluable tool to enable efficient processing of these applications. As our experiments show, XtraPuLP is scalable to multi-billion vertex graph instances and is able to produce cuts of comparable quality to the current state-of-the-art. The capability of XtraPuLP to balance cuts via minimization of the max per-part cut makes it especially useful to a number of graph applications which utilize full heavyweight all-to-all exchanges during communication.

The current version of XtraPuLP is available on *GitHub* at https://github.com/HPCGraphAnalysis/PuLP/. This version contains a stand-alone executable as well as a library interface. In addition, we currently have an additional interface within the *Zoltan2* package of the *Trilinos* scientific computing library at https://github.com/trilinos/Trilinos. The focus of future development of this software is to better improve cut quality, possibly via a memory-efficient multilevel scheme. As mentioned, traditional multilevel partitioning algorithms tend to be too heavyweight to process tera-scale graph instances. Through a single coarsening phases or some similar reduced approach, sufficient scalability might still be feasible. In addition, we are currently working on other complex constraints and objectives targeting specific internal scientific computing applications. One specific partitioning formulation being considered is computing fully connected parts while retaining the balance and optimization goals of the baseline partitioning problem. One final thrust of ongoing research is to adapt XtraPuLP to forthcoming exascale systems, which are likely to utilize GPUs for the bulk of their computational workload.

Acknowledgments

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under Contract No. DE-AC02-05CH11231 at Rensselaer Polytechnic Institute and Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This work was also supported by NSF grants ACI-1253881 and CCF-1439057.

References

[1] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *Proc. IEEE Int'l. Conf. on Big Data (BigData)*, 2014.

[2] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering: Selected Results and Surveys*, L. Kliemann and P. Sanders, Eds. Springer, 2016, pp. 117–158.

[3] A. Pothen, "Graph partitioning algorithms with applications to scientific computing," Old Dominion University, Tech. Rep., 1997.

[4] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[5] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proc. Symp. on Theory of Computing (STOC)*, 2000.

[6] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, pp. 251–262, 1999.

[7] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[8] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 527–543.

[9] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 393–404.

[10] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: graph processing at Facebook-scale," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[11] G. M. Slota and K. Madduri, "Parallel color-coding," *Parallel Computing, Systems & Applications*, vol. 47, pp. 51–69, August 2015.

[12] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 788–799, 2012.

[13] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 71–95, 1998.

[14] C. Chevalier and F. Pellegrini, "PT-Scotch: A tool for efficient parallel graph ordering," *Parallel computing*, vol. 34, no. 6, pp. 318–331, 2008.

[15] 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. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[16] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *Proc. Int'l. Conf. on Data Engineering (ICDE)*, 2014.

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

[18] G. M. Slota, K. Madduri, and S. Rajamanickam, "Complex network partitioning using label propagation," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S620–S645, 2016.

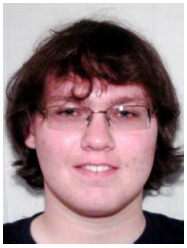
[19] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel Graph Partitioning for Complex Networks," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 9, pp. 2625–2638, 2017.

[20] M. Onizuka, T. Fujimori, and H. Shiokawa, "Graph partitioning for distributed graph processing," *Data Science and Engineering*, vol. 2, no. 1, pp. 94–105, 2017.

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

[22] H. Lin, X. Zhu, B. Yu, X. Tang, W. Xue, W. Chen, L. Zhang, T. Hoefler, X. Ma, X. Liu *et al.*, "Shentu: processing multi-trillion edge graphs on millions of cores in seconds," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 56.
[23] M. Hanai, T. Suzumura, W. J. Tan, E. Liu, G. Theodoropoulos, and W. Cai, "Distributed edge partitioning for trillion-edge graphs," *Proceedings of the VLDB Endowment*, vol. 12, no. 13, pp. 2379–2392, 2019.
[24] G. M. Slota, S. Rajamanickam, , and K. Madduri, "Order or shuffle: Empirically evaluating vertex order impact on parallel graph computations," in *Graph Algorithms Building Blocks Workshop (GABB)*, 2017.
[25] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Proc. ACM/IEEE Conference on Supercomputing (SC)*, 1998.
[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] 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.
[28] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "Adaptive partitioning for large-scale dynamic graphs," in *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 144–153.
[29] J. Buurlage, "Self-improving sparse matrix partitioning and bulk-synchronous pseudo-streaming," Master's thesis, Utrecht University, 2016.
[30] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, 2016.
[31] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
[32] A. George and J. W. Liu, *Computer solution of large sparse positive definite systems*. Prentice Hall, 1981.
[33] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *International Parallel & Distributed Processing Symposium (IPDPS)*, 2017.
[34] G. M. Slota and S. Rajamanickam, "Experimental design of work chunking for graph algorithms on high bandwidth memory architectures," in *International Parallel & Distributed Processing Symposium (IPDPS)*, 2018.
[35] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. Design Automation Conf. (DAC)*, 1982.
[36] 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.
[37] 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.
[38] J. Kunegis, "KONECT - the Koblenz network collection," konect.uni-koblenz.de, last accessed Feb 2017.
[39] 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.
[40] 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.
[41] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. Int'l. World Wide Web Conf. (WWW)*, 2004.
[42] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proc. Int'l. World Wide Web Conf. (WWW)*, 2011.
[43] P. Boldi, M. Santini, and S. Vigna, "A large time-aware graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.
[44] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int'l. Conf. on Data Mining (SDM)*, 2004.
[45] D. A. Bader and K. Madduri, "GTgraph: A synthetic graph generator suite," Georgia Tech, Tech. Rep., 2006.
[46] 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.
[47] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2016.
[48] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 22–31.
[49] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps *et al.*, "An overview of the Trilinos project," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 397–423, 2005.

George M. Slota is an Assistant Professor in Computer Science at Rensselaer Polytechnic Institute in Troy, NY. He received his B.S. in Computer Engineering and his Ph.D. in Computer Science and Engineering from The Pennsylvania State University. His research interests are in graph algorithms, data analytics, and their relation to parallel, scientific, and high performance computing.



Cameron Root was a graduate student at Rensselaer Polytechnic Institute in Troy, NY. He received his Master's degree in Computer Science from Rensselaer in 2019 after receiving his Bachelors in 2018.

Karen Devine is a Distinguished Member of Technical Staff in the Scalable Algorithms Department at Sandia National Laboratories in Albuquerque, New Mexico. She received her PhD in Computer Science from Rensselaer Polytechnic Institute and her undergraduate degree from Wilkes College. Her research interests include high performance computing (HPC), parallel partitioning and load balancing, task mapping, scientific computing, and other parallel algorithms.

Kamesh Madduri is an Associate Professor in the Computer Science and Engineering department at The Pennsylvania State University in University Park, PA. He received his PhD from the College of Computing at Georgia Institute of Technology and his undergraduate degree from the Indian Institute of Technology Madras. His research interests include high-performance scientific computing, graph computations, and scientific data analysis.

Sivasankaran Rajamanickam is a Principal Member of Technical Staff in the Scalable Algorithms Department at Sandia National Laboratories in Albuquerque, New Mexico. He earned his B.E. from Madurai Kamaraj University, India, and his Ph.D. in Computer Engineering from University of Florida. His research interests are in sparse factorizations and preconditioners for linear solvers, combinatorial algorithms, and high performance computing.