Partitioning Trillion-edge Graphs in Minutes

George M. Slota
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY
slotag@rpi.edu

Sivasankaran Rajamanickam & Karen Devine
Scalable Algorithms Department
Sandia National Laboratories
Albuquerque, NM
srajama@sandia.gov

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

***Abstract*—We introduce XTRAPULP, a new distributed-memory graph partitioner designed to process trillion-edge graphs. XTRAPULP is based on the scalable label propagation community detection technique, which has been demonstrated as a viable means to produce high quality partitions with minimal computation time. On a collection of large sparse graphs, we show that XTRAPULP partitioning quality is comparable to state-of-the-art partitioning methods. We also demonstrate that XTRAPULP can produce partitions of real-world graphs with billion+ vertices in minutes. Further, we show that using XTRAPULP partitions for distributed-memory graph analytics leads to significant end-to-end execution time reduction.**

I. 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 [28]. Graph partitioning is an essential preprocessing step to ensure load-balanced computation and to reduce inter-node communication in parallel applications [7], [26]. 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 [21], [35], and others are referred to as "power-law" graphs [2], [14].

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. Traditional tools for partitioning graphs (e.g., ParMETIS) are limited either in the size of the graphs they can partition or the partitioning objective metrics that they support. Further, as the 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 these 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 [6], label propagation-based graph partitioning methods [29], [32], [34], adaptations of traditional partitioning methodologies to small-world graph instances [25], and methods using two-dimensional distributions [6]. Among these, label propagation-based techniques are the most promising in terms of meeting our requirements [29]. We consider extending these techniques for graph inputs several orders-of-magnitude larger than previously processed by any other partitioner. Problems at the trillion edge scale have been attempted only recently [11], [12], but not in the context of a problem as computationally challenging as graph partitioning.

The following are our key contributions:

- We describe XTRAPULP, a distributed-memory partitioning method that can scale to graphs with billion+ vertices and trillion+ edges. Implementing a partitioning algorithm at this scale (relative to the billion-edge scale) requires careful consideration to computation, communication, and memory requirements of the partitioner itself. Significant changes from our shared-memory partitioner PULP are required, including development of entirely new routines for in-memory graph storage, inter-node communication, and processing of part assignment updates.
- We demonstrate the scalability of our MPI+OpenMP parallel partitioner by running on up to 131,072 cores of the NCSA Blue Waters supercomputer, using graph instances with up to 17 billion vertices and 1.1 trillion edges.
- We demonstrate state-of-the-art partitioning quality for computing partitions satisfying multiple constraints and optimizing for multiple objectives simultaneously. We show comparable quality relative to PULP, ParMETIS, and Meyerhenke et al. [25].
- We utilize partitions from XTRAPULP in two settings. First, we demonstrate reduction in end-to-end time for six graph analytics with various performance characteristics. Second, we show reduction in time for parallel sparse matrix vector multiplications with two dimensional matrix layouts calculated from XTRAPULP's vertex partitions.

Our XTRAPULP code is publicly available on GitHub (https://github.com/HPCGraphAnalysis/PuLP). The repository also contains an extended version of this paper (which we will reference in subsequent sections), including additional algorithm discussion and experimental results.

II. BACKGROUND

A. Graph Partitioning

Given an undirected graph $G = (V, E)$ and vertex and edge imbalance ratios Rat_v and Rat_e and target max part sizes Imb_v and Imb_e, the graph partitioning problem can be formally described as partitioning V into p disjoint parts. Let $\Pi = \{\pi_1, \ldots, \pi_p\}$ be a balanced partition, such that $\forall\ i = 1 \ldots p$,

$$|V(\pi_i)| \leq (1 + Rat_v)\, V_t = Imb_v \quad (1)$$

$$|E(\pi_i)| \leq (1 + Rat_e)\, E_i = Imb_e \quad (2)$$

where $V_i = |V|/p$ and $E_i = |E|/p$. $V(\pi_i)$ is the set of vertices in part π_i and $E(\pi_i)$ is the set of edges such that both its endpoints are in part π_i. We define the set of cut edges as $C(G, \Pi) = \{\{(u, v) \in E\} \mid \Pi(u) \neq \Pi(v)\}$, and set of cut edges in any part as $C(G, \pi_k) = \{\{(u, v) \in C(G, \Pi)\} \mid (u \in \pi_k \vee v \in \pi_k)\}$. Our partitioning problem is then to minimize the two metrics $|C(G, \Pi)|$ and $\max_k |C(G, \pi_k)|$.

B. Related Work

The label propagation community detection algorithm [27] is a fast and scalable method for detecting communities in large networks. Since it is fairly easy to parallelize and its output can be used for graph partitioning, label propagation has seen widespread adoption as an effective means to 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 Meyerhenke et al. [25] and Wang et al. [34]. 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 [18] at the coarsest level. Meyerhenke et al. 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. These multilevel methods have not been experimentally demonstrated to process real-world irregular graphs larger than about 3 billion edges in size or more regular synthetic networks larger than about 20 billion edges.

The second approach uses label propagation directly to compute the partitions. Early efforts utilizing this approach include Ugander et al. [32] and Vaquero et al. [33]. Wang et al. [34] additionally used a variant of their coarsening scheme to compute balanced partitions, although at a non-negligible cost to cut quality. In general, this cost was observed in early single level methods, which demonstrated good scalability and performance, but often with a high cost in terms of partition quality. In our recent prior work, we introduced PULP [28], which uses weighted label propagation variants for various stages of a multi-constraint and multi-objective shared memory parallel partitioning algorithm. Buurlage [8] 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. We only perform graph partitioning in our work due to considerably lower overheads and higher scalability relative to hypergraph partitioning. The graphs we consider are several orders of magnitude larger than those partitioned with HYPER-PULP [8].

Our work extends these two recent efforts significantly, as we strive to offer a highly performant label propagation-based distributed parallel partitioner that also computes high quality partitions of very large, irregular input graphs.

III. XTRAPULP

This section provides algorithmic and implementation details of XTRAPULP. We note explicitly that our primary contribution is technical and not algorithmic, in that 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 implementation is capable. The three main extensions needed for the distributed implementation relative to 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-task information should be stored to reduce memory overhead. Access to task-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 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.

We build upon techniques and optimizations discussed in other prior work [30]. Additionally, we offer a novel initialization strategy that is observed to substantially improve

final partition quality for certain graphs, while not negatively impacting partition quality for other graphs.

A. XTRAPULP *Overview*

a) **Graph Representation:** We use a distributed one-dimensional compressed sparse row-like representation, where each task owns a subset of vertices and their incident edges (representing a local graph $G(V, E)$). 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. Each vertex's global identifier is mapped to a task-local one using a hash map. Local to global translation uses values stored in a flat array. Each task 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 task). When computing the partition, a task will calculate updates only for its owned vertices and communicate the updates so the task's neighbors update assignments for the ghosts. Each task's memory requirements are bounded by $O(\frac{n}{t} + \frac{m}{t})$ using this representation, where n and m are the number of vertices and edges in the input graph and t is the number of processing tasks.

b) **XTRAPULP Algorithm:** We implement all of the original PULP algorithms (PULP, PULP-M, and PULP-MM from [28]) but focus our discussion on the PULP-MM algorithm for multiple objective (minimizing the global cut and maximal cut edges of any part) and multiple constraint (vertex and edge balanced) partitioning. There are three stages to the algorithm. The first stage is a fast initialization strategy which allows some imbalance among parts. The second stage balances the number of vertices for each part while minimizing the global number of cut edges. The third stage balances vertices *and* edges, and minimizes the global edge cut *and* maximal edges cut on any part. We have observed in practice that minimizing the maximal per-part cut has the side affect of also balancing the cut edges among all parts. We alternate between balance and refinement for 3 iterations during each of the latter two stages. 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. Label propagation is typically run for fixed iteration counts as convergence is not quickly guaranteed and convergence doesn't necessarily guarantee a higher quality.

B. XTRAPULP *Initialization*

We introduce the XTRAPULP initialization algorithm (Algorithm 1), a hybrid between the two shared-memory PULP initialization strategies of unconstrained label propagation [28] and breadth-first search-based graph growing [16], [19], [29]. We utilize a bulk synchronous parallel approach for all the stages, while maximizing intra-task parallelism through threading and minimizing communication load with a queuing strategy for pushing updates among tasks.

Algorithm 1 XTRAPULP Initialization:
$parts \leftarrow$ XTRAPULP-Init($G(V, E)$)

```
procid ← localTaskNum()
if procid = 0 then
    Roots(1 ... p) ← UniqueRand(1 ... |V_global|)
Bcast(Roots)
parts(1 ... |V|) ← −1
for i = 1 ... p do
    if Roots(i) ∈ V then
        parts(Roots(i)) ← i
updates ← p
while updates > 0 do
    updates ← 0
    for all v ∈ V do                                   ▷ across threads
        if parts(v) = −1 then
            isAssigned(1 ... p) ← false
            for all ⟨v, u⟩ ∈ E do
                if parts(u) ≠ −1 then
                    isAssigned(parts(u)) ← true
                    updates ← updates + 1
            w ← RandTrueIndex(isAssigned)
            if w ≠ −1 then
                Q_thread ← ⟨v, w⟩
    Q_task ← Q_thread                  ▷ merge thread into task queue
    Q_recv ← ExchangeUpdates(parts, Q_task, G)
    for all ⟨v, w⟩ ∈ Q_recv do                         ▷ across threads
        parts(v) ← w
for all v ∈ V do                                       ▷ across threads
    if parts(v) = −1 then
        parts(v) ← Rand(1 ... p)
        Q_thread ← ⟨v, parts(v)⟩
Q_task ← Q_thread                      ▷ merge thread into task queue
Q_recv ← ExchangeUpdates(G, parts, Q_task)
for all ⟨v, w⟩ ∈ Q_recv do                             ▷ across threads
    parts(v) ← w
```

The master task (process 0) first randomly selects p unique vertices from the global vertex set (array $Roots$) and broadcasts the list to other tasks. Each task initializes its local part assignments to -1, and then, if it owns one of the roots, assigns to that root a part corresponding to the order in which that root was randomly selected.

In each iteration of the primary loop of the initialization algorithm, every task considers all of its local vertices that are yet to be assigned a part using thread level parallelism. For a given unassigned local vertex v, all neighbors' part assignments (if any) are examined. Similar to label propagation, we track all parts that appear in the neighborhood ($isAssigned$); however, unlike label propagation, we randomly select one of these parts instead of assigning to v the part that has the maximal count among v's neighbors. In practice, doing so tends to result in slightly more balanced partitions.

A thread-local queue Q_{thread} is used to maintain any new part assignment to thread-owned vertices. All threads update a MPI task-level queue which is used in ExchangeUpdates(). ExchangeUpdates() also returns a queue of updates Q_{recv} for the local task's ghost vertices. We describe ExchangeUpdates() in Algorithm 2. Algorithm 1 iterates as long as tasks have updated part assignments. The number of iterations needed is on the order of the graph diameter, which can be very large for certain graph classes (e.g., road networks), leading to long execution times for this initialization stage. However, for the small-world networks that we are designing for, this issue is

minimal. For other graph classes, alternative strategies such as random or block assignments can be used.

Algorithm 2 XTRAPULP Communication Routine:
$Q_{recv} \leftarrow$ ExchangeUpdates($parts$, Q_{task}, $G(V, E)$)

$Q_{recv} \leftarrow$ ExchangeUpdates($G(V, E)$, $parts$, Q_{task})
$procid \leftarrow$ localTaskNum()
$nprocs \leftarrow$ numTasksMPI()
$sendCounts(1 \ldots nprocs) \leftarrow 0$
for all $v \in Q_{task}$ **do** ▷ across threads
  $toSend(1 \ldots nprocs) \leftarrow$ **false**
  for all $\langle v, u \rangle \in E$ **do**
    $task \leftarrow$ getTask(u)
    if $task \neq procid$ **and** $toSend(task) =$ **false then**
      $toSend(task) =$ **true**
      $sendCounts(task) \leftarrow sendCounts(task) + 2$
$sendOffsets(1 \ldots nprocs) \leftarrow$ prefixSums($sendCounts$)
$tmpOffsets \leftarrow sendOffsets$
for all $v \in Q_{task}$ **do** ▷ across threads
  $toSend(1 \ldots nprocs) \leftarrow$ **false**
  for all $\langle v, u \rangle \in E$ **do**
    $task \leftarrow$ getTask(u)
    if $task \neq procid$ **and** $toSend(task) =$ **false then**
      $toSend(task) =$ **true**
      $sendBuffer(tmpOffsets(task)) \leftarrow v$
      $sendBuffer(tmpOffsets(task) + 1) \leftarrow parts(v)$
      $tmpOffsets(task) \leftarrow tmpOffsets(task) + 2$
Alltoall($sendCounts$, $recvCounts$)
$recvOffsets(1 \ldots nprocs) \leftarrow$ prefixSums($recvCounts$)
Alltoallv($sendBuffer$, $sendCounts$, $sendOffsets$,
  Q_{recv}, $recvCounts$, $recvOffsets$)

a) **ExchangeUpdates:** This method does an Alltoallv exchange into Q_{recv}. Each task creates an array ($sendCounts$) for the number of items sent to other tasks and an array ($sendOffsets$) that has start offsets for the items being sent in the send buffer. $sendCounts$ is updated by examining all v in Q_{task} that have updated part assignments in the current iteration. The vertex and new part assignment is sent to any process in its neighborhood. We use the boolean array $toSend$ to avoid redundant communication. A prefix sum on $sendCounts$ yields $sendOffsets$.

A temporary copy of $sendOffsets$ ($tmpOffsets$) is used to loop through Q_{task} to fill the send buffer $sendBuffer$. Both loops through Q_{task} can use thread-level parallelism. The updates to the buffer, offsets, and counts arrays can either be done atomically or with thread local arrays synchronized at the end. Our implementation does the latter as it shows better performance in practice. Once $sendBuffer$ is ready, an Alltoall exchange of $sendCounts$ allows to find the number of items each task will receive ($recvCounts$). We use $recvCounts$ to create an offsets array $recvOffsets$ for the receiving buffer Q_{recv}. With all six arrays initialized, an Alltoallv exchange can be completed. We use Alltoallv exchanges for our communication methods as we expect every task to be communicating with most other tasks during execution.

C. XTRAPULP *Vertex Balancing Phase*

There can be considerable imbalance after the initialization phase. The vertex balancing stage of XTRAPULP utilizes label propagation with a weighting function W_v to achieve the balance objective. W_v is roughly proportional to the target part size Imb_v divided by the estimated current part size; its value changes as vertices are assigned to parts. We highlight the primary differences of our algorithm (Algorithm 3) from the shared memory version [28] here. We omit details for brevity, but the reasoning behind the calculation and updating the baseline weighting function W_v was provided previously [28].

Algorithm 3 XTRAPULP Vertex Balancing Phase:
$parts \leftarrow$ XTRAPULP-VertBalance($G(V, E)$, $parts$, I_{bal}, Imb_v)

$nprocs \leftarrow$ numTasksMPI()
$S_v(1 \ldots p) \leftarrow$ numVertsPerPart($1 \ldots p$)
$C_v(1 \ldots p) \leftarrow 0$
$iter \leftarrow 0$
while $iter < I_{\text{bal}}$ **do**
  $Max_v \leftarrow$ Max($S_v(1 \ldots p)$, Imb_v)
  $mult \leftarrow nprocs \times ((X - Y)(\frac{iter_{\text{tot}}}{I_{\text{tot}}}) + Y)$
  for $i = 1 \ldots p$ **do**
    $W_v(i) \leftarrow$ Max($Imb_v/(S_v(i) + mult \times C_v(i)) - 1$, 0)
  for all $v \in V$ **do** ▷ across threads
    $counts(1 \ldots p) \leftarrow 0$
    for all $\langle v, u \rangle \in E$ **do**
      $counts(parts(u)) \leftarrow counts(parts(u)) + degree(u)$
    for $i = 1 \ldots p$ **do**
      if $S_v(i) + mult \times C_v(i) + 1 > Max_v$ **then**
        $counts(i) \leftarrow 0$
      else
        $counts(i) \leftarrow counts(i) \times W_v(i)$
    $x \leftarrow parts(v)$
    $w \leftarrow$ Max($counts(1 \ldots p)$)
    if $x \neq w$ **then**
      Update($C_v(x)$,$C_v(w)$) ▷ atomic update
      Update($W_v(x)$,$W_v(w)$)
      $parts(v) \leftarrow w$
      $Q_{thread} \leftarrow \langle v, w \rangle$
  $Q_{task} \leftarrow Q_{thread}$ ▷ merge thread into task queue
  $Q_{recv} \leftarrow$ ExchangeUpdates($parts$, Q_{task}, G)
  for all $\langle v, w \rangle \in Q_{recv}$ **do** ▷ across threads
    $parts(v) \leftarrow w$
  Allreduce(C_v, SUM)
  for $i = 1 \ldots p$ **do**
    $S_v(i) \leftarrow S_v(i) + C_v(i)$
  $iter \leftarrow iter + 1$
  $iter_{\text{tot}} \leftarrow iter_{\text{tot}} + 1$

There are a few major differences between our work and that of prior methods. We do not explicitly update 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 task i in the current iteration. When updating the weights applied to each task 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:

$$S_v(i) + mult \times C_v(i)$$

This multiplier allows fine-tuned control of imbalance when running on thousands of processors in distributed-memory. This was not an issue in previous shared-memory work. The basic idea is to use the multiplier to limit how many new vertices a single task can add to a part. This prevents all tasks from calculating a high W_v value for a presently underweight part and reassigning a large number of new vertices to that

part (as there is no communication before the assignment). As the iterations progress, we linearly tighten the limit on how many updates a task can do to each part, until a final iteration, where each task 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. The multiplier is computed as

$$mult \leftarrow nprocs \times ((X - Y)(\frac{iter_{\text{tot}}}{I_{\text{tot}}}) + Y),$$

where $iter_{\text{tot}}$ is a counter of iterations performed across the outer loops during the balance-refinement stages, I_{tot} is the maximum number of iterations, and X and Y are input parameters. The function $mult$ is a linear function 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 task being allowed to add up to $4\times$ its "share" of updates to a task at an initial iteration and just its "share" at the final iteration. We discuss the selection of these X and Y parameters in our results.

D. XTRAPULP *Refinement Phase*

The XTRAPULP refinement phase greedily minimizes the global number of cut edges without exceeding the vertex target part size 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. This algorithm can be considered a variant of FM-refinement [15] or a constrained variant of baseline label propagation. The refinement algorithm is similar to the balancing algorithm, except that the $counts$ array is not weighted. Instead, the 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 that part won't increase the parts size (or estimated size with the multiplier) to larger than Max_v. The algorithm pseudocode is included in the extended version of our paper.

E. XTRAPULP *Edge Balancing Phase*

After the 3 outer iterations of the vertex balance-refinement stages, the edge balance-refinement stages begin. We don't show these algorithms for brevity, but instead will describe their differences from the vertex balance and refinement phase. For these algorithms, we use both the target number of edges (Imb_e) and vertices (Imb_v) per task. The goal is to balance the number of edges per task while not creating vertex imbalance. The vertex weighting terms $W_v(1 \ldots p)$ are replaced by edge and cut imbalance weighting terms $W_e(1 \ldots p)$ and $W_c(1 \ldots p)$ using the current global maximum edge size per part Max_e and maximum cut size per part Max_c. W_e and W_c are then used to highly weight parts that are currently underweight both in terms of the number of edges and cut edges. We weight the counts of part i with the equation:

$$counts(i) \leftarrow counts(i) \times (R_e W_e(i) + R_c W_c(i))$$

R_e and R_c initially create bias (by first linearly increasing R_e while holding R_c fixed) for parts that are underweight in the number of edges. Once the edge balance constraint has been achieved, R_e becomes fixed and R_c correspondingly increases the bias to both minimize the maximum per-part edge cut and balance cut edges among parts.

Using our multiplier for distributed-memory updates, we restrict the number of edges and cut edges transferred to any part per iteration; we use the same X and Y constants as before. However, in addition to tracking the vertex changes per part with C_v, edges (C_e) and cut edges changed per part (C_c) are also tracked and exchanged among tasks, as in Algorithm 3. 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 $I_{\text{bal}} = 5$ 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 *and* 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, and cut size.

IV. EXPERIMENTAL SETUP

We evaluate XTRAPULP performance on several small-world graphs. While XTRAPULP is not designed for regular high-diameter graphs, we do evaluate performance on several mesh and mesh-like graphs. We use graphs from the University of Florida Sparse Matrix Collection [3]–[5], [13] (indochina, arabic, it, sk, uk-2002, uk-2005, nlpkktXXX), the 10th DIMACS Implementation Challenge website [1] (uk-2007), the Stanford Network Analysis Platform (SNAP) website [23], [24], [36] (lj, orkut, friendster), the Koblenz Network Collection [22] (wikilinks, dbpedia), Cha et al. [9] (twitter), and meshes used internally in our group (InternalMeshX).

We perform large-scale evaluations on the 2012 Web Data Commons hyperlink graph[1], which is created from the Common Crawl web corpus[2]. This graph contains 3.56 billion vertices and 128 billion edges, and is the largest publicly available real-world graph known to us. 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) [10] 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 two 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. 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

[1] http://webdatacommons.org/hyperlinkgraph/

[2] http://commoncrawl.org

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 against ParMETIS version 4.0.3 [20] and PULP version 0.1 [28]. 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* and *Compton*. Unless otherwise specified, we use one MPI task per compute node for multi-node parallel runs of XTRAPULP, and set the number of OpenMP threads to the number of shared-memory cores.

V. RESULTS

A. Performance and Scalability

1) Scaling on Blue Waters*:* We first analyze XTRAPULP performance when running in a massively parallel setting on the *Blue Waters* supercomputer. Figure 1 (left) gives the execution time for partitioning the real-world Web Data Commons hyperlink graph (WDC12) and three generated graphs (RMAT, RandER, RandHD) of nearly the same size (3.56 billion vertices and 128 billion edges). We run on 256-2048 nodes of *Blue Waters* (4096-32768 cores), and compute 256 parts.

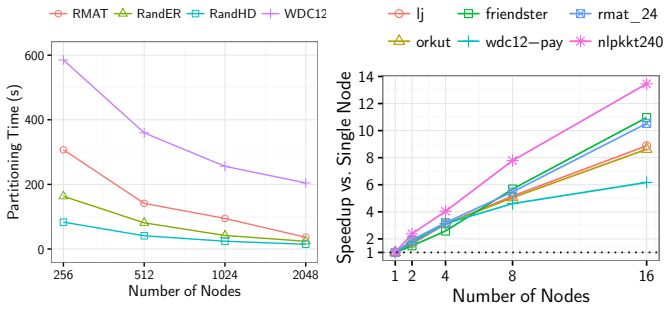

Fig. 1. XTRAPULP parallel performance (*strong scaling*) results on *Blue Waters* for computing 256 parts of various test graphs (left) and speedups on *Compton* for computing 16 parts of various test graphs (right).

As shown in Figure 1, XTRAPULP exhibits good *strong scaling* up to 2048 nodes on all tested graphs. The speedups achieved are 2.9×, 8.4×, 6.8×, and 5.7× for WDC12, RMAT, RandER, and RandHD graphs, respectively, when going from 256 to 2048 nodes (8× increase in parallelism). As expected, we see better speedups for the synthetic graphs due to better computational and communication load balance. The running times depend on the initial vertex ordering. The partitioning time for the RandHD network on 256 nodes is nearly $\frac{1}{7}$ the partitioning time for WDC12, even though the graphs are the same size. This is due to significantly lower inter-node communication time (which relates to the initial edge cut) in both the vertex and edge balancing steps.

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

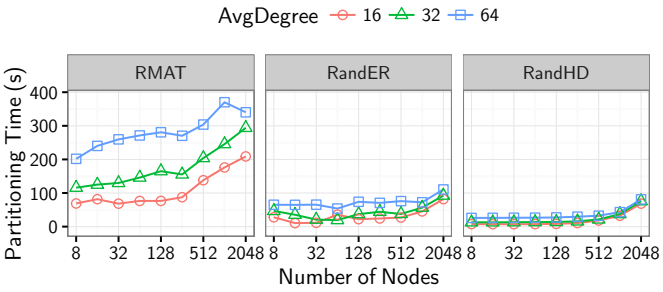

Fig. 2. XTRAPULP parallel performance (*weak scaling*) results on *Blue Waters* for various RMAT, RandER, and RandHD graphs. The number of graph vertices per node is $\approx 2^{22}$. The number of parts computed is set to number of nodes.

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 rise in times beyond 256 nodes. As graph size increases in RMAT graphs, so does the maximum degree, and vertices with high degrees lead to computation imbalance with the one-dimensional graph distribution used in XTRAPULP.

2) Trillion Edge Runs: We ran additional experiments on up to 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. The largest RMAT graph we could partition on 8192 nodes had half as many edges (2^{34} vertices and 2^{39} edges); it took 608 seconds. XTRAPULP strong and weak scaling results on *Blue Waters* demonstrate that there are no performance-crippling bottlenecks at scale in our implementation.

3) Scaling on Compton*:* We extensively test XTRAPULP at a smaller scale (16 nodes of *Compton*), for direct performance comparisons to ParMETIS and PULP. For MPI-only ParMETIS, we run 16, 8, 4, and 1 tasks per node and report the best time in order to provide a conservative comparison. OpenMP-only PULP results are with full threading on a

TABLE I
XTRAPULP, PULP, AND PARMETIS PARALLEL PERFORMANCE RESULTS ON *Compton* FOR COMPUTING 16 PARTS OF VARIOUS TEST GRAPHS. XTRAPULP AND PULP RESULTS INCLUDE 16-WAY MULTITHREADED PARALLELISM, AND PARMETIS RESULTS ARE THE BEST ONES OBTAINED WITH 16- TO 256-WAY MPI TASK CONCURRENCY. †/‡ SYMBOLS INDICATES RELATIVE SPEEDUP WITH RESPECT TO 2/4-NODE XTRAPULP RUNS. WE ALSO INCLUDE THE SCALE OF THE GRAPH IN MILLIONS OF VERTICES (N) AND MILLIONS OF EDGES (M).

Graph	n	m	Partitioning Time (s) XTRAPULP (16 nodes)	PULP (1 node)	ParMETIS (16 nodes)	XTRAPULP Speedup vs PULP	Rel. to 1 node
lj	5.4	69	**4.9**	10	59	2.0×	8.9×
orkut	3.1	117	**4.8**	18	110	3.8×	8.6×
friendster	66	1806	**232**	1672		**7.2**×	11 ×
twitter	53	1963	**1647**	3611		2.2×	2.3†×
wikilinks	26	601	**137**	467		3.4×	5.9×
dbpedia	67	258	**35**	70		2.0×	**14** ×
indochina	7.3	149	**4.4**	8.1	130	1.8×	11.3×
arabic	23	552	**12**	16	754	1.3×	8.2×
it	41	1151	**22**	32		1.4×	9.4×
sk	51	1949	**33**	67		2.1×	9.1×
uk-2002	1.8	298	**5.1**	9.2	85	1.8×	**12.8**×
uk-2005	39	781	**18**	34		1.9×	9.8×
uk-2007	106	3302	**49**	71		1.4×	3.9†×
wdc12-pay	39	623	**241**	1062		4.4×	6.2×
wdc12-host	89	2043	**422**	2443		**5.7**×	8.6×
rmat_22	4.2	67	**6.7**	14	126	2.1×	4.9×
rmat_24	17	268	**30**	147	923	4.9×	10.5×
rmat_26	67	1074	**183**	1022		5.6×	**12.5**×
rmat_28	268	4295	**981**	5454		**5.6**×	3.2‡×
InternalMesh1	0.3	3.5	**0.1**	0.1	0.6	1.6×	20.0×
InternalMesh2	2.2	28	**0.5**	0.9	0.7	2.0×	23.4×
InternalMesh3	18	220	2.9	6.8	**1.2**	2.3×	**27.9**×
InternalMesh4	140	1819	24	46	**4.6**	1.9×	26.7×
nlpkkt160	8.3	112	1.6	3.8	**1.5**	2.4×	11.5×
nlpkkt200	16	216	2.6	6.4	**2.2**	**2.5**×	13.6×
nlpkkt240	28	373	4.6	11	**3.6**	2.4×	13.5×

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.

We present 16-node performance results in Table I. Empty cells in the table indicate cases where ParMETIS failed to run to completion due to out-of-memory and related errors on some MPI task. We indicate in bold font the best timing results for each graph, and the best XTRAPULP absolute speedup (with respect to single-node PULP) and relative speedup results in each of the four graph classes (interaction, web crawl, R-MAT, and mesh). The single-node shared-memory PULP is consistently faster than distributed-memory parallel ParMETIS for the first three classes of graphs. For the fourth class of regular, high-diameter graphs, ParMETIS outperforms PULP and XTRAPULP; ParMETIS is optimized to partition these types of graphs.

For all of the small-world graphs, 16-node XTRAPULP running times are better than single-node PULP running times. XTRAPULP and PULP have several key algorithmic differences, and single-node PULP is faster than single-node XTRAPULP. We omit a direct comparison between single node performance, but these values can be inferred through the last two columns in the table. We created XTRAPULP in order to scale to multi-node settings, and we see that the 16-node speedup (with respect to single-node XTRAPULP) is quite good, being 14× for dbpedia 12.8× for uk-2002. This is despite XTRAPULP being communication-intensive and Blue Waters having only a 3D-torus network; we performed a subset of additional runs on a 5D-torus Blue Gene/Q and observed near perfect speedup for a majority of the graphs in Table I. We consider the speedup relative to PULP to be also quite good, considering the difficulties and overheads in reformulating an asynchronous shared-memory algorithm into a synchronous distributed-memory implementation. E.g., in the current (June 2016) version of the graph500.org benchmark, the per-core performance ratio between the fastest shared-memory implementation and fastest distributed-memory implementation is approximately 6.5×; our ratios are of a similar order, being between 11× for it and uk-2007 and only 2.2× for friendster, despite our implementation not being as finely optimized as the Graph500 benchmark code.

Figure 1 (right) shows XTRAPULP strong scaling for six representative graphs. Note that graph sizes vary significantly, ranging from the 69 million-edge lj graph to the 1.8 billion-edge friendster graph. We observe a range of relative speedups, attributable to the graph structure. There appear to be no intrinsic scaling bottlenecks even at this smaller 16-node scale.

B. Partitioning Quality

We next evaluate XTRAPULP partitioning quality by comparing results to PULP and ParMETIS. We use the two architecture-independent metrics for quality comparisons: Edge cut ratio (number of edges cut divided by the number of edges) and Scaled max edge cut ratio (maximum over all parts of the ratio of the number of edges cut to the average number of edges per part). For both metrics, lower values are preferred. These two metrics correspond to the objectives that the three partitioning methods optimize. In Figure 3, we report these metrics, varying the number of parts from 2 to 256 as quality results can vary with number of parts. We use the same six representative graphs that were used for strong scaling experiments on *Compton*. Again note that we perform analysis at this scale only for relative comparison. At the scale for which XTRAPULP is designed, the only competing methods are random and block partitioning; random partitioning produces an edge cut ratio that scales approximately as $\frac{p-1}{p}$, where p is the number of parts, while the quality of block partitioning is highly variable and dependent on how the graph is stored.

Our first observation is that both quality metrics – edge cut ratio and scaled max cut ratio – can vary dramatically based on the graph structure. The quality results for nlpkkt240 are in stark contrast to the rest of the graphs. On increasing the number of parts, the two metrics increase only slightly for nlpkkt240, but at a much faster rate for the rest of the graphs. The edge cut ratio quickly approaches 1.0 (all edges cut) with increasing part count when partitioning rmat_24. For graphs with intrinsically high edge cut ratios, any quality gains must be assessed taking partitioning running times into consideration. Ideally, the partitioning method should finish quickly for

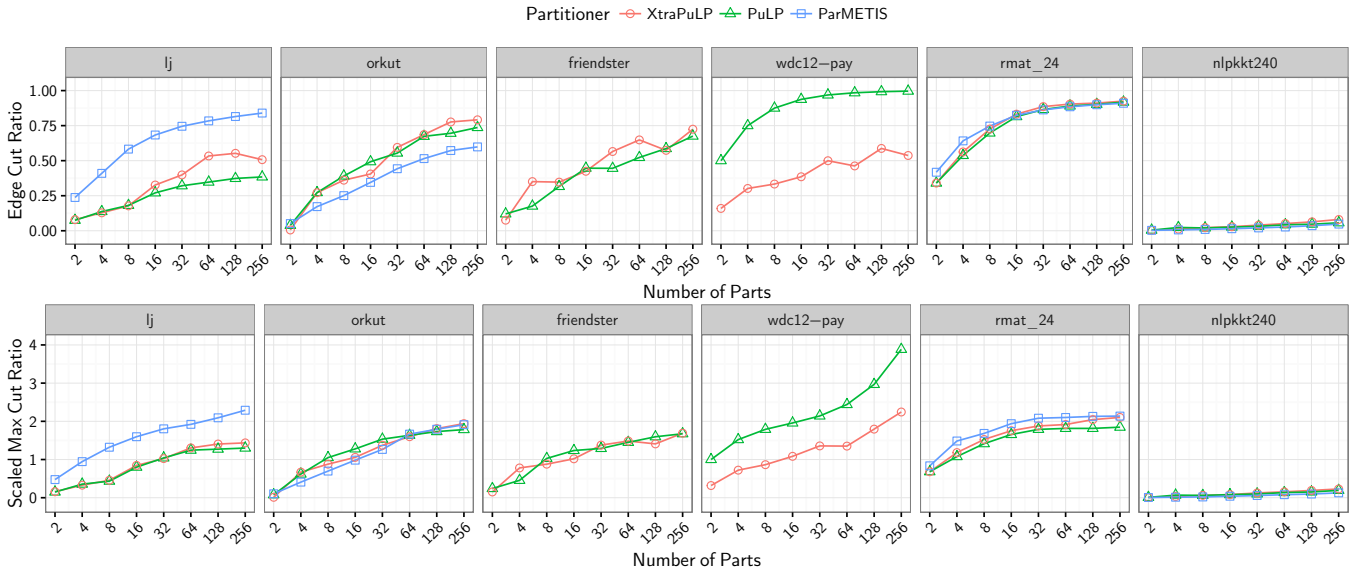

Fig. 3. Partition quality comparison when varying the number of parts computed. For both Edge Cut ratio and Scaled Max Cut ratio, lower values are better.

cases where quality metrics cannot be substantially improved. Comparing PuLP and XtraPuLP results, we observe that the metrics are relatively close, despite the asynchronous intra-task updates in XtraPuLP. We observe much better performance for XtraPuLP on the wdc-pay graph, likely due to the novel initialization strategy. ParMETIS fails to run for two of the six graphs. XtraPuLP outperforms ParMETIS on lj, whereas ParMetis does slightly better on orkut.

To numerically quantify quality gains/losses, we compute "performance ratios" for all partitioners over all tested graphs in Table I for computing 2-256 parts. Here, the performance ratio is defined as the geometric mean over all tests of each partitioner's edge cut or max per-part cut, divided by the best edge cut or max per-part cut for that test (graph and number of parts). A lower value is better, with a ratio of exactly 1.0 indicating that the partitioner produced the best quality for every single test. We calculate performance ratios for edge cut to be 1.18, 1.33, and 1.37 and max per-part cut to be 1.19, 1.40, and 1.41 for ParMETIS, PuLP, and XtraPuLP, respectively. When we consider only the irregular graphs for which ParMETIS completes, the values are much closer, with edge cut ratios of 1.36, 1.36, and 1.46 and max per-part cut ratios of 1.39, 1.43, and 1.49 for ParMETIS, PuLP, and XtraPuLP, respectively. We thus claim that partitioning quality is not compromised for small-world graphs when using XtraPuLP. XtraPuLP also provides users the ability to partition large graphs that do not fit on a single node, and achieves good strong and weak scaling.

Our final quality experiment on *Blue Waters* measures how partition quality varies with large-scale parallelism. We examine how the edge cut ratio and partition edge imbalance vary when partitioning WDC12 into 256 parts using 256-2048 nodes. We observe the edge cut ratio produced from XtraPuLP to vary between 0.04 and 0.07, which is considerably lower than the values of 0.16 for vertex block partitioning and almost 1.0 for random partitioning. Note that the relatively low edge cut here for block partitioning is a result of the crawling method, but it comes at a high cost: the edge imbalance ratio is 1.85. Our edge imbalance ratio stays under 10% for all node counts. We plot the full results in the extended version of our paper.

C. Additional Comparisons

Here we provide additional comparisons to the recent state-of-the-art partitioner of Meyerhenke et al. [25] (KaHIP), which uses size-constrained label propagation during the graph coarsening phase. This partitioner solves the single-constraint and single-objective graph partitioning problem, optimizing for edge cut and balancing vertices per part. Therefore, we modify our XtraPuLP code by eliminating the edge balancing and max per-part cut phase to provide a direct comparison. We also run shared-memory PuLP and ParMETIS. All codes are run using 16-way parallelism to allow a direct comparison to shared-memory PuLP. We partition 2-256 parts of the lj, rmat_22, and uk-2002 graphs with a 3% load imbalance constraint. In Figure 4, we compare edge cut (top) and execution time (bottom).

Overall, we observe XtraPuLP to be within a small fraction of the Meyerhenke et al. and ParMETIS codes in terms of part quality, while running only slightly slower than shared-memory PuLP. This is despite XtraPuLP being designed and optimized for the multi-objective multi-constraint problem. Performance ratios for cut quality on this limited test set are 1.05 for Meyerhenke et al., 1.23 for ParMETIS, 1.51 for PuLP, and 1.61 for XtraPuLP. Performance ratios for execution time are 1.27 for PuLP, 1.73 for XtraPuLP, 11.81 for ParMETIS, and 26.5 for Meyerhenke et al. Although we only report the results for 16-way parallelism, testing at a larger scale between XtraPuLP, ParMETIS, and Meyerhenke et al. reveal similar relative performance. These results in general demonstrate the efficiency tradeoff between quality and time to solution, the choice for which to optimize being

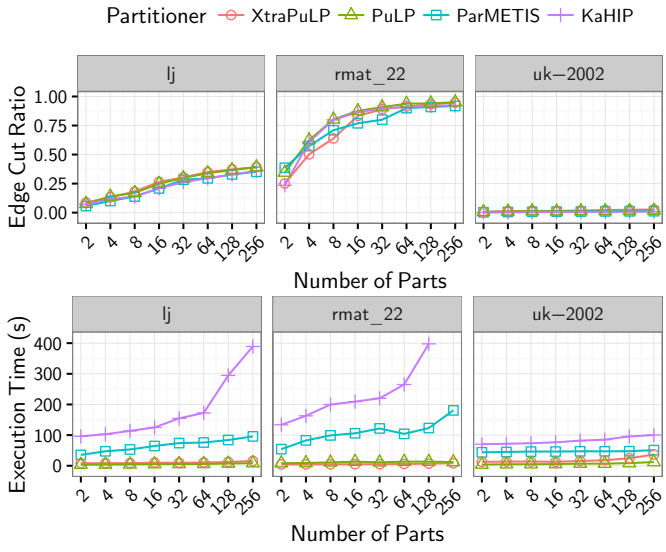

Fig. 4. Partitioning quality (top) and execution time (bottom) for multiple partitioners solving the single objective single constraint partitioning problem.

application-dependent. However, we emphasize again that we provide these results only to establish a relative baseline for comparison of the performance of XTRAPULP, as the engineering decisions driving its design were made to enable scalability to partition graphs several orders-of-magnitude larger than the graphs presented here.

D. Multiplier Parameters

We also analyze the effect that varying the X and Y parameters have on the final partition quality. Using lj, uk-2002, rmat_22, and nlpkkt160 as representative examples for each graph class, we computed from 2-128 parts each on 2-16 compute nodes of *Compton* (all powers of 2 in between). We examine the effect of varying both X and Y between 0 and 4 on edge cut, max cut, vertex balance, and edge balance. We observe multiple trends. A lower X and Y indicates a higher quality cut. This is because lower values allow the highest number of part reassignments and therefore the greatest overall refinement. We also note that a higher X value relative to Y will, on average, also result in a better cut. This is due to how a higher initial limit on part reassignments (Y) and a lower final limit (X) enables considerable refinement during the initial iterations while limiting the potential imbalance possible on the final iterations. In general, the degree of balance is inversely proportional with the quality of cut. The optimal X, Y pair of values should therefore be selected along some *threshold* where high quality and balance are concurrently achieved. We selected our test values of $X = 1.0$ and $Y = 0.25$ empirically, as they gave us the overall best quality in terms of cut and balance on our test suite. Plots demonstrating overall trends for cut quality and balance relative to varying X and Y are shown in the extended version of our paper.

E. Applications

We next demonstrate that XTRAPULP can significantly improve performance of real-world analytics. Consider analytics

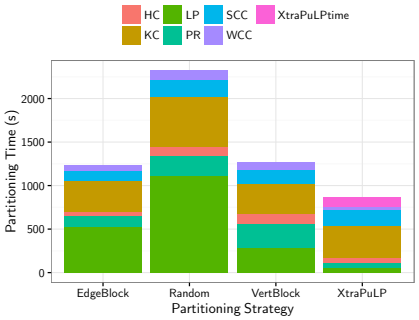

Fig. 5. The parallel performance results of various parallel graph analytics (HC, KC, LP, PR, SCC, WCC) on 256 nodes of *Blue Waters*, executed on the WDC12 graph with different graph partitioning strategies.

optimized to run on the 128 billion edge WDC12 [31]. Without a partitioner that can process graphs of this size, 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 5, we give the execution times of six analytics on WDC12 with four partitioning strategies. EdgeBlock partitioning stores a contiguous set of vertices and all their adjacencies on each task such that each task has approximately the same number of edges. VertexBlock partitioning stores roughly the same number of vertices and all their adjacencies on each task. Random partitioning assigns vertices to tasks randomly. XTRAPULP assigns vertices based on the computed partition. The six analytics considered (algorithms presented in [30]) are Harmonic Centrality (HC) computation of 100 vertices, approximate K-core decomposition (KC), Label Propagation-based community detection (LP), PageRank (PR), strongly connected component decomposition (SCC), and weakly connected components decomposition (WCC). For XTRAPULP, we *include* the partitioning time in comparisons.

Using balanced XTRAPULP partitions reduces end-to-end execution time 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), but this can be explained: We suspect that the way vertex block and edge block retain the original vertex ordering of the WDC12 graph, which is highly clustered as a result of the crawling methodology, has a beneficial impact on the number of iterations and total time required to complete these algorithms.

We also examine partitioning impact on sparse matrix vector multiplication (SpMV) by using the Epetra package of the Trilinos scientific computing library [17] to perform 100 SpMV operations on the six graphs used in Figure 3. We use several partitioning strategies, including one dimensional vertex block, random, ParMETIS, and XTRAPULP. We also utilize 2D distributions with vertex block and random partitions. Additionally, using a strategy for mapping 1D partitions into 2D distributions [6], we run with 2D distributions produced from our 1D ParMETIS and XTRAPULP partitions. We

run these tests on 1, 8 and 16 nodes of *Compton* with 16, 128, and 256 MPI ranks, respectively. We observe a **2.77**× (geometric mean) reduction in execution time when using 2D-XTRAPULP relative to 1D-Rand for 256-way parallel code, and 2D-XTRAPULP results in the best performance in **60**% of the total tests. A table of the full results is shown in the extended version of our paper.

VI. Conclusion

We demonstrate how XTRAPULP can scale to graphs several orders-of-magnitude larger than prior work. This work significantly extended our prior shared-memory-only partitioner, PULP. We show comparable partition quality to prior methods at the small scale, and, at the large scale, we significantly improve upon the existing competing methods, block and random partitioning. We also demonstrate faster execution times and comparable parallel efficiency relative to the state-of-the-art. Using partitions computed by XTRAPULP, we also improve performance on highly tuned matrix-vector multiplication kernels and several graph analytics running on the current largest publicly available web crawl.

Acknowledgment

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work is also supported by NSF grants ACI-1253881, CCF-1439057, and the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. We also thank Meyerhenke, Sanders, and Schulz for providing the source code for their partitioner [25].

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*, R. Alhajj and J. Rokne, Eds., 2014, pp. 73–82.

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

[3] 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.

[4] 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.

[5] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. Int'l. World Wide Web Conf. (WWW)*, 2004.

[6] 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.

[7] 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., 2016, pp. 117–158.

[8] J. Buurlage, "Self-improving sparse matrix partitioning and bulk-synchronous pseudo-streaming," Master's thesis, Utrecht University, 2016.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] A. George and J. W. Liu, *Computer solution of large sparse positive definite systems*. Prentice Hall, 1981.

[17] 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.

[18] 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.

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

[20] ——, "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.

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

[22] J. Kunegis, "KONECT - the Koblenz network collection," konect.uni-koblenz.de, last accessed Feb 2017.

[23] 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.

[24] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[25] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," in *Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, 2015.

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

[27] 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.

[28] 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.

[29] ——, "Complex network partitioning using label propagation," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S620–S645, 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] ——, "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.

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

[33] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "xDGP: A dynamic graph processing system with adaptive partitioning," arXiv: 1309.1049 [cs.DC], 2013.

[34] 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.

[35] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[36] 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.