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.**

***Index Terms*—graph partitioning; label propagation; distributed-memory processing.**

I. INTRODUCTION

We introduce XTRAPULP, a new graph partitioner exploiting distributed-memory parallelism plus threading to efficiently partition extreme-scale real-world graphs. XTRAPULP can be considered a significant extension to our prior shared-memory-only partitioner, PULP [27]. Graph partitioning is an essential preprocessing step to ensure load-balanced computation and to reduce inter-node communication in parallel applications [7], [25]. With the sizes of online social networks, web graphs, and other non-traditional graph data sets (e.g. brain graphs) growing at an exponential pace, scalable and efficient algorithms are necessary to partition and analyze them. Online social networks and web crawls 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], [34], 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 methods 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 can support. Furthermore, as the analytics themselves are often quite fast in comparison to other 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, scale better on irregularly structured inputs, and scale to emerging real-world problem sizes. We also desire the method to (1) be more memory-efficient than partitioning methods used for traditional scientific computing problems; (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 that are based on random or edge-based distributions [36], label propagation-based graph partitioning methods [31], [33], adaptations of traditional partitioning methodologies to small-world graph instances [24], and methods using two-dimensional distributions [6]. Among these, label propagation-based techniques are the most promising in terms of meeting our requirements. 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 e.g. a billion edges, the limit of traditional methods) 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 and ParMETIS.
- 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.

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 $\forall\ i = 1 \ldots p$,

$$|V(\pi_i)| \quad \leq (1 + Rat_v)\,\frac{|V|}{p} \quad = Imb_v \qquad (1)$$

$$|E(\pi_i)| \quad \leq (1 + Rat_e)\,\frac{|E|}{p} \quad = Imb_e \qquad (2)$$

$V(\pi_i)$ 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. Label Propagation

The label propagation community detection algorithm [26] is a fast and scalable method for detecting communities in large networks. The primary motivation for using label propagation for the community detection problem is that its per-iteration cost is linear in the graph size, or $O(|V|+|E|)$. Label propagation is also shown to find communities in a constant number of iterations on several real-world graphs. Community detection methods also naturally lend themselves towards use in partitioning, as the optimization problems solved are similar. Community detection algorithms attempt to find tightly knit communities having a high relative portion of internal edges among members of the community versus external edges to members of other communities. This goal corresponds to a partitioning method attempting to separate a graph into some number of parts where each part has a high number of internal and few external (cut) edges.

C. Related Work

Due to its linear work bound, scalability, and similar optimization goal, label propagation has seen relatively 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. In multilevel partitioning, an input graph is iteratively coarsened to a much smaller graph, the coarse graph is partitioned, and then iterative stages of uncoarsening and partition refinement take place until a partition of the original graph is produced. Partitioners that utilize these techniques include Meyerhenke et al. [24] and Wang et al. [33]. 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 these methods, and multilevel methods in general, are the high memory requirements that result from having to store copies of the graph at the multiple levels of coarsening, and the need to use poorly scaling or serial partitioning methods at the coarsest level. Multilevel methods have not been experimentally demonstrated to process irregular graphs larger than approximately $O(1\ billion)$ edges in size.

The second approach uses label propagation directly to compute the partitions. Early efforts utilizing this approach include Ugander et al. [31] and Vaquero et al. [32]. Wang et al. [33] 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 [27], 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're considering in our work are over 20 million times larger than those partitioned with Hyper-PuLP.

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, our distributed-memory label propagation-based partitioner. 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. 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 the global quantities required by out weighting functions. We implement highly 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 use a dynamic multiplier $mult$ that iteratively adjusts to enable partitions to attain balance in a more stable fashion. We also analyze the parameters controlling $mult$ and its effect on partition quality and achieving the balance constraints.

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 and 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-specific 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.

b) **XtraPuLP Algorithm:** We implement and optimize the original PuLP-MM algorithm from [27] for multiple objective (minimizing the global cut and maximal cut edges of any part) and multiple constraint (vertex and edge balance) partitioning. An overview of the three stage algorithm is given in Algorithm 1. The first stage is a fast initialization strategy allowing large imbalance among partitions. The second stage balances the number of vertices for each part while minimizing the global number of cut edges. The final 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 stages for I_{outer} iterations. Each balance and refinement algorithm iterates I_{bal} or I_{ref} times internally. We also track $Iter_{\text{tot}}$ and I_{tot} as part of our distributed communication strategy (explained below). The default values for I_{outer}, I_{bal}, and I_{ref} are shown in Algorithm 1 and used in all experiments.

Algorithm 1 XtraPuLP Multi-Constraint Multi-Objective Algorithm

```
procedure XtraPuLP($G(V, E)$)
    $parts \leftarrow$ XtraPuLP-Init($G(V, E)$)
    $I_{\text{outer}} \leftarrow 3$    $I_{\text{bal}} \leftarrow 5$    $I_{\text{ref}} \leftarrow 10$
    $I_{\text{tot}} \leftarrow I_{\text{outer}} \times (I_{\text{bal}} + I_{\text{ref}})$
    $Iter_{\text{tot}} \leftarrow 0$
    $Imb_v \leftarrow$ targetMaxVerticesPerPart()
    $Imb_e \leftarrow$ targetMaxEdgesPerPart()
    for $iter = 1 \ldots I_{\text{outer}}$ do
        XtraPuLP-VertBalance($G(V, E), parts, I_{\text{bal}}, Imb_v$)
        XtraPuLP-VertRefine($G(V, E), parts, I_{\text{ref}}, Imb_v$)
    $Iter_{\text{tot}} \leftarrow 0$
    for $iter = 1 \ldots I_{\text{outer}}$ do
        XtraPuLP-EdgeBalance($G(V, E), parts, I_{\text{bal}}, Imb_v, Imb_e$)
        XtraPuLP-EdgeRefine($G(V, E), parts, I_{\text{ref}}, Imb_v, Imb_e$)
    return $parts$
```

B. XtraPuLP *Initialization*

We introduce the XtraPuLP initialization algorithm (Algorithm 2), a hybrid between the two shared-memory PuLP initialization strategies of unconstrained label propagation [27] and breadth-first search-based graph growing [16], [19], [28]. 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.

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 at the end of initialization.

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 3. Algorithm 2 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

Algorithm 2 XtraPuLP Initialization:
$parts \leftarrow$ XtraPuLP-Init($G(V, E)$)

```
$procid \leftarrow$ localTaskNum()
**if** $procid = 0$ **then**
    $Roots(1 \ldots p) \leftarrow$ UniqueRand($1 \ldots |V_{global}|$)
Bcast($Roots$)
$parts(1 \ldots |V|) \leftarrow -1$
**for** $i = 1 \ldots p$ **do**
    **if** $Roots(i) \in V$ **then**
        $parts(Roots(i)) \leftarrow i$
$updates \leftarrow p$
**while** $updates > 0$ **do**
    $updates \leftarrow 0$
    **for all** $v \in V$ **do**                              ▷ across threads
        **if** $parts(v) = -1$ **then**
            $isAssigned(1 \ldots p) \leftarrow$ **false**
            **for all** $\langle v, u \rangle \in E$ **do**
                **if** $parts(u) \neq -1$ **then**
                    $isAssigned(parts(u)) \leftarrow$ **true**
                    $updates \leftarrow updates + 1$
            $w \leftarrow$ RandTrueIndex($isAssigned$)
            **if** $w \neq -1$ **then**
                $Q_{thread} \leftarrow \langle v, w \rangle$
    $Q_{task} \leftarrow Q_{thread}$                    ▷ 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
        $parts(v) \leftarrow w$
**for all** $v \in V$ **do**                                 ▷ across threads
    **if** $parts(v) = -1$ **then**
        $parts(v) \leftarrow$ Rand($1 \ldots p$)
        $Q_{thread} \leftarrow \langle v, parts(v) \rangle$
$Q_{task} \leftarrow Q_{thread}$                        ▷ merge thread into task queue
$Q_{recv} \leftarrow$ ExchangeUpdates($G$, $parts$, $Q_{task}$)
**for all** $\langle v, w \rangle \in Q_{recv}$ **do**                    ▷ across threads
    $parts(v) \leftarrow w$
```

small-world networks that we are considering, this issue is minimal. For other graph classes, alternative strategies such as random or block assignments can be used.

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.

Algorithm 3 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*)
```

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 4) from the shared memory version [27] here. We omit details for brevity, but the reasoning behind the calculation and updating the baseline weighting function W_v was provided previously [27].

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

```
Algorithm 4 XtraPuLP Vertex Balancing Phase:
$parts \leftarrow$ XtraPuLP-VertBalance$(G(V,E), parts, I_{\text{bal}}, Imb_v)$
    $nprocs \leftarrow$ numTasksMPI()
    $S_v(1 \dots p) \leftarrow$ numVertsPerPart$(1 \dots p)$
    $C_v(1 \dots p) \leftarrow 0$
    $iter \leftarrow 0$
    while $iter < I_{\text{bal}}$ do
        $Max_v \leftarrow$Max($S_v(1 \dots p)$, $Imb_v$)
        $mult \leftarrow nprocs \times ((X - Y)(\frac{iter_{\text{tot}}}{I_{\text{tot}}}) + Y)$
        for $i = 1 \dots p$ 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
            $counts(1 \dots p) \leftarrow 0$
            for all $\langle v, u \rangle \in E$ do
                $counts(parts(u)) \leftarrow counts(parts(u))$ + degree($u$)
            for $i = 1 \dots p$ 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($counts(1 \dots p)$)
            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 $i = 1 \dots p$ do
            $S_v(i) \leftarrow S_v(i) + C_v(i)$
        $iter \leftarrow iter + 1$
        $iter_{\text{tot}} \leftarrow iter_{\text{tot}} + 1$
```

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 each of the two outer loops in Algorithm 1, I_{tot} is the maximum number of iterations allowed, 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× its "share" of updates to a task at an initial iteration and just its "share" at the final iteration.

We will show experimentally that X and Y values close to zero results in wild imbalance swings, as the currently most underweight part can get a high number of new vertices from each task, becoming overweight. We will also show that X and Y values greater than about 1.5 can have a large negative impact on overall partition quality in terms of edge cut. Additionally, it is usually desired that X be larger than Y, as this allows a larger number of part assignment updates on initial iterations to improve overall cut while limiting updates on later iterations to allow a lower final imbalance.

```
Algorithm 5 XtraPuLP Refinement Phase:
$parts \leftarrow$ XtraPuLP-VertRefine$(G(V,E), parts, I_{\text{ref}}, Imb_v)$
    $nprocs \leftarrow$ numTasksMPI()
    $S_v(1 \dots p) \leftarrow$ numVertsPerPart$(1 \dots p)$
    $C_v(1 \dots p) \leftarrow 0$
    $iter \leftarrow 0$
    while $iter < I_{\text{ref}}$ do
        $Max_v \leftarrow$Max($S_v(1 \dots p)$, $Imb_v$)
        $mult \leftarrow nprocs \times ((X - Y)(\frac{iter_{\text{tot}}}{I_{\text{tot}}}) + Y)$
        for all $v \in V$ do                                      ▷ across threads
            $counts(1 \dots p) \leftarrow 0$
            for all $\langle v, u \rangle \in E$ do
                $counts(parts(u)) \leftarrow counts(parts(u)) + 1$
            for $i = 1 \dots p$ do
                if $S_v(i) + mult \times C_v(i) + 1 > Max_v$ then
                    $counts(i) \leftarrow 0$
            $x \leftarrow parts(v)$
            $w \leftarrow$ Max($counts(1 \dots p)$)
            if $w \neq x$ then
                Update($C_v(x)$,$C_v(w)$)                         ▷ atomic updates
                $parts(v) \leftarrow w$
                $Q_{thread} \leftarrow \langle v, w \rangle$
        $Q_{task} \leftarrow Q_{thread}$                      ▷ merge thread into task queue
        $Q_{recv} \leftarrow$ ExchangeUpdates($G$, $parts$, $Q_{task}$)
        for all $\langle v, w \rangle \in Q_{recv}$ do                       ▷ across threads
            $parts(v) \leftarrow w$
        Allreduce($C_v$, SUM)
        for $i = 1 \dots p$ do
            $S_v(i) \leftarrow S_v(i) + C_v(i)$
        $iter \leftarrow iter + 1$
        $iter_{\text{tot}} \leftarrow iter_{\text{tot}} + 1$
```

D. XtraPuLP *Refinement Phase*

After I_{bal} iterations of the balancing phase, XtraPuLP uses I_{ref} iterations of the refinement phase (Algorithm 5). The 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 Max($S_v(1 \dots p)$). 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.

E. XtraPuLP *Edge Balancing Phase*

After I_{outer} iterations of vertex balance-refinement phases, the edge balance-refinement stages begin. We don't show these algorithms for brevity, but instead will describe their differences from Algorithms 4 and 5. 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 $W_v(1 \dots p)$ 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. 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 4. 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 stage similar to Algorithm 5 is used. The only change in this stage 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. Table I lists the graphs used from the University of Florida Sparse Matrix Collection [3]–[5], [13], the 10th DIMACS Implementation Challenge website [1], the Stanford Network Analysis Platform (SNAP) website [23], [30], [35], the Koblenz Network Collection [22], and Cha et al. [9]. Meshes used internally in our group are listed as InternalMeshX. Wherever applicable, we list statistics for the graphs. The maximum vertex degree and the graph diameter can have considerable performance impact on the label propagation and breadth first search steps that comprise our algorithm. For test instances that did not include an approximate diameter estimate, we estimate it using 10 iterative breadth first searches with a vertex randomly selected from the farthest level on the previous search. We treat all graph edges as undirected edges.

Table I has six sections. The first section lists four graphs that are snapshots of online social and communication networks (lj, orkut, friendster, twitter) and two hyperlink graphs (wikilinks, dbpedia). The next section includes nine web crawls of various sizes. The third section lists synthetic graphs generated using the R-MAT graph model [10]. The fourth section lists regular scientific computing graphs. 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. For performance and scaling comparisons, we also use R-MAT (labeled RMAT) 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. Cluster-1 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 nodes. We used only the XE6 nodes. Each node has two eight-core 2.45 GHz AMD Opteron 6276 (Interlagos) CPUs and

TABLE I

Test graphs: # Vertices n, # Edges m, Vertex degrees (average d_{avg} and max d_{max}), and approximate diameter $\widetilde{\text{D}}$ are listed.

Graph	n	m	d_{avg}	d_{max}	$\widetilde{\text{D}}$
	($\times 10^6$)			($\times 10^3$)	
lj	5.4	69	14	23	16
orkut	3.1	117	38	33	9
friendster	66	1806	53	5.200	32
twitter	53	1963	38	780	19
wikilinks	26	601	23	39	830
dbpedia	67	258	4	7333	8
indochina	7.3	149	41	256	27
arabic	23	552	49	576	48
it	41	1151	29	1327	26
sk	51	1949	38	8563	308
uk-2002	1.8	298	16	195	30
uk-2005	39	781	40	1776	21
uk-2007	106	3302	31	975	25
wdc12-pay	39	623	16	4933	13
wdc12-host	89	2043	23	3391	19
rmat_22	4.2	67	16	121	7
rmat_24	17	268	16	389	9
rmat_26	67	1074	16	670	9
rmat_28	268	4295	16	1153	9
InternalMesh1	0.3	4	13	0.026	116
InternalMesh2	2.2	28	13	0.026	232
InternalMesh3	18	220	13	0.026	464
InternalMesh4	140	1819	13	0.026	631
nlpkkt160	8.3	112	13	0.027	142
nlpkkt200	16	216	13	0.027	203
nlpkkt240	28	373	13	0.027	243
Graphs used for Blue Waters strong scaling runs					
WDC12	3564	128 373	36	95 097	5200
RMAT	3564	128 290	36	-	-
RandER	3564	128 290	36	-	-
RandHD	3564	128 290	36	-	-
Graphs used for Blue Waters weak scaling runs					
RMAT	2^{25} to 2^{34}	2^{29} to 2^{40}	16/32/64		
RandER	2^{25} to 2^{34}	2^{29} to 2^{40}	16/32/64		
RandHD	2^{25} to 2^{34}	2^{29} to 2^{40}	16/32/64		

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

[2] http://commoncrawl.org

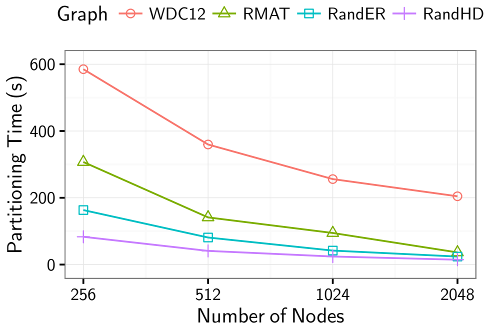

Fig. 1. XtraPuLP parallel performance (*strong scaling*) results on Blue Waters for computing 256 parts of various test graphs.

64 GB memory. Our experiments used up to 8192 nodes of Blue Waters, which is about 36% of the XE6 total capacity.

V. Results

We will demonstrate the performance of our new partitioner by assessing its scalability, partition quality, and impact on distributed graph analytics. We compare against ParMETIS version 4.0.3 [20] and PuLP version 0.1 [27]. 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 Cluster-1. 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. We fix load imbalance at 10% for both contraints – vertices and edges. Our choice to compare to ParMETIS is driven by the fact that it is the only openly available multi-constraint partitioner.

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 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.

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 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.

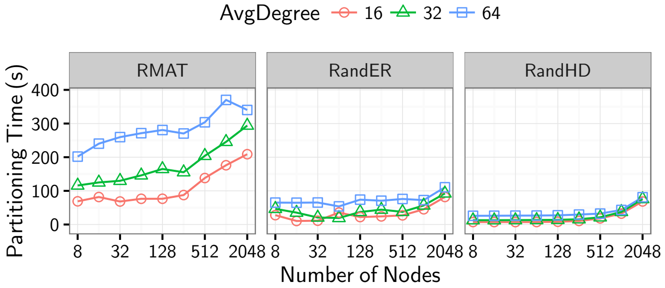

Fig. 2. XtraPuLP parallel performance (*weak scaling*) results on Blue Waters for various RMAT, RandER, and RandHD graphs. The number of graph vertices per node is $\approx 2^{22}$. The number of parts computed is set to number of nodes.

2) Trillion Edge Runs: We ran additional experiments on up to 8192 nodes, or 131072 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. It should be noted that parallel partitioning is partly a "chicken-and-egg" problem: in order to further improve weak scaling performance for, say, RMAT graphs, we would need to statically reduce inter-node data volumes exchanged, which is dependent on the initial vertex ordering. XtraPuLP strong and weak scaling results on Blue Waters demonstrate that there are no performance-crippling bottlenecks at scale in our implementation.

TABLE II

XTRAPULP, PULP, AND PARMETIS PARALLEL PERFORMANCE RESULTS ON CLUSTER-1 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.

	Partitioning Time (s)			**XTRAPULP Speedup**	
Graph	XTRAPULP (16 nodes)	PULP (1 node)	ParMETIS (16 nodes)	vs PULP	Rel. to 1 node
lj	**4.90**	10	59	2.0×	8.9×
orkut	**4.80**	18	110	3.8×	8.6×
friendster	**232**	1672		**7.2**×	11 ×
twitter	**1647**	3611		2.2×	2.3†×
wikilinks	**137**	467		3.4×	5.9×
dbpedia	**35**	70		2.0×	**14** ×
indochina	**4.40**	8.10	130	1.8×	11.3×
arabic	**12**	16	754	1.3×	8.2×
it	**22**	32		1.4×	9.4×
sk	**33**	67		2.1×	9.1×
uk-2002	**5.10**	9.20	85	1.8×	**12.8**×
uk-2005	**18**	34		1.9×	9.8×
uk-2007	**49**	71		1.4×	3.9†×
wdc12-pay	**241**	1062		4.4×	6.2×
wdc12-host	**422**	2443		**5.7**×	8.6×
rmat_22	**6.70**	14	126	2.1×	4.9×
rmat_24	**30**	147	923	4.9×	10.5×
rmat_26	**183**	1022		5.6×	**12.5**×
rmat_28	**981**	5454		**5.6**×	3.2‡×
InternalMesh1	**0.09**	0.14	0.63	1.6×	20.0×
InternalMesh2	**0.45**	0.89	0.71	2.0×	23.4×
InternalMesh3	2.90	6.80	**1.20**	2.3×	**27.9**×
InternalMesh4	24	46	**4.60**	1.9×	26.7×
nlpkkt160	1.60	3.80	**1.50**	2.4×	11.5×
nlpkkt200	2.60	6.40	**2.20**	**2.5**×	13.6×
nlpkkt240	4.60	11	**3.60**	2.4×	13.5×

3) Scaling on Cluster-1: We extensively test XTRAPULP at a smaller scale (16 nodes of Cluster-1), for direct performance comparisons to ParMETIS and PULP. XTRAPULP exploits hybrid MPI and thread-level parallelism, and we use a single MPI task per compute node. 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 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 II. Empty cells in the table indicate cases where ParMETIS failed to run to completion, and this was mostly 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. The single-node shared-memory PULP is consistently faster than distributed-memory parallel ParMETIS for the first three classes of graphs. This is expected, given that the label propagation achieves good shared-memory strong scaling and has been shown to work well as a community detection and partitioning approach for small-world graphs. For the fourth class of regular, high-diameter graphs, ParMETIS outperforms PULP and XTRAPULP; ParMETIS is optimized to partition these types of graphs.

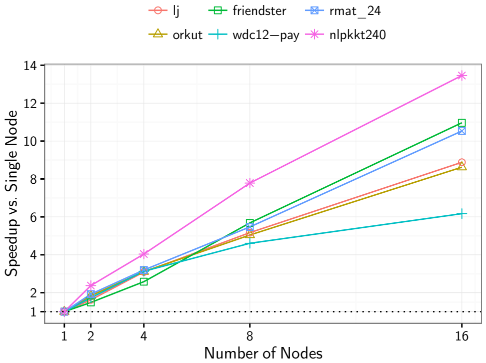

Fig. 3. XTRAPULP relative speedup results on Cluster-1 for computing 16 parts of various graphs.

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. The speedup relative to PULP is 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.

For large graphs such as wdc12-host, we achieve a significant reduction in running time by exploiting multiple compute nodes. We also note that the superlinear relative speedups for InternalMesh graphs are due to the fact that the initial partitioning of these graphs is actually quite good, thereby leading to a low communication-to-local computation ratio.

Figure 3 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

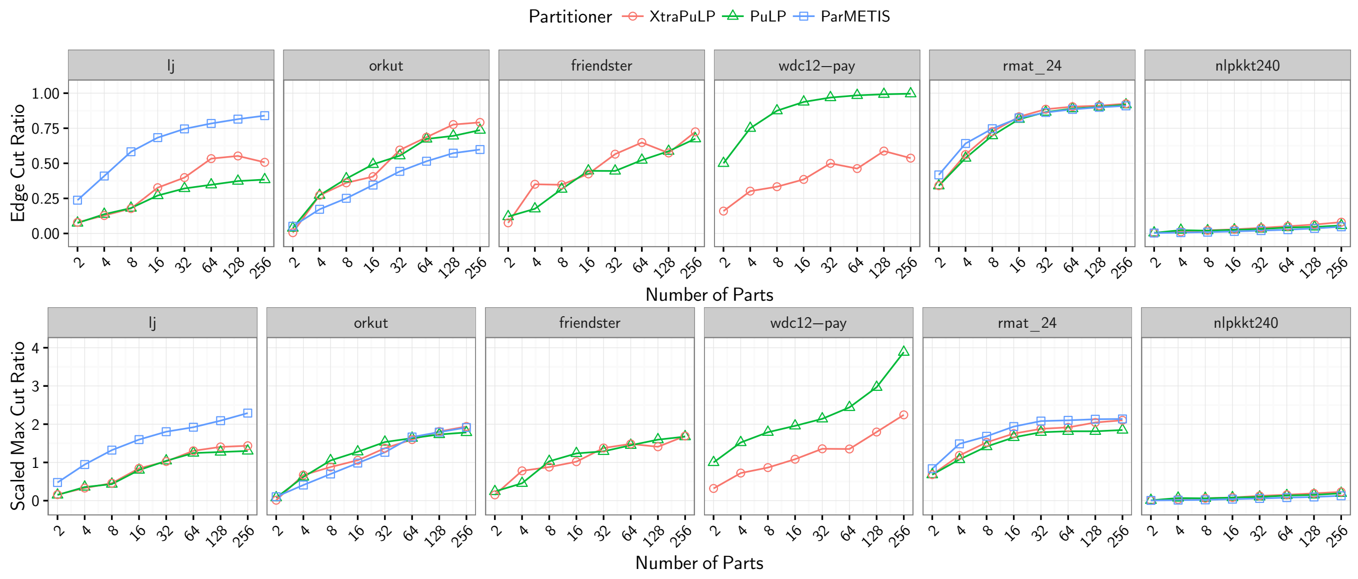

Fig. 4. Partition quality comparison when varying the number of parts computed. For both Edge Cut ratio and Scaled Max Cut ratio, lower values are better.

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 4, we report these metrics, varying the number of parts from 2 to 256 as quality results can vary with number of parts. We join the individual data points to indicate trends. We use the same six representative graphs that were used for strong scaling experiments on Cluster-1. Again note they 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 with increasing part count when partitioning rmat_24. An edge cut ratio value close to 1.0 means that nearly every edge is a cut edge. For graphs with intrinsically high edge cut ratios (such as the graphs in the first class, online social networks and communication networks), any quality gains must be assessed taking partitioning running times into consideration. Ideally, the partitioning method should finish quickly for 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. We also observe that XTRAPULP trends are not as "smooth" as PULP trends for some graphs (e.g., lj, friendster), which may be due to the distributed-memory parallelization of label propagation or the iteration counts for balancing and refinement phases. 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. 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. 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.

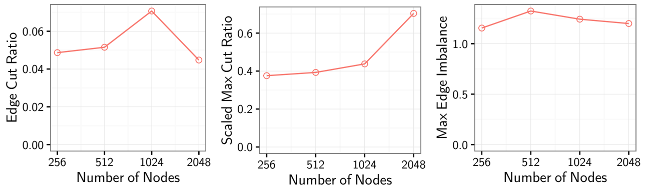

Fig. 5. Partitioning quality results for computing 256 parts of the WDC12 graph on Blue Waters.

Our final quality experiment on Blue Waters measures how partition quality varies with large-scale parallelism. In Figure 5, we plot how the edge cut ratio, scaled max per-part cut ratio, and partition edge count imbalance vary with large MPI task counts, when partitioning WDC12 into 256 parts. The edge cut ratio is 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. The resulting partitions from XtraPuLP are well-balanced. The increase in max cut ratio is possibly due to the fact that, as task count increases, the number of updates allowed per task and per iteration decreases due to the multiplier $mult$, introduced in the XtraPuLP algorithms section. In future work, we will experiment further with the (X, Y) parameters, the multiplier, and alternate communication schemes in order to ensure more consistent partitions when scaling the number of tasks.

C. Additional Comparisons

Here we provide additional comparisons to the recent state-of-the-art partitioner of Meyerhenke et al. [24], 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 with a 3% load imbalance constraint, and we compute 2-256 parts of the lj social network, scale 22 R-MAT graph, and uk-2002 web crawl. In Figure 6, we compare edge cut (top) and execution time (bottom).

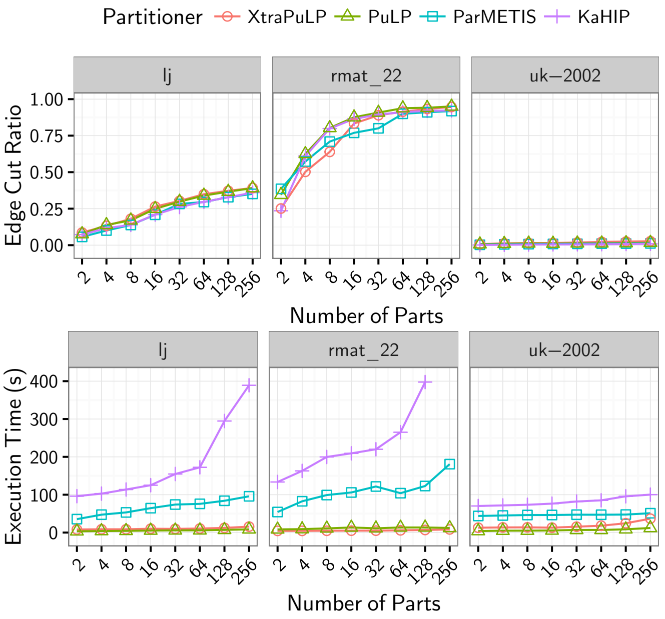

Fig. 6. Partitioning quality (top) and execution time (bottom) for multiple partitioners solving the single objective single constraint partitioning problem.

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 were 1.27 for PuLP, 1.73 for XtraPuLP, 11.81 for ParMETIS, and 26.5 for Meyerhenke et al. These results demonstrate the efficiency tradeoff between quality and time to solution, the choice of which to optimize for being 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

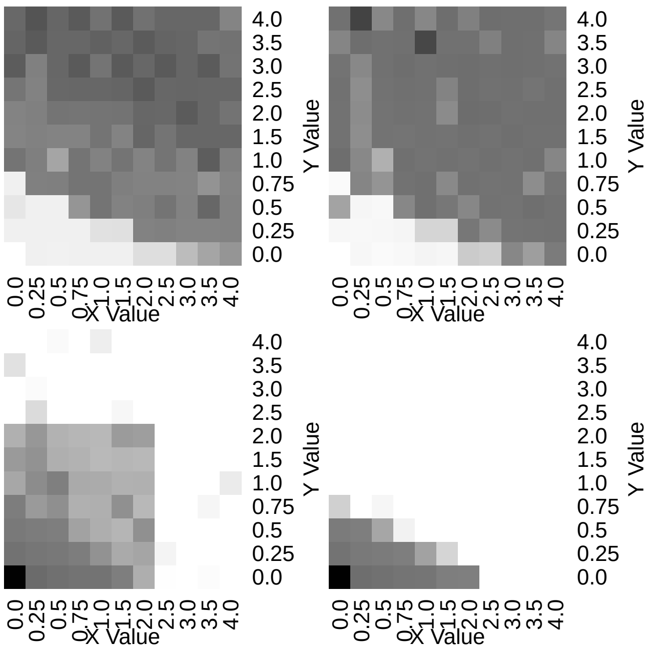

Fig. 7. Clockwise from upper left: Edge Cut versus X, Y; Max Cut versus X, Y; Vertex Balance versus X, Y; Edge Balance versus X, Y.

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 plot heatmaps of the average results in Figure 7, with white indicated higher quality or better balance and black indicating poorer quality or balance. For the two quality plots, we omit results that exceed the 10% balance constraint by at least 5%

or more. For the balance plots, solid white indicates that all tests conducted for that X, Y achieved the balance constraints.

The top two plots give the edge cut versus X, Y (left) and the max per-part cut versus X, Y (right). From these plots we observe two trends. Most obviously, a lower X and Y indicates a higher quality cut. This is because lower values for these parameters allow the highest number of part reassignments and therefore the greatest overall refinement. Second, we notice 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) can greatly refine the initial parts while limiting the potential imbalance possible on the final iterations.

The bottom two plots show overall average edge balance (left) and vertex balance (right). In general, the level of balance achieved is opposite the quality of cut. The optimal X, Y pair of values should therefore be selected along the *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.

E. Applications

We next demonstrate that XTRAPULP can significantly improve performance of real-world analytics. Consider analytics on the 128 billion edge WDC12. 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 8, 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 in each node such that each node has approximately the same number of edges. VertexBlock partitioning stores roughly the same number of vertices and all their adjacencies in each node. Random partitioning assigns vertices to nodes randomly. XTRAPULP assigns vertices based on the computed partition. The six analytics considered (algorithms presented in [29]) are Harmonic Centrality (HC) computation of 100 vertices, approximate K-core decomposition (KC), Label Propagation-based community detection (LP), PageRank (PR), extraction of the largest strongly connected component (SCC), and weakly connected components decomposition (WCC). For XTRAPULP, we *include* the partitioning time in comparisons. For XTRAPULP, we exploit prior knowledge [29] and run the balancing stage of XTRAPULP after first initializing with vertex block partitioning.

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, such as PR and LP, even when including the XTRAPULP partitioning time. We are unaware of other partitioning methods that are capable of processing graphs that are structurally similar to WDC12.

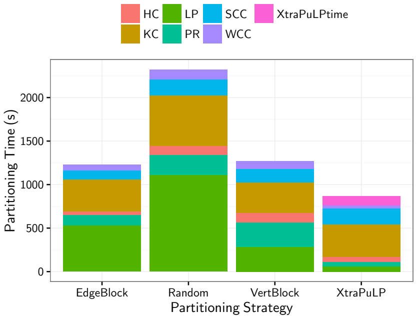

Fig. 8. The parallel performance results of various parallel graph analytics (HC, KC, LP, PR, SCC, WCC) on 256 nodes of Blue Waters, executed on the WDC12 graph with different graph partitioning strategies.

In Table III, we present results that show partitioning impact on sparse matrix vector multiplication (SpMV). We use the Epetra package of the Trilinos scientific computing library [17] to perform 100 SpMV operations, using matrices constructed from the six select test graphs previously shown. 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). 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 [6], 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 Cluster-1 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. Regular meshes such as nlpkkt240 do not directly benefit from a 2D distribution, and hence 1D-Rand partitioning fares poorly.

VI. CONCLUSION

We introduced XTRAPULP, our distributed-memory partitioner that 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

TABLE III
THE PERFORMANCE RESULTS OF PARALLEL SPARSE MATRIX VECTOR MULTIPLICATION (SpMV) ON 1 NODE (16 MPI TASKS) TO 16 NODES (256 MPI TASKS) OF CLUSTER-1, WITH DIFFERENT GRAPH PARTITIONING STRATEGIES. PM: ParMETIS. WE REPORT TIME FOR 100 SpMVS.

		Execution time for 100 SpMVs (s)								**Speedup**
Graph	MPI	**1D partitioning**				**2D partitioning**				2D XtraPuLP
	tasks	Block	Rand	PM	XtraPuLP	Block	Rand	PM	XtraPuLP	over 1D Rand
lj	16	19.87	12.24	10.90	9.81	15.74	10.09	9.26	**8.32**	1.47×
	128	6.31	4.29	3.41	3.05	4.04	1.92	2.07	**1.85**	2.32×
	256	4.73	2.96	2.65	2.41	2.32	1.26	1.28	**1.12**	2.64×
orkut	16	28.80	26.62	23.31	23.82	19.88	17.29	17.86	**16.51**	1.61×
	128	10.05	10.43	7.77	8.35	4.20	**3.07**	3.56	3.42	3.05×
	256	6.55	6.27	5.63	5.25	2.25	1.94	**1.82**	1.91	3.28×
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×
	128	44.20	50.49	27.69	27.28	25.57	25.31	15.93	**15.52**	3.25×
	256	32.22	37.37	19.63	19.76	14.67	15.93	**9.97**	10.02	3.73×
nlpkkt240	16	30.96	26.67	19.86	17.55	**16.08**	40.96	16.96	18.03	1.48×
	128	2.59	26.73	**2.14**	2.57	2.71	14.55	2.24	2.56	10.44×
	256	1.62	22.61	**1.04**	1.37	1.81	10.97	1.12	1.56	14.50×

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 Henning Meyerhenke, Peter Sanders, and Christian Schulz for providing the source code for their partitioner.

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*, 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)*. ACM Press, 2011.

[5] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. Int'l. World Wide Web Conf. (WWW)*. ACM Press, 2004, pp. 595–601.

[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. on High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2013, p. 50.

[7] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," *CoRR, abs/1311.3144*, 2015.

[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 Symposium (IPDPS)*. IEEE, 2014, pp. 425–434.

[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. ACM, 1999, pp. 251–262.

[15] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. Conf. on Design Automation*, 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 (TOMS)*, 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 *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. ACM, 2000, pp. 163–170.

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

[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] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," in *Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, 2015, pp. 1055–1064.

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

[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. M. Slota, K. Madduri, and S. Rajamanickam, “PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks,” in *Proc. IEEE Conference on Big Data (BigData 2014)*, 2014.

[28] ——, “Complex network partitioning using label propagation,” *SIAM Journal on Scientific Computing*, to appear.

[29] 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.

[30] “Stanford large network dataset collection,” http://snap.stanford.edu/data/index.html.

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

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

[33] 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)*. IEEE, 2014, pp. 568–579.

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

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

[36] A. Yoo, A. H. Baker, R. Pearce *et al.*, “A scalable eigensolver for large scale-free graphs using 2D graph partitioning,” in *Proc. Int’l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2011, p. 63.