A Scalable Distributed Graph Partitioner

Daniel Margo Harvard University
Cambridge, Massachusetts
dmargo@eecs.harvard.edu

Margo Seltzer Harvard University
Cambridge, Massachusetts
margo@eecs.harvard.edu

ABSTRACT

We present Scalable Host-tree Embeddings for Efficient Partitioning (Sheep), a distributed graph partitioning algorithm capable of handling graphs that far exceed main memory. Sheep produces high quality edge partitions an order of magnitude faster than both state of the art offline (e.g., METIS) and streaming partitioners (e.g., Fennel). Sheep's partitions are independent of the input graph distribution, which means that graph elements can be assigned to processing nodes arbitrarily without affecting the partition quality.

Sheep transforms the input graph into a strictly smaller elimination tree via a distributed map-reduce operation. By partitioning this tree, Sheep finds an upper-bounded communication volume partitioning of the original graph.

We describe the Sheep algorithm and analyze its space-time requirements, partition quality, and intuitive characteristics and limitations. We compare Sheep to contemporary partitioners and demonstrate that Sheep creates competitive partitions, scales to larger graphs, and has better runtime.

1. INTRODUCTION

Graph partitioning is an important problem that affects many graph-structured systems. For example, partitioning quality greatly impacts the performance [26] of distributed graph analysis frameworks such as Giraph [4] and PowerGraph [14]. PowerGraph even integrates a novel streaming partitioner into its data loader. These designs have received considerable attention and invited much comparison [34].

METIS [19] is the gold standard for graph partitioning and remains competitive after 15 years. Though METIS and related work "solve" the small graph partitioning problem, these approaches do not scale to today's large graphs. Distributed systems, such as PowerGraph, have emerged because there are now many graph-structured problems that exceed the main memory of a single machine, and METIS and similar approaches are unable to partition graphs of this scale. Additionally, there is growing interest in partitioning algorithms that minimize metrics other than edge-cut. For example, the minimum communication volume metric [8] has become attractive for the growing classes of graphs and analyses that do not partition well in edge-cut models.

Fundamentally, graph partitioning for distributed computing is a chicken and egg problem. We want to partition large graphs so we can process them at memory speed when they exceed the memory of a single machine. Distribution lets us handle larger graphs and parallelize computation, but it is only efficient when the partitions distribute the data well. Unfortunately, partitioning requires us to solve an NP-complete problem on an out-of-memory graph without an *a priori* well-partitioned data distribution! Streaming graph partitioners [35] and streaming graph analysis systems, such as Graphchi [22], are recent approaches to this problem.

But streaming graph systems pose two problems. First, they are sensitive to the stream order, which can affect performance (as in triangle counting [2]) or solution quality (as in PowerGraph or the more recent Fennel partitioner [36]). These results are unsurprising, because many graph ordering problems are NP-complete [5] and related to partitioning. The second problem is that streams cannot always take advantage of parallel scaling. Some streaming algorithms are difficult to parallelize (Fennel), while others support multiple streams (PowerGraph). However, if a multi-stream algorithm is sensitive to the cross-stream input partitioning, it becomes yet another partitioning chicken and egg problem.

We present Sheep, a distributed graph partitioner that embraces the relationship between ordering and partitioning. Given an order or ranking on an undirected graph's vertices, Sheep finds partitions by a method that does not vary with how the input graph is distributed among tasks. Thus, Sheep can arbitrarily divide the input graph to exploit parallelism and fit tasks in memory. Using simple degree ranking, Sheep creates competitive edge partitions an order of magnitude faster than both offline and streaming partitioners. As a result, Sheep scales well on large graphs.

Sheep is founded on a synthesis of insights between sparse matrix and complex network theories. Sheep reduces the input graph to a small elimination tree [30], a venerable structure that expresses vertex separators of the input graph. Sheep then solves a partitioning problem on this tree that translates to an upper-bounded communication volume partitioning on the original graph. This "reduce and partition" technique is similar to METIS, but the theory and details are quite different. In particular, Sheep's tree transformation is a distributed map-reduce operation. This distributed reduction is itself an interesting avenue for future research.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

The contributions of this work are:

- A new parallel and distributed partitioning algorithm that addresses the minimum communication volume partitioning problem on undirected graphs.
- A demonstration that Sheep scales well to graphs that exceed single machine memory, and is faster than competing algorithms without sacrificing partition quality.
- A distributed elimination tree construction that avoids construction of a chordal graph.
- A novel theory that relates partitioning, complex networks, and sparse matrix theories.

This paper is structured as follows. In the following Section we present background material and related work. In Section 3 we present the high-level Sheep algorithm and show how it creates partitions. In Section 4 we go into detail on the distributed tree reduction step, and give some theory and intuitions governing why Sheep works. We evaluate Sheep in comparison to other partitioners and against itself at various scales in Section 5. Section 6 addresses the limitations of Sheep and suggests ideas for future research.

2. BACKGROUND AND RELATED WORK

There are four areas of research on which this work builds: graph partitioning algorithms, graph analytic systems that are impacted by partitioning, sparse matrix theory, and complex network analysis. We address the first three topics below, but we defer complex networks to Section 4.3.

2.1 Graph Partitioning

METIS [19], which has been a reliable standard for graph partitioning, is a *multi-level* graph partitioner that creates a sequence of "coarsened" graphs where each vertex represents a union of vertices in the previous graph. It computes partitions on the smallest graph and then projects them back to each larger graph in succession, refining the partitions as it does so. METIS can optimize edge cuts or communication volumes, but the solutions discovered in either case are sometimes similar [18]. Multi-level methods are frequently used for graphs with tens or hundreds of millions of elements. However, they consume memory and are an order of magnitude slower than streaming methods, so multi-level partitioners are challenged by the billion-element graphs becoming common today. ParMETIS [20] is a distributed version of METIS, but it suffers from a partitioning chicken and egg problem: each distributed task works on a subgraph and needs to communicate with other tasks in proportion to the edges between subgraphs. So the performance of this method is harmed without some *a priori* partitioning.

The streaming partitioning model [35] was created to address partitioning problems on large scale graphs. In this model each graph element arrives in sequence and must be immediately assigned to a partition. Streaming forbids partition refinement or any global introspection such as spectral analysis; it is also well suited for integration with the data loaders of graph analysis engines. Fennel [36] is a representative work in this area that interpolates between two established heuristics [31] [35]. Bourse et al. [8] extend Fennel's method to communication volume partitioning. However, all streaming partitioners are sensitive to the stream order and random orders are pessimal approximations [36]. Thus, it is difficult to quantify the quality of a streaming partitioner; but in general streaming partitioners produce worse partitions than METIS, though they operate more quickly.

Sheep outperforms both METIS and Fennel in runtime, is competitive with METIS in quality for a small number of partitions (i.e., less than 5), and is competitive with Fennel for larger numbers of partitions.

2.2 Partitioning in Graph Analytic Engines

PowerGraph [14] is representative of graph analysis frameworks that use stream partitioning to break a large graph into pieces small enough to run on individual nodes. It uses an edge placement partitioning model, assigning edges to machines and duplicating vertices on multiple nodes as necessary. PowerGraph integrates a novel multi-streaming partitioner into its data loader to minimize duplicates. This design has received much attention but is well known to have a problem with severe partition imbalances [8]. Pregel [25] and Giraph [4] are frameworks that partition vertices instead of edges, whereas GPS [33] and PowerLyra [9] are recent hybrid systems: they partition the edge sets of high-degree vertices but keep the edge sets of low-degree vertices together. Sheep is most effective for edge partitions, and it intuitively produces partitions that exploit this same property as GPS and PowerLyra, although it does so indirectly by a different method; we discuss this further in Section 4.4

GraphChi [22] is a single-machine graph analysis system that handles out-of-memory graphs by creating partitions that it processes as parallel streaming working sets. While in principle this system could benefit from well partitioned sets, out of memory partitioning is traditionally addressed by adding more memory; so Graphchi, which targets low-memory systems, does not feature any partitioner at all. X-Stream [32] is a similar streaming analysis system.

2.3 Sparse Matrix Theory

Sheep partitions a graph by partitioning a data structure called an *elimination tree* [30], as described in the following Section. Elimination trees are a famous data structure, but Sheep constructs the tree using a novel distributed method. Nested dissection [12] is an alternate method to construct elimination trees in parallel. It works by finding small vertex separators and then recursing into the remaining components of the graph. Because vertex separators are a form of partitioning (Section 3.4), this is a chicken and egg problem for partitioning applications. However, there is a sense in which Sheep "reverses" nested dissection by deriving partitions from an elimination tree constructed by other means. Ashcraft and Liu explored a similar idea [3] with an algorithm that extracts separators from an elimination tree and then sorts the separators to find a better tree, although they optimize for different parameters than does Sheep.

3. OVERVIEW

3.1 The Sheep Algorithm

Given an undirected graph, G, we partition it by:

1. Sorting the vertices,
2. Reducing the the graph to an elimination tree [30], T, according to the order in step 1,
3. Partitioning the elimination tree, and then
4. Translating the tree partitions into graph partitions.

Elimination trees are defined in detail in the rest of this Section, as is the partitioning method for T and the translation of partitions from T to G. The vertex sort and tree reduction are discussed in detail in Section 4.

3.2 Conventions

$G = (V, E)$ is an *undirected graph* where V is a set and E is a subset of $V \times V$. G_V and G_E distinguish the elements of different graphs. By convention, $n = |V|$ and $m = |E|$.

$P = (V, \leq)$ is a *partial order* on a set V given by the binary relation $\leq$ over V. For clarity we may refer to $\leq_P$ and $<_P$. We say P is *total* iff $x \leq y$ or $y \leq x$ for all $x, y \in V$. In this paper, the symbol P is generally a total order.

Because $\leq$ is a binary relation over V, any partial order defines a graph (V, E'), although unlike G, this graph is a *directed acyclic graph* (DAG). By convention if $x \leq y$ then $(x, y) \in E'$ where x is the source and y is the target. Conversely, the transitive closure (reachability graph) of any DAG is a partial order, so we can say that a DAG "defines a partial order". In particular, an elimination tree is a DAG and defines a partial order (Section 3.3). This order is not a total order unless the elimination tree is a linear path graph.

3.3 Elimination Trees

Let $G = (V, G_E)$ be an undirected graph and $T = (V, T_E)$ be a *directed rooted* tree or forest with the same vertex set: that is, for any connected component $T_C \in T$ there is a unique $r \in T_C$ that is reachable from all $x \in T_C$. This implies that edges point from leaves towards the root. Then, T is an *elimination tree* of G iff T holds the following property:

PROPERTY 1 (ELIMINATION PROPERTY). *For each $(x, y) \in G_E$, either x is reachable from y in T or vice-versa. Equivalently, x and y share an ancestor-descendant relation in T, and T defines a partial order where $x < y$ or $y < x$.*

If G is a complete graph (clique) then T must be a linear path graph. T is usually more interesting than a line, but in general T is not a balanced tree. If G is connected then T must be a single tree, but for graphs with k connected components, T can be a forest with up to k trees. An elimination tree is deeply related to the components and connectivity of its associated graph via the following corollaries:

COROLLARY 1.1. *Let x and y be children of z in T. Let $subt(x)$ be the set of vertices in the subtree rooted at x in T. Then, there does not exist any edge $(x', y') \in G_E$ between $subt(x)$ and $subt(y)$. This is a well known result [15].*

COROLLARY 1.2. *Let x and y be children of z in T. Let $supr(z)$ be the set of z and reachable vertices from z in T. If we delete $supr(z)$ from G, then in the resulting graph $subt(x)$ is not reachable from $subt(y)$ and vice-versa.*

Corollary 1.2 is true because any path from $subt(x)$ to $subt(y)$ must contain a vertex in $supr(z)$. We say $supr(z)$ is a *vertex separator* of $subt(x)$ and $subt(y)$, because it is a set of vertices whose removal *separates* sets of graph elements. Note that $supr(z)$ is not necessarily *minimal*, because it may contain vertices whose removal is unnecessary to separate $subt(x)$ and $subt(y)$. However, $supr(z)$ is bounded above by the *tree-depth* [6] of the elimination tree. Figure 1 depicts an example tree; we discuss tree construction in Section 4.

3.4 Partitioning

An *edge k-partitioning* of G is an assignment $E \to \{1, ...k\}$ to k partition sets such that each edge is assigned to one partition and no partition is larger than $(1 + b)(m/k)$, where b is called the *balance factor*. Similarly, a *vertex k-partitioning* is an assignment $V \to \{1, ...k\}$ where no partition is larger than $(1+b)(n/k)$. Partition optimizations usually minimize the balance factor and another constraint called the *cost*.

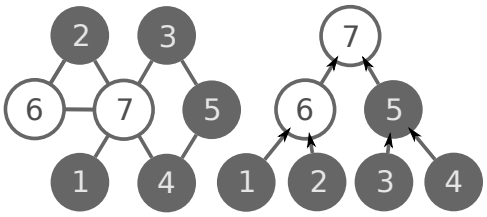

Figure 1: Graph G and elimination tree T. For $(x, y) \in G_E$, x is reachable from y or vice-versa. Note that $supr(6) = \{6, 7\}$ is a vertex separator of 1 and 2, and in addition 7 is a separator of $subt(5) = \{3, 4, 5\}$.

Sheep is a *communication volume* [8] minimizing partitioner, like PowerGraph and some versions of Fennel. Communication volume is a partition cost that counts the unique partitions touching each vertex, minus one for normalization so that the cost of a 1-partitioned graph is 0.

$$\sum_{x \in V} |\{partition(element) : element \in adjacent(x)\}| - 1$$

Note that for edge partitions *element* is an edge, but for vertex partitions *element* is a vertex and includes x itself (e.g. the cost of a two vertex line graph with each vertex in a different partition is 2). In general, edge partitionings achieve lower volumes because there are more edges than vertices and therefore more degrees of freedom to assign partitions.

Informally, communication volume counts the number of "duplicate" vertices in some graph-structured systems. For example, PowerGraph partitions edges across machines and then instantiates adjacent vertices on each machine. Each vertex has one duplicate instance for each partition adjacent to it, minus one primary instance. Of course, which instance is primary does not affect the count. If we say the primary instance rests with the highest-ordered partition in some arbitrary order, then we can equivalently express communication volume by a summation over the partitions.

Let K be the set of partitions. Then, an equivalent expression of communication volume is:

$$\sum_{i \in K} |\{x : x \in V, x \in adjacent(i) \cap adjacent(j), i < j \in K\}|$$

That is, each vertex x adjacent to partition i is a duplicate iff x is also adjacent to a higher-ordered partition j; else, the primary instance of x rests with i. For vertex partitions $adjacent(i)$ is inclusive of all $x \in i$. This duplicate set is trivially a vertex separator of $adjacent(i)$ from all $adjacent(j)$ in the subgraph induced on G by the union of $adjacent(i)$ and all $adjacent(j)$. Therefore, we can model a partitioning as a separator series: each partition in arbitrary order separates its primary vertices from the graph of "remaining" unclaimed vertices. The duplicate vertices are the separators, and the communication volume is the sum of the separators.

An elimination tree also expresses a series of separators that are upper bound by its tree-depth. Recall that by Corollary 1.2, if x and y are children of z in T, then $supr(z)$ is a vertex separator of $subt(x)$ and $subt(y)$ in G. Furthermore, by recursion $supr(z)$ is a separator of $subt(x)$ and $subt(y')$ for all y' such that y' is a child of $z', z' \in supr(z)$.

In effect, $supr(z)$ is a vertex separator of $subt(x)$ from the "remaining" vertices not in $subt(x)$ (see Figure 1.

Using these properties we establish a map between the elements of T and G such that a vertex partitioning of T maps to an edge or vertex partitioning of G, with an upper bound on the communication volume given by the tree-depth of T.

3.5 Translating Partitionings

First, we model a cut-minimizing vertex partitioning problem on T that will translate to a bounded communication edge partitioning problem on G. Let the partition of each edge $(x, z') \in G_E$ be the partition of $x \in T_V$, where $x \in subt(z')$ by the Elimination Property and x is a child of z in T. It follows that $z' \in supr(z)$, and that the adjacencies of (x, z') and the adjacencies of all (y', z') where $y' \notin subt(x)$ intersect in $supr(z)$, or else $supr(z)$ is not a vertex separator of x and y' and contradicts Corollary 1.2. More generally, the adjacencies of every edge mapped in $subt(x)$ and every edge *not* mapped in $subt(x)$ must intersect in $supr(z)$.

Therefore, let the weight of $x \in T_V$ be $|\{(x, z') \in G_E : x \in subt(z')\}|$, and let the cut cost of $(x, z) \in T_E$ be $|supr(z)|$. These costs decrease from leaf to root. It turns out that weighted tree partitioning is trivial for decreasing edge costs: there is a simple leaf to root dynamic program [21] that, given a decreasing edge-costed tree and a maximum subtree weight, finds a minimum cost partitioning of T into subtrees less than that weight. Each subtree maps to an edge partition in G, and its cut cost $|supr(z)|$ gives an upper bound on the intersection of the adjacencies of that partition and every partition that follows it in T. The sum of these cuts is an upper bound on the communication volume in G.

If the maximum subtree size is $(1+b)(m/k)$ this will sometimes produce $k' \geq k$ partitions. To achieve exactly k partitions it may be necessary to bin-pack subtrees into partitions: this is a common feature of balanced tree partitioning algorithms. However, we can always achieve exactly k partitions by relaxing the balance factor. Since bin packing has a constant approximation factor, we know that the amount we may relax the balance is similarly bounded; in practice we achieve partitions with less than 3% imbalance. Packing can only decrease the communication volume, so this does not affect the correctness of the upper bound, which is at worst $O(k' \times depth(T))$ because $depth(T) \geq |supr(z)|$.

For a vertex partitioning of G, let the partition of each $x \in G_V$ be the partition of $x \in T_V$. However, in this case the cut cost of each edge $(x, z) \in T_E$ must be $|subt(x)| + |supr(z)|$, because the adjacencies in G of vertices $x' \in subt(x)$ and $z' \in supr(z)$ may intersect in $subt(x)$ as well as $supr(z)$. Intuitively, in an edge partitioning we have the freedom to map each edge to the "lower" vertex, and this lets us restrict the partition intersections to $supr(z)$, the set above the lower vertex. However, a vertex partitioning constrains edges to map to both endpoints. We could also construct an edge partitioning that assigns each edge to the higher vertex with a cut cost of $|subt(x)|$, but these costs do not decrease from leaf to root. If the costs do not decrease then we must partition T by some other algorithm. However, since T is small compared to G we can use a powerful method like METIS even for large G. For example, the UK Web dataset [7] is 44.8GB as a graph file, but only 841MB as a tree file.

Sheep produces better edge partitioning results because its bounds are tighter. This reflects the greater degree of freedom in edge partitioning problems, particularly the freedom to diffusely partition the edge sets of higher vertices. However, for both edge and vertex partitionings the cut cost is upper bound by the tree-depth of T, and so in both cases Sheep's partitions improve with shallow trees. Tree-depth minimization is also NP-complete [6] but this aspiration opens up some novel ideas. In particular, elimination tree construction is usually modeled as an ordering problem, so this lets us reason about partitioning in terms of a vertex order or ranking that would give an ideal tree. This leads to a novel observation regarding elimination trees and complex network theories, which we discuss in Section 4.3.

4. THE TREE CONSTRUCTION

We present our tree construction in three parts. First, we review the elimination game, a classic elimination algorithm (Section 4.1). We then present our own elimination algorithm and prove that we can distribute it across arbitrary partitions of the graph(Section 4.2). Finally, we discuss how vertex orders affect the trees we and how we derive good orders from complex networks theory (Section 4.3). Finally, we give some intuition for these results (Section 4.4).

4.1 Elimination Games

The *elimination game* is a classic algorithm [29] that takes an undirected graph and produces an elimination tree.

Require: G is an undirected graph (V, G_E)
Require: P is a total order $(V, \leq)$
  function ELIMINATION GAME(G, P)
    $H \leftarrow G$
    $T \leftarrow (V, \emptyset)$
    for all $x \in V$ in order P **do**
      $S \leftarrow \{y : (x, y) \in H_E, x <_P y\}$
      $H_E \leftarrow H_E \cup \{(y, z) : y, z \in S\}$
      $T_E \leftarrow T_E \cup (x, \min_P(y \in S))$

For each vertex x, we add edges to the graph between all y such that y is a P-greater neighbor of x. The parent of $x \in T$ is the P-minimum over all such neighbors. The Elimination Property that $(x, y) \in G_E$ share an ancestor descendant relationship in T holds because in iteration x either T_E gains (x, y) or some (x, x'), in which case H_E gains (x', y). Then, in iteration x' either T_E gains (x', y), or we continue until T_E gains some (x'', y). Then, $x, x'...x'', y$ is a path from x to y in T. Note that T is not a subtree of G, because T_E gains (x, y) from H_E and H is a supergraph.

H is called a *chordal graph* or elimination graph. Chordal graphs are important generalizations of trees with many interesting properties [15], some of which are inherited by the elimination tree T, which is called a *host tree* of H. In particular, chordal graphs have $O(n)$ minimal separators. However, H is an unbounded supergraph of G and in practice is often expensive to construct. P is called an *elimination order* of G and a "perfect" elimination order of H.

Our distributed tree construction algorithm requires a special observation about the elimination game. The proof is simple but is delegated to the appendix for brevity:

THEOREM 1. *Let $G[V <_P z]$ be the subgraph induced on G by vertices less than z. Then, z is the parent in T of exactly the P-maximum vertices in the disjoint components of $G[V <_P z]$ that z joins together in $G[V \leq_P z]$.*

This gives an elegant characterization of the trees constructed by the elimination game as products of a union-find algorithm. Because union-find algorithms are easy to distribute, this leads to a distributed tree construction.

4.2 Distributed Reduction

Let $U = (V, P)$ be a union-find data structure over a set V that chooses as each subset's representative the maximum element in that subset according to a total order $P = (V, \leq)$.

Require: G is an undirected graph (V, G_E)
Require: P is a total order $(V, \leq)$
  function PERSISTENT UNION FIND(G, P)
    $U \leftarrow (V, P)$
    $T \leftarrow (V, \emptyset)$
    for all $z \in V$ in order P **do**
      for all $(x, z) \in G_E, x <_p z$ **do**
        $y \leftarrow U.find(x)$
        if $y \neq z$ **then**
          $U.union(y, z)$
          $T_E \leftarrow T_E \cup (y, z)$
    return T

In each outer iteration vertex z adopts the P-maximum vertex y in each disjoint component of $G[V <_P z]$ that z joins in $G[V \leq_P z]$. By Theorem 1, these are the same children of z as in the elimination game, so T is an elimination tree. Afterwards z is the new P-maximum representative for this set of components, which are now one and joined through z. We call this a "persistent" union-find because T captures the development of the union-find data structure.

This algorithm uses less space and time than an algorithm that creates a chordal supergraph. Supergraphs are at best $o(n+m)$ but usually much worse; conversely, the union-find U and tree T use $O(n)$ space and $O(n+a(n)m)$ time, where $a()$ is the near-constant inverse Ackermann function [11]. However, many elimination algorithms dynamically order themselves by inspecting the chordal graph, so this method is not clearly practical without a good order *a priori*.

The observation that this union-find algorithm can be efficiently distributed is at the core of Sheep. We prove that:

THEOREM 2. *Let G_1 and G_2 be two subgraphs of G such that $G_1 \cup G_2 = G$. Let $t(G, P)$ be the elimination tree produced by union-find on G in order P. Then,*

$$t(t(G_1, P) \cup t(G_2, P), P) = t(G, P)$$

Note that though $t(G_1, P) \cup t(G_2, P)$ is a directed graph, it is interpreted as undirected when input to the union-find algorithm. This proof is given in the appendix.

We emphasize this creates the same exact tree as $t(G, P)$. By this theorem we can split G into any number of subgraphs, construct trees independently for each, and then union and reduce the intermediate trees in parallel to create a final tree for G. The result is insensitive to how the graph is split and the space-time requirements remain nearly linear for each subgraph. This lets us reduce large graphs quickly. An implementation using a map-reduce might look like:

  function REDUCE TO TREE(G, P)
    $G_1, G_2 ... G_w \leftarrow Split(G)$
    $T_1, T_2 ... T_w \leftarrow \{Mapper(G', P) : G' \in G_1, G_2 ... G_w\}$
    $T \leftarrow Reducer()$ over $T_1, T_2 ... T_w$ and fixed P
  function MAPPER(G', P)
    return $PersistentUnionFind(G', P)$
  function REDUCER(T_L, T_R, P)
    $U \leftarrow Undirected(T_L \cup T_R)$
    return $PersistentUnionFind(U, P)$

Let $G_1, G_2 ... G_w$ be a set of subgraphs of G such that $G_1 \cup G_2 \cup ... G_w = G$. Let n' and m' be the maximum numbers of vertices and edges respectively in any such G_w. Then, the parallel runtime of this distributed tree construction is:

$$O(n' + a(n')m' + log(w)(n + a(n)n))$$

where the log expression is the reduce operation over intermediate trees. Because the log expression is essentially $O(n)$ for fixed w, and because m typically dominates n, it is more important to balance $m' = (m/w)$ than n'. This is achieved by evenly splitting an edge list of G_E. If we also want to balance n' we may do so by randomizing the list. In either case, this avoids the partitioning chicken and egg problem.

4.3 Ordering Vertices

Trees created by elimination algorithms are the result of a graph G in an order P. So far we have held P constant, but in real applications the graph is constant and the order can vary. Thus, the tree and the vertex separators it expresses are entirely determined by P, hence our earlier statement that Sheep embraces the relationship between ordering and partitioning: Sheep's partitions result from the separators expressed by an elimination order. Formally, if S is a minimal separator of components C_1 and C_2 in G, then any P such that $\forall x \in C_1 \cup C_2, \forall y \in S, x <_P y$ will express S in T.

Order sensitivity is a challenge faced by all streaming graph partitioners. Tsourakakis et al. proved that not only must every streaming partitioner have adversarial orders, but also that random input orders are approximately adversarial [36]. Therefore, for arbitrary input orders one cannot make guarantees as to the quality of streaming partitioning results. Sheep is subject to this argument if P is arbitrary.

However, Sheep accommodates its order in its underlying theory, so we can define what is meant by a good order. As discussed in Section 3, tree-depth upper bounds the separators expressed by the elimination tree; therefore, a minimum tree-depth order will produce smaller bounds and better communication volumes. Unsurprisingly, tree-depth minimization is NP-complete. There are many depth heuristics in the literature, but in general these inspect the graph or chordal graph. We need a compatible heuristic for our distributed construction, since it cannot easily inspect the total graph and does not create a chordal graph.

We found a valuable resource in the complex networks community. Albert et al. pioneered the empirical use of attack plots that, for a given vertex order on the x-axis, plot the size of the largest remaining connected component when one deletes vertices in that *attack order* [1]. The purpose of these plots is to show how different networks dissassemble under different attack orders, and to find orders that fully dissassemble networks in a minimum number of attacks.

Consider a walk on T from root to leaves. At each vertex z in the walk, the subtrees of the children of $z \in T$ represent components in the "remaining graph" $G[V <_p z]$: it is like we delete $supr(z)$ and examine the remaining components. The subtree with the most depth is the component that requires the most steps to fully disassemble. So, disassembling the graph in a minimum number of attacks is exactly the same goal as minimizing its elimination tree-depth. Attack orders and elimination orders are simply opposite orders.

Complex network research repeatedly observes [1][17] that many natural networks, and in particular, networks with skew degree distributions, are vulnerable to degree-ordered attacks. More sophisticated attack orders use centrality measures such as betweenness [17]. It is not surprising that this works: degree may be characterized as a "greedy edge attack" and betweenness as a "greedy shortest path attack."

Elimination algorithms sometimes use a similar degree heuristic [13] , although this is usually applied online to the chordal graph in order to optimize related parameters called tree-width and matrix fill-in for matrix factorization. Tree-width is a strictly tighter separator bound than tree-depth [15], but because the chordal graph is a supergraph of the input graph these parameters do not lead to graph size reductions. For matrix factorization this is not a concern because factorization requires instantiation of the chordal graph (or "fill-in graph"). This research area arises from the application of graph theory to graphical models of matrices; Sheep reapplies some of these theories to the partitioning problems faced by large scale graph analysis frameworks.

Thus, it is well known that degree orders are sometimes "good" for elimination trees. However, so far as we know it is a novel observation that complex networks research gives empirical characterizations of both the classes of graphs on which degree orders are tree-depth minimizing, and the orders that outperform degree orders on these graphs. By default Sheep assumes a degree elimination order for input graphs; even in distributed graphs this is easy to compute by broadcasting local degree vectors. Note that Sheep *need not* physically sort the graph in degree order; it merely uses the order logically in its tree construction algorithm.

Our results show that degree orders on skew networks produce low cost partitions that are competitive with other partitioners; this method works extremely well for bipartitioning and often outperforms METIS. We also show that Sheep is improved when better rankings are available, e.g., across repeated analytics runs. This ability to improve the graph's data organization with purely analytic results may be an interesting technique for graph database cracking [16].

4.4 Intuition

Degree sorting is also a classic heuristic optimization for triangle counting algorithms, because it improves reference locality in networks with skew degree distributions. The many low-degree vertices tend to reference the few high-degree vertices, so clustering the high-degree vertices gives a block-structured cache efficient partitioning. However, this heuristic is topologically naive. In particular, many low-degree vertices are not clustered with their adjacencies. For example, if a 2-degree vertex is adjacent to another 2-degree vertex, they may not be clustered in the sort order even though this constraint may be easily fulfilled.

Intuitively, Sheep exploits a partially ordered tree that is more informative than its linear input order. Unlike a total order, the tree expresses *antichains*: sets of independent elements in the underlying graph. Independence is notably present in sparse topologies such as the low-degree vertices of a skew network. Shallow trees generally exhibit more antichains, and a tighter bound on the set of vertices that other vertices may reference. Sheep clusters related elements better than a sorting heuristic, because it does *not* cluster *unrelated* elements in common cases where a sorting heuristic would. In very dense graphs, Sheep devolves to a sort.

One consequence of the above is that for edge partitions, Sheep usually divides the edges of high-degree vertices and keeps the edges of low-degree vertices together. This is because Sheep maps $(x, y) \in G_E$ to the lower vertex $x \in subt(y)$ in T, which is also the lower-degree vertex if P is a degree order. Therefore, the edges of the high degree vertices are spread across T, whereas the edges of the low degree vertices are concentrated in the peripheral and leaves of T. Due to the clustering property described above the peripherals are usually independent and well-localized. This result may be intuitively compared to the high-degree vertex partitioning methods used by GPS [33] and PowerLyra [9].

5. EVALUATION

We evaluate Sheep to demonstrate the following claims:

- Sheep scales in the following ways: parallel processing on one machine (Section 5.2), out-of-memory processing on one machine (Section 5.2), and parallel processing in a distributed environment (Section 5.3).
- Sheep is faster than other partitioners on large graphs (Section 5.4).
- Sheep's partitions are competitive and are improved by better vertex orders (Section 5.5).

We compare Sheep to results published in KDD'14 [8], which evaluated METIS, PowerGraph, and both vertex and edge streaming Fennel on the edge balanced minimium communication volume partitioning problem. Fennel [36] is a good representative for streaming partitioners, because it is a simple per-vertex or per-edge loop that considers each partition for each element and chooses the partition that minimizes a special cost function. This design is typical of streaming partitioners [31] [35]. We contacted the authors to ensure that our results can be accurately compared. Fennel is called "IC" in Bourse's results, but we have confirmed this is a modification of Fennel to optimize communication volumes instead of edge cuts.

In addition to these graphs we added a few others to cover interesting cases. The Twitter graph and UK Web graph are popular billion-edge networks for graph systems evaluations, and the High Energy Physics coauthorship network is a well known complex networks dataset. Most of these graphs are available through the Stanford Large Network Dataset Collection [23]. Table 1 summarizes these graphs.

5.1 Implementation and Setup

We implemented Sheep in C++ using LLAMA [24], an open-source graph storage library. LLAMA is based on the venerable compressed sparse row (CSR) representation, but allows mutability, and, for read-only algorithms like Sheep,

name	$n = \|V\|$	$m = \|E\|$	file size	reason
HEphysics [28]	7,610	15,751	189KB	fig. 12
com-youtube	1,135k	2,988k	36MB	bourse
cit-patents	3,775k	16,519k	198MB	bourse
com-liveJ	3,998k	34,681k	416MB	bourse
soc-liveJ	4,848k	68,735k	828MB	fig. 2
com-orkut	3,072k	117m	1.4GB	bourse
twitter_rv	42m	1,468m	17.6GB	scale
uk_2007_05 [7]	106m	3,739m	44.9GB	scale

Table 1: Graph datasets used in this evaluation.

it adds little overhead relative to conventional CSR implementations. It is not distributed, so for distributed tasks we simply open LLAMA subgraphs in different processes. Sheep uses the MPI map-reduce library both for distributed sorting and for the tree reduction operation described in Section 4.2. However, Sheep can fall back on a parallel filesystem for inter-process communication if MPI is not available.

Sheep uses CSR for most variable-length data structures to reduce allocator overhead. However, a tree can be serialized as an array of parent pointers, so this representation is preferred for inter-process communication. As with many iterative graph algorithms, the inner loop of Sheep is performance sensitive, so it was important to optimize our union-find implementation and to use vertex isomorphisms between data structures for fast comparison in the order P.

Typically algorithms that process an out-of-memory graph in order should sort the graph first. However, for Sheep it is efficient to divide the graph into in-memory working sets and then process each subgraph as-is. Because each intermediate tree is a partial suborder of the total order P, and because these trees are merged and reduced in order P, there is a sense in which Sheep is "its own external merge sort."

For single machine experiments, we use a 6-core Intel i7-970 at 3.20GHz with 12GB of RAM and a Samsung 840 Pro SSD. For distributed processing, we use a cluster of Dell PowerEdge M915 servers; each has 64 AMD Abu Dhabi cores at 2.30GHZ with 256GB of RAMand 41.25GBps Infiniband. The local disk is not measured because all our cluster benchmarks are hot cached. Graph input files are binary 96-bit edge lists, as in the Graph500 benchmark [27], but none of the graphs in this study exceed four billion (32-bit) vertices. LLAMA itself is a 64-bit CSR system. In distributed experiments we first copy the graph to local storage on each node, but we do not include the copy time because it is not a feature of the algorithms and may vary greatly between data center architectures. We do however include graph file ingest times, because distributed ingest is an important feature of distributed algorithms and is measured in Graph500.

Bourse et al. did not evaluate runtime, so we measured timings for several competing partitioners. There is no public Fennel implementation, but it is a simple algorithm, so we implemented the versions in Bourse's study. We implemented both edge streaming Fennel and vertex streaming Fennel with an initial CSR ingest. Despite the ingest time, Fennel's vertex streaming is an order of magnitude faster than its edge streaming, because the critical partitioning work is $O(kn + m)$ rather than $O(n + km)$, where k is the partition count. Because Bourse gives quality results for both versions of Fennel, we evaluated vertex Fennel to privilege Fennel's timing results. Vertex-streaming Fennel would be even faster with a pre-sorted vertex adjacency list, but so would Sheep, and we want to use the Graph500 standard.

We ran METIS with default settings. By default METIS is an edge cut partitioner, but it accepts a communication volume minimizing goal with some time overhead. Whether this actually improves the volume varies with the graph [18]. We timed METIS without this option because it slows down METIS and does not always improve its quality, and our quality results come from Bourse. METIS requires that adjacency lists have dense vertex IDs (i.e., $n = max(id)$), so we privileged METIS further by providing it this format.

The PowerGraph partitioner is deeply integrated with the PowerGraph bootstrap process, so it would not be fair to take timing results directly from PowerGraph. We do, however, include quality results for PowerGraph from Bourse.

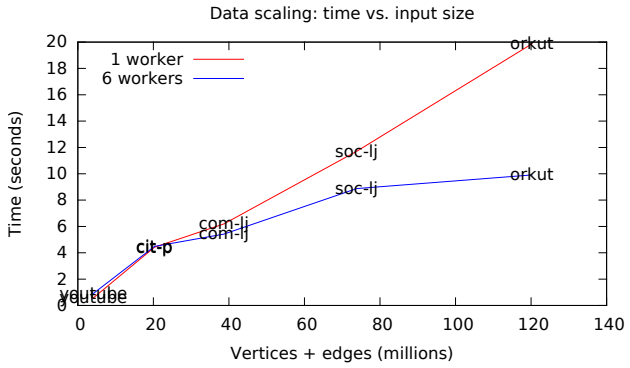

Figure 2: Sheep runtime as a function of graph scale, expressed as the sum of the number of vertices and edges. We show results for both 1 and 6 workers on a variety of graphs, run on our commodity machine.

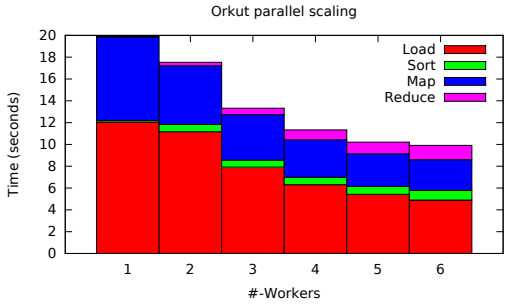

Figure 3: Parallel time scaling for Sheep on the Orkut data set and our commodity machine. Tree partitioning takes less than 200ms and is omitted.

5.2 Single-machine Scaling

Figure 2 plots the runtime of Sheep on our commodity machine for a variety of input graphs for one and six parallel workers. As expected, single-worker Sheep's runtime is linear in the size of the graph and additional workers speed it up. The speed up is relatively poor for small graphs, but it improves with larger graphs. We will see this trend in the cluster setting as well. This is a common pattern in parallel algorithms, but the cause here is especially interesting.

Figure 3 shows a detailed breakdown of runtime versus the number of workers for Orkut, the largest of the graphs shown in Figure 2. The load time represents the time to ingest into the in-memory CSR, the sort time measures computing the global degree order, the "map" time measures construction of the intermediate tree from the initial subgraphs, and the reduce time measures the distributed reduce that combines the intermediate trees into a final tree. The actual tree partitioning step takes less than 200ms and is not distributed, so we ellide it for visibility. We observe that, while the sort and reduce costs are not insignificant, scaling is limited because the load and reduce times do not scale linearly in the number of workers. This is surprising, because each worker processes m/k edges, and the algorithm is near-linear.

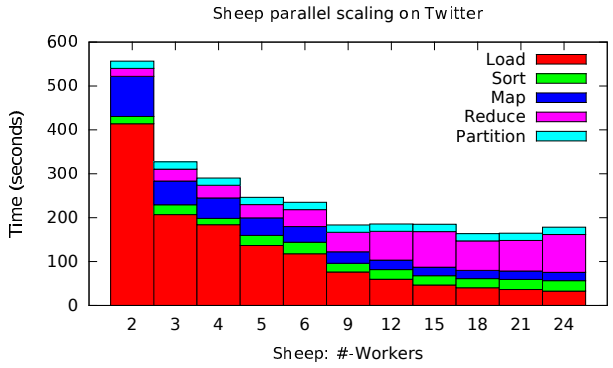

Figure 4: Parallel time scaling for Sheep on the Twitter data set on a single cluster node.

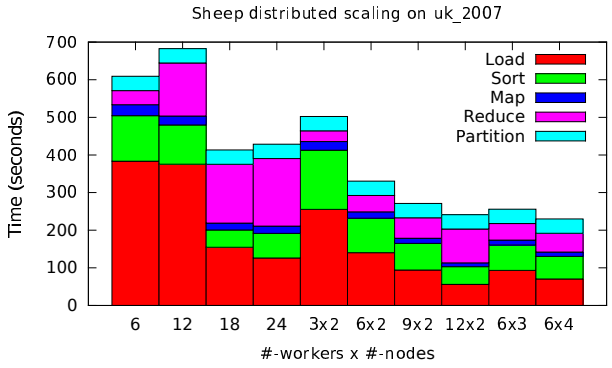

Figure 5: Time versus the workers and machines for Sheep on the UK Web dataset and multiple cluster nodes. 6x4 signifies 6 cores on 4 nodes, i.e. 24 cores. Sheep is data bound, so adding extra memory controllers improves its runtime at scale.

The cause of this limitation is imbalance in the underlying graph structure. If we divide a graph edge list into w random parts, then in expectation, each part contains m/w edges but some n' vertices, where n' is a function of the degree distribution and is generally greater than n/w. This is not an implementation detail but rather a fundamental property of distributed graph algorithms that divide the graph into subgraphs. Additionally, there is no guarantee that the input edge list is randomly distributed, and in fact, processes that produce edge lists typically exhibit locality. Therefore, there is some skew in the number of vertices represented in each subgraph, although this effect is less significant. This is not an instance of the partitioning chicken and egg problem – random hashing solves this. However, because this problem is specific to certain graphs and diminishes with scale, it is generally not worth randomizing the input.

When graphs exceed the memory of a single machine, Sheep scales by dividing the graph into memory-sized working sets. For example, the undirected Twitter graph is approximately 23GB in CSR, and therefore 1.9x the memory of our commodity machine. If we break the graph into 10 parts we can run 2 tree constructions simultaneously in memory and partition Twitter in just 7.5 minutes. Compare this result to our in-memory Orkut results. The Twitter graph is approximately 12.5 times the size of the Orkut graph, and 7.5 minutes is approximately 25x the runtime of Orkut with 2 workers, producing a factor of 2x overhead. Since Twitter is out of memory and Orkut is hot cached, this overhead is entirely expected and seems reasonable.

Twitter fits in memory on our 256GB cluster machines. Figure 4 plots the runtime of Sheep on Twitter versus the number of parallel workers on a single cluster machine. Twitter is large enough that we see serious parallel scaling in both our load and map times, such that the distributed reduction becomes the limiting factor, as predicted by the complexity equation in Section 4.2. In fact, for graphs as large as Twitter our ingest scales better than $1/k$, because CSR ingest requires a partial edge sort. Using 18 cores we load and partition this graph in just 2.8 minutes. In comparison, Fennel takes over 20 minutes, and METIS takes hours.

5.3 Distributed Scaling

Of course, for out of core graphs we are inevitably interested in distributed scaling. Figure 5 plots time versus increasing workers and nodes for Sheep on the 3.7 billion edge UK Web dataset [7]. While our single node time is quite reasonable (less than 8 minutes), adding more cores does not produce linear scaling. The ingest step does not reliably improve, and the reduction step is surprisingly expensive. The underlying cause of this is that we are badly thrashing the various caches of the 256GB NUMA node; the reduce, in particular, involves many inter-process buffer copies.

As Sheep is distributed, it can scale to more machines to relieve memory pressure. The addition of just one machine improves the runtime as we move from three cores per machine to twelve. The x-axis labels of CxM signify C cores on M machines. 3x2 cores is significantly faster than 6x1 cores because, unsurprisingly, the data ingest is faster. With 12x2 cores we get almost twice the performance of 24x1 cores, not only because ingest is faster, but also because we relieve the memory burden of buffer copying in the reduce step. Note also that 6x4 cores introduces no overhead over 12x2, so in our data center it is practical to simply ignore single-machine scaling and launch Sheep horizontally across machines.

Sheep, like any distributed algorithm, has counter-scaling costs that eventually cause its performance to approach an asymptote. The eventual indivisibility of the graph and the growing cost of the reduce tree must eventually prevent Sheep from leveraging additional resources. It is also clear that optimal performance requires some parameter tweaking with respect to the graph size and features of the data center architecture. However, even with suboptimal parameters Sheep is extremely fast relative to other partitioners.

5.4 Comparative Time

Next we compare Sheep to other partitioning algorithms. Figure 6 returns to Orkut on our commodity machine and plots runtimes for the various partitioning algorithms as a function of different numbers of partitions. First, note that Sheep is invariant to the number of partitions; not only is our tree partitioning step invariant to the partition count, but the partitioning is a trivial fraction of the runtime. Other partitioners evaluate multiple partitions for each element, so their times scale with the number of partitions. Nevertheless, at the scale of the Orkut graph, Fennel is a reasonable competitor for Sheep, while METIS is comparatively slow.

We now reexamine Figure 4, which plots time for Sheep

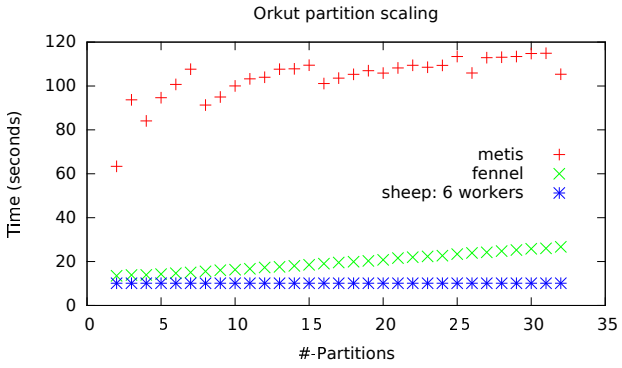

Figure 6: Runtime versus # of partitions on Orkut. Sheep is insensitive to the partition count, because Sheep spends less than 200ms partitioning.

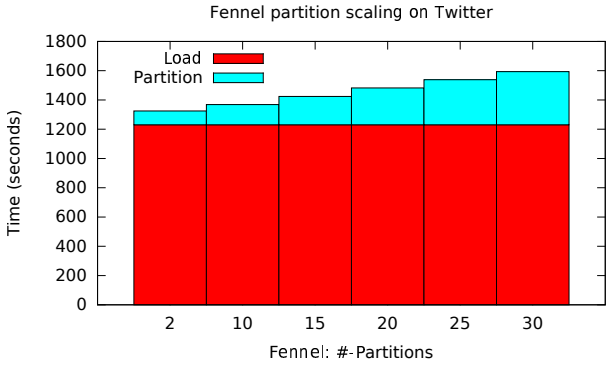

Figure 7: Time versus the partition count for Fennel on the Twitter dataset on a single cluster node. Note that the y-axis is 3x taller than Figure 4.

on Twitter versus the number of parallel workers on a single cluster machine. Compare to Figure 7, which plots time for Fennel on Twitter versus the partition count on a single cluster machine. Sheep is invariant to the partition count, and Fennel is a serial streaming algorithm. We measure our implementation of Fennel, but Tsourakakis et al. report a similar Twitter time of 40 minutes [36]. We were unable to get METIS to partition in-memory on our 256GB machines, but Tsourakakis used 1TB machines and reports 8.5 hours.

We observe that Sheep is many times faster than Fennel, primarily, but by no means entirely, because of rapid data ingest. But even if we exclude the ingest, Sheep is 2.4x faster than Fennel at 30 partitions, because Sheep is insensitive to the partition count. This ingest-free comparison is quite unfair to Sheep because rapid ingest is a major advantage of a distributed partitioner. And note that our Fennel implementation's loader supports some multithreading, so the advantage is not purely due to parallelism. Fundamentally it is cheaper to build several small CSRs than one large one.

Figure 8 plots the runtimes of various 2-partitioning algorithms at different input scales; this is the least favorable comparison for Sheep. For smaller graphs Sheep and Fennel run in essentially the same time, suggesting that both algorithms are data-bound. We see the same result when 2-partitioning Twitter if we discount ingest times, but at no

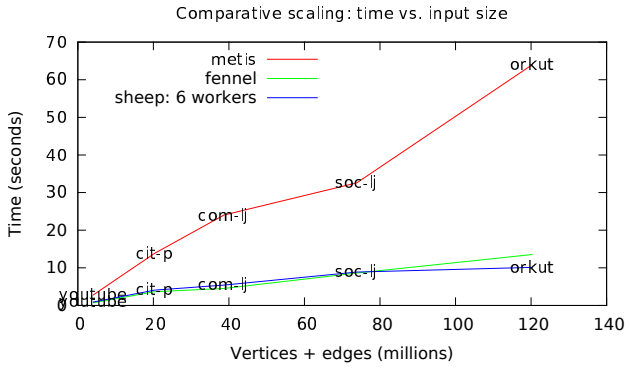

Figure 8: Partitioner runtime as a function of graph scale for 2-partitioning with various algorithms.

point does Fennel significantly outperform Sheep. However, Fennel is harmed by growing partition counts, and on larger graphs, Sheep benefits from rapid parallel ingest.

On small graphs, where METIS is a viable option, the partition quality of METIS and similar multi-level techniques are on average much better than other methods. Since METIS is almost two decades old, we think that small-graph partitioning is simply not an interesting problem at this time. We created Sheep to target larger graphs, and on these graphs Sheep is much faster than other partitioners.

5.5 Partition Quality

Figures 9 and 10 add Sheep results to partition quality results from Bourse et al. The y-axes are edge partitioned communication volumes as defined in Section 3.4. In the Bourse results the y-axes are normalized by m, but this compresses the curves and makes it difficult to compare partitioners within each plot. We do not reproduce balance factors from Bourse, because Sheep and every other partitioner except PowerGraph achieve balance factors less than 3%, which is the default in METIS. PowerGraph's balance is unbounded, and Bourse reports factors as high as 50% on Orkut and over 200% on a Youtube dataset.

Sheep is clearly competitive with other partitioners. It is slightly better than even METIS on bipartitioning problems, competitive with METIS up to 5 partitions, significantly better than other streaming partitioners up to 10, and slightly worse than some partitioners at more than 20. Taken together with our timing results, this shows that Sheep achieves its goal of producing competitive partitions with significantly improved runtimes and scalability.

Fennel and Powergraph are both sensitive to the vertex input order, and the ideal orders for the two are not necessarily the same. Bourse et al. address this issue by using a randomized input order. However, this penalizes Fennel, as Tsourakakis et al. showed that a random order is not ideal for Fennel [36]. To provide a more meaningful comparison, we include results from our own Fennel implementation in the graph's "natural" vertex ID order. While this may not be an optimal input order for Fennel (the optimal order is unknown), natural vertex orders are often correlated with some stochastic process such as a walk, and Tsourakakis argues that this often makes them appropriate for streaming. We tested Fennel with degree and reverse degree orders, but these gave worse results. Our Fennel implementation per-

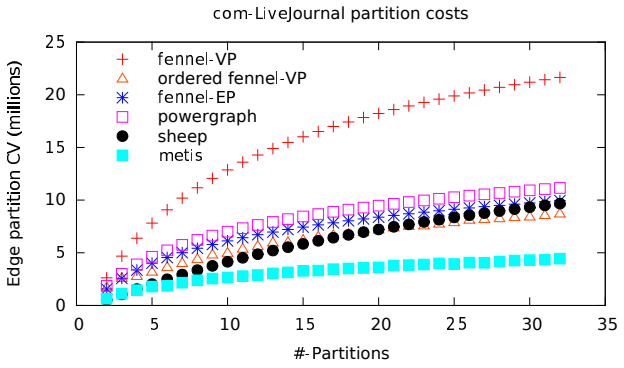

Figure 9: Edge-partitioned communication volumes versus # partitions on com-LiveJournal. ECV may be thought of as a count of duplicate vertices.

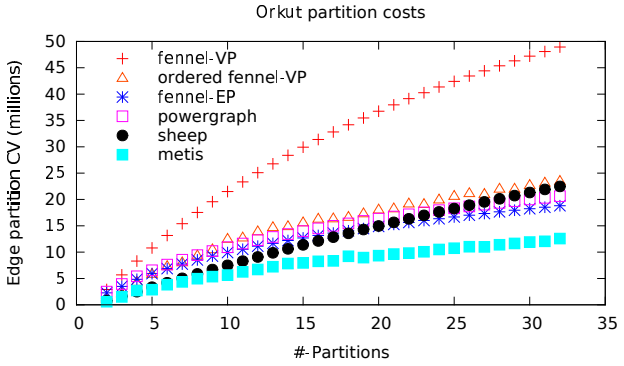

Figure 10: Edge-partitioned communication volumes versus # partitions on Orkut.

forms similarly to implementations reported elsewhere. It produces vertex partitions that are then transformed to edge partitions by Bourse's degree weight and random assignment method [8]. It may be possible to derive better partitions from a well ordered edge partitioned Fennel, but this is an order of magnitude slower than our vertex implementation.

Figure 11 plots Sheep against our Fennel implementation on Twitter; these results are not from Bourse. In this case the natural order of Twitter is unkind to Fennel, as Sheep outperforms it significantly. Nevertheless, this plot is similar to the Bourse plots if we assume that the natural order of Twitter is "random" for Fennel, and that our implementation should behave like Bourse's randomized fennel-VP. A well defined ideal vertex order is one of Sheep's advantages.

Sheep's partition quality improves using lower tree-depth elimination orders. Our observations regarding complex networks predict "sequential" betweenness centrality should improve over degree (Section 4.3). Figure 12 shows how Sheep improves when using sequential betweenness order [17]. The input graph is a small complex networks dataset observed as disassembling faster under betweenness attack than degree attack. As predicted, the tree-depth improves from 754 in degree order to 459 using sequential betweenness. The partition quality improvement is significant and roughly proportional to both the depth and the attack quality improvement reported by Iyer. Our results are nearly identical to METIS.

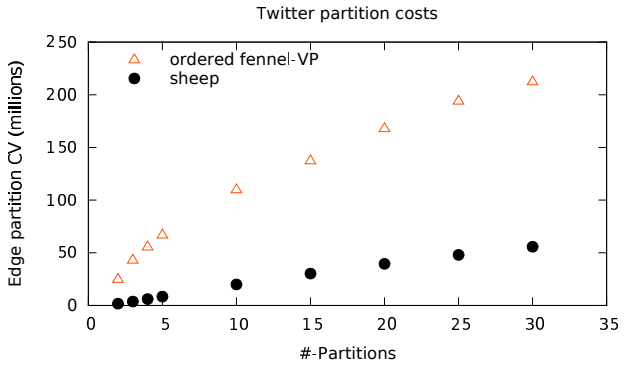

Figure 11: Edge-partitioned communication volumes versus # partitions on Twitter.

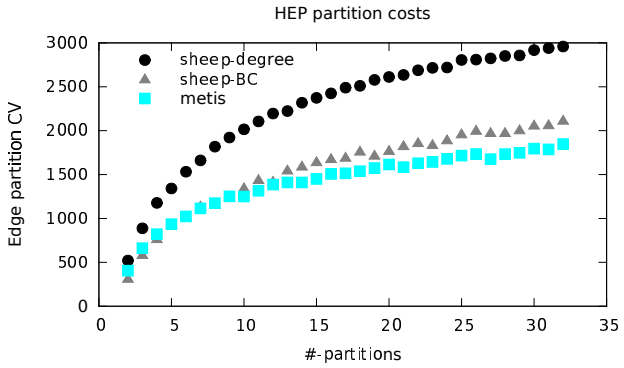

Figure 12: Edge-partitioned communication volumes versus # partitions on High Energy Physics.

Unfortunately, betweenness centrality is expensive to compute, so this exact method is impractical. However, there are methods to approximate and parallelize betweenness, and there are other efficiently-obtained centralities such as the k-core decomposition [10]. Conversely, it may be possible to derive useful centralities "in reverse" from elimination trees produced by e.g., METIS. While the importance of degree order is widely recognized in elimination trees, as far as we know the observation that non-local *analytic* centralities may reduce tree-depth is novel. This is an interesting line of research that we expect to address in future work.

6. CONCLUSION

Sheep is a graph partitioning algorithm that is many times faster than competing algorithms on large graphs without sacrificing partition quality. Sheep scales to effectively partition multi-billion edge graphs in as little as 4 minutes. Sheep's partition quality is comparable to, or even better than, METIS for small partition counts and competitive with streaming partitioners for larger partition counts. However, we see Sheep's most exciting contribution as the relationship it establishes between partition quality, tree decomposition theories, and analytic centralities. This is a rich space for innovative theories and system designs.

Sheep is free and open source software and is available at `https://github.com/dmargo/sheep`.

6.1 Limitations and Future Research

Sheep is an undirected communication volume partitioner, because its underlying theory works with vertex separators. There is a similar body of theory for edge-cut tree decompositions, called carving decompositions, so it may be possible to derive an edge-cut Sheep algorithm. The edge cut costs of our algorithm are unbounded and, in practice, may be quite bad, even though the communication volume results are consistently good. For edge-partitioned systems such as PowerGraph, an "edge cut" is not meaningfully defined, so this is not obviously a problem. However, the fact that high cut and low volume partitionings exist at all is somewhat interesting and a worthy topic for future research.

Sheep requires a good order to produce good partitions, but cannot easily create an order by introspection of the graph, because it splits the graph. Sheep works best for natural graphs with a skew degree distribution, such as "power law" graphs, because complex networks research shows that degree orders attack these graphs. Skew graphs are common in contemporary large graph analysis. For graphs with other distributions, such as finite element meshes, it may be possible to find an ordering heuristic by reviewing complex networks research, but we reserve this for future work. However, many of these graphs already have a rich history of successful partitioning methods such as planar bisection, whereas interest in skew networks is more recent.

Sheep creates a small partitioned tree that identifies the partition assignment of any graph element. This design is different from streaming partitioners, which place elements as they arrive, and affects best practices to integrate Sheep with existing systems. For example, an analysis system might ingest the graph, construct the tree, share it among nodes, and then direct each node's ingested data to the appropriate partition. If a placement step is necessary it should be trivially parallel, since the elements can be placed independently in consultation with the partitioned tree.

7. ACKNOWLEDGEMENTS

Thanks to Charalampos Tsourakakis, Florian Bourse, and Milan Vojnovic for helping us fairly compare to the results in their respective papers. Thanks to Peter Macko, both for LLAMA and for implementing several novel features for us.

8. REFERENCES

[1] R. Albert, H. Jeong, and A.-L. Barabási. Error and attack tolerance of complex networks. *Nature*, 406(6794):378–382, 2000.

[2] S. Arifuzzaman, M. Khan, and M. Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *ACM International Conference on Information and Knowledge Management*, 2013.

[3] C. Ashcraft and J. W. Liu. Robust ordering of sparse matrices using multisection. *SIAM Journal on Matrix Analysis and Applications*, 19(3):816–832, 1998.

[4] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Hadoop Summit*, 2011.

[5] H. L. Bodlaender, F. V. Fomin, A. M. Koster, D. Kratsch, and D. M. Thilikos. A note on exact algorithms for vertex ordering problems on graphs. *Theory of Computing Systems*, 50(3):420–432, 2012.

[6] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18(2):238–255, 1995.

[7] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[8] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *20th ACM International Conference on Knowledge Discovery and Data mining*, pages 1456–1465. ACM, 2014.

[9] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *10th ACM SIGOPS European Conference on Computer Systems*. ACM, 2015.

[10] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. K-core organization of complex networks. *Physical review letters*, 96(4):040601, 2006.

[11] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *21st ACM Symposium on Theory of Computing*, pages 345–354. ACM, 1989.

[12] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[13] A. George and J. W. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Operating Systems Design and Implementation*, volume 12, page 2, 2012.

[15] P. Heggernes. Minimal triangulations of graphs: A survey. *Discrete Mathematics*, 306(3):297–317, 2006.

[16] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *Conference on Innovative Data systems Research*, volume 3, pages 1–8, 2007.

[17] S. Iyer, T. Killingback, B. Sundaram, and Z. Wang. Attack robustness and centrality of complex networks. *PloS ONE*, 8(4):e59613, 2013.

[18] G. Karypis. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Minneapolis, MN*, 2013.

[19] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[20] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, Jan. 1998.

[21] S. Kundu and J. Misra. A linear tree partitioning algorithm. *SIAM Journal on Computing*, 6(1):151–154, 1977.

[22] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Operating Systems Design and Implementation*, volume 12, pages 31–46, 2012.

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

[24] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. Llama: Efficient graph analytics using large

multiversioned arrays. In *International Conference on Data Engineering*. IEEE, 2015.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010.

[26] H. Miao, X. Liu, B. Huang, and L. Getoor. A hypergraph-partitioned vertex programming approach for large-scale consensus optimization. In *International Conference on Big Data*, pages 563–568. IEEE, 2013.

[27] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[28] M. E. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*, 98(2):404–409, 2001.

[29] S. Parter. The use of linear graphs in gauss elimination. *SIAM Review*, 3(2):119–130, 1961.

[30] A. Pothen and S. Toledo. Elimination structures in scientific computing. *Handbook on Data Structures and Applications*, pages 59–1, 2004.

[31] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan. Managing large graphs on multi-cores with graph awareness. In *USENIX Annual Technical Conference*, pages 41–52, 2012.

[32] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *24th ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[33] S. Salihoglu and J. Widom. Gps: A graph processing system. In *25th International Conference on Scientific and Statistical Database Management*. ACM, 2013.

[34] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *ACM SIGMOD International Conference on Management of Data*, pages 979–990. ACM, 2014.

[35] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *18th ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, pages 1222–1230. ACM, 2012.

[36] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *7th ACM International Conference on Web Search and Data Mining*, pages 333–342. ACM, 2014.

APPENDIX

In Section 4.1 we claimed this about the elimination game:

THEOREM 1. *Let $G[V <_P z]$ be the subgraph induced on G by vertices less than z. Then, z is the parent in T of exactly the P-maximum vertices in the disjoint components of $G[V <_P z]$ that z joins together in $G[V \leq_P z]$.*

PROOF. *Lemma 1.* For all $(x, y) \in T_E$, $x <_T y$ in the partial order defined by T. Because $(x, y) \in T_E$ iff $x <_P y$, it follows that T defines a suborder of P. Therefore, for all $x \in T_V$, x must be the P-maximum vertex in $subt(x)$.

Lemma 2. Let $G[V <_P z]$ be the subgraph induced on G by vertices less than z. By Corollary 1.1, $subt(x)$ and $subt(y)$ are disconnected in $G[V <_P z]$. But, there must exist an edge (x', z) in $G[V \leq_P z]$ such that x' in $subt(x)$:

If x is a child of z, then in iteration x of the elimination game, $(x, z) \in H_E$. If $(x, z) \in H_E$, either $(x, z) \in G_E$ or there exists a prior iteration x' where $(x', z) \in H_E$. Again, either $(x', z) \in G_E$ or there exists a prior iteration. Because initially $H = G$ this must terminate in an edge in G_E. There must also exist (y', z) such that y' in $subt(y)$.

Lemma 3. If we apply Lemma 2 to $G[V <_P x]$ it follows that for each child of x there must be an edge in $G[V \leq_P x]$ that connects x to that child's subtree. The same is recursively true of the children of x's children, etc. It follows inductively that $subt(x)$ must be a connected component in $G[V <_P z]$. The same is true of $subt(y)$. By Lemma 2, $subt(x)$ and $subt(y)$ are disjoint components in $G[V <_P z]$, but z connects these components in $G[V \leq_P z]$.

Therefore, by Lemmas 1 and 3 vertex z is the parent of exactly the P-maximum vertices in the disjoint components of $G[V <_P z]$ that z joins in $G[V \leq_P z]$. □

In Section 4.2 we made the following claim:

THEOREM 2. *Let G_1 and G_2 be two subgraphs of G such that $G_1 \cup G_2 = G$. Let $t(G, P)$ be the elimination tree produced by union-find on G in order P. Then,*

$$t(t(G_1, P) \cup t(G_2, P), P) = t(G, P)$$

PROOF. Let G_1 and G_2 be subgraphs of G such that $G_1 \cup G_2 = G$. For clarity, let $t(G) = t(G, P)$ for constant P. Let $G' = t(G_1) \cup t(G_2)$. We must show that $t(G') = t(G)$.

By induction we will show that in iteration z of the union-find $t(G')$ gains exactly the same children of z as would $t(G)$. Clearly this is true in the first iteration where no children are gained: assume this is true of every iteration before z.

In iteration z let the *predecessors of z in $t(G)$* be all $(x, z) \in G_E, x <_p z$. The predecessors of z in $t(G')$ are all $(x, z) \in G'_E, x <_p z$. Since $G' = t(G_1) \cup t(G_2)$, and $t(G_1)$ and $t(G_2)$ both define suborders of P, the predecessors of z in $t(G')$ are exactly the children of z in $t(G_1)$ and $t(G_2)$.

The predecessors of z in $t(G_1)$ are all $(x, z) \in G_E, x <_p z$, and similarly G_2. But since G_{1E} and G_{2E} are subsets of G_E and $G_{1E} \cup G_{2E} = G_E$, every predecessor of z in $t(G)$ must be a predecessor of z in $t(G_1)$ or $t(G_2)$. So, every predecessor in G is seen by the algorithm in either $t(G_1)$ or $t(G_2)$. By the definition of the algorithm, for every such predecessor the maximum vertex y in its component in $G_1[V <_P z]$ or $G_2[V <_P z]$ is a child of Z in $t(G_1)$ or $t(G_2)$. In either case, y is then a predecessor of z in $t(G')$.

Therefore, for any predecessor (x, z) of z in $t(G)$, there exists a corresponding predecessor (y, z) in $t(G')$ such that y is a vertex in a component that contains x in either $G_1[V <_P z]$ or $G_2[V <_P z]$. Since both are subgraphs of $G[V <_P z]$, it must be the case that y is contained in the same component as x in $G[V <_P z]$. Therefore, by inductive assumption x and y are contained in the same set of the union-find U at iteration z of $t(G')$, and therefore $U.find(x) = U.find(y)$. Therefore, $t(G')$ finds the same children as $t(G)$, *though the predecessors by which it finds them may differ.* □