CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics

Loc Hoang, Roshan Dathathri, Gurbinder Gill, Keshav Pingali
Department of Computer Science
The University of Texas at Austin
{loc, roshan, gill, pingali}@cs.utexas.edu

***Abstract*—Graph analytics systems must analyze graphs with billions of vertices and edges which require several terabytes of storage. Distributed-memory clusters are often used for analyzing such large graphs since the main memory of a single machine is usually restricted to a few hundreds of gigabytes. This requires partitioning the graph among the machines in the cluster. Existing graph analytics systems usually come with a built-in partitioner that incorporates a particular partitioning policy, but the best partitioning policy is dependent on the algorithm, input graph, and platform. Therefore, built-in partitioners are not sufficiently flexible. Stand-alone graph partitioners are available, but they too implement only a small number of partitioning policies.**

This paper presents CuSP, a fast streaming edge partitioning framework which permits users to specify the desired partitioning policy at a high level of abstraction and generates high-quality graph partitions fast. For example, it can partition wdc12, the largest publicly available web-crawl graph, with 4 billion vertices and 129 billion edges, in under 2 minutes for clusters with 128 machines. Our experiments show that it can produce quality partitions 6× faster on average than the state-of-the-art stand-alone partitioner in the literature while supporting a wider range of partitioning policies.

***Index Terms*—Graph analytics, graph partitioning, streaming partitioners, distributed-memory computing.**

I. Introduction

In-memory analysis of very large graphs with billions of vertices and edges is usually performed on distributed-memory clusters because they have the required compute power and memory [1], [2], [3], [4], [5], [6]. This requires partitioning the graphs between the hosts of the cluster.

Graph partitioners can be evaluated along three dimensions.

- *Generality:* Does the partitioner support a variety of partitioning policies or is it restricted to one or a small number of baked-in policies? This is important because there is no one best partitioning policy that is optimal for all algorithms, input graphs, and platforms [7].
- *Speed:* How fast can it partition graphs?
- *Quality:* How good are the generated partitions for the algorithms, inputs, and platforms of interest? For example, even if we restrict attention to edge-cuts, there are many edge-cut partitions for a given graph, and some will be better than others for a given algorithm or platform.

This research was supported by NSF grants 1337217, 1337281, 1406355, 1618425, 1725322 and DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. This work used XSEDE grant ACI-1548562 through allocation TG-CIE170005. We used the Stampede system at Texas Advanced Computing Center, University of Texas at Austin.

Most existing graph analytics systems come with their own partitioning routines that implement a single partitioning policy and are tightly integrated with the system, but this is not flexible enough [2], [3], [4], [5], [6]. Stand-alone graph partitioners, which partition graphs offline for later use in an application, are available [8], [9], [10], but they are restricted to a small number of partitioning strategies; for example, Metis [8], XtraPulp [9], and Spinner [10] are restricted to edge-cuts. Additionally, even if a graph analytics system supports multiple policies such as D-Galois[1], partitioning time has been shown to take almost as much or even longer than execution of a graph analytics application itself [1], [4], [5], [6]: for example, D-Galois and Gemini [6] take roughly 400 seconds and 1250 seconds, respectively, to partition clueweb12, a large web-crawl, on 256 machines, and running pagerank on the partitioned graph takes no longer than 300 seconds [1]. This demonstrates the need for faster partitioners even in existing graph processing systems.

Ideally, a graph partitioner would be (i) customizable by the application programmer and (ii) fast so that the time to partition graphs will not be much more than the time it takes to read the graphs in from disk while (iii) producing partitions competitive to those that existing systems produce.

This paper presents CuSP, a fast, customizable, streaming edge partitioning framework that aspires to this ideal. It has been carefully designed to exploit both distributed-memory and shared-memory parallelism to achieve its speed objectives.

This paper makes the following contributions:

- We present an abstract framework for graph partitioning that can express the streaming partitioning strategies used in the literature.
- We present CuSP, an implementation of this abstract partitioning framework, that can be easily customized by application programmers. CuSP utilizes a large amount of parallelism for fast partitioning.
- Evaluation of CuSP in comparison with XtraPulp [9], the state-of-the-art offline graph partitioner, shows that it can produce partitions more rapidly than XtraPulp can, and with matching or higher quality than XtraPulp can. For example, the partitioning time of a policy called Cartesian Vertex-Cut (CVC) [11], [1] in CuSP is 11.9× faster than XtraPulp on average. Furthermore, the application execution time with CVC partitions is faster than with XtraPulp partitions by 1.9× on average.

The rest of the paper is organized as follows. Section II presents background in graph partitioning. Section III presents the abstract framework for efficient streaming graph partitioning. Section IV presents the design of CuSP, an implementation of the abstract framework, and the various optimizations it uses to utilize parallelism. Section V presents an experimental evaluation of CuSP's partitioning time and partitioning quality on a large production cluster using D-Galois [1], the state-of-the-art distributed graph analytics system. Section VI discusses related partitioning work and Section VII concludes the paper.

II. Background to Partitioning

This section gives an overview of the graph partitioning approaches considered in the literature. Given k hosts and a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, G must be divided into k subgraphs $\{G_i = (V_i, E_i) | 1 \leq i \leq k\}$, such that $V_1 \cup V_2 \cup ... \cup V_k = V$ and $E_1 \cup E_2 \cup ... \cup E_k = E$.

One way to generate the subgraphs G_i is the following: (i) partition the edges of G into k subsets and (ii) add vertices to each partition for the endpoints of its edges. Therefore, if the edges connected to a given vertex v in graph G are partitioned among several subgraphs, each subgraph will have a vertex corresponding to v. We term these *proxy vertices* for the original vertex v. One of these proxy vertices is designated as the *master vertex* for this collection of proxy vertices, and the others are designates as *mirror vertices*. Figure 1b shows a partitioning of the graph in Figure 1a. During the computation, the master vertex holds the canonical value of the vertex, and it communicates this value to mirrors when needed. The average number of proxy vertices created for a given vertex in the original graph is called the average *replication factor*.

We observe that a graph partition is completely defined by (i) the assignment of edges to subgraphs, and (ii) the choice of master vertices.

It is useful to classify partitioning algorithms along two dimensions: (i) structural invariants of the subgraphs they create and (ii) how the subgraphs are created. Table I lists the examples in literature for the different classes.

A. Structural invariants of subgraphs

1) Edge-Cut: Algorithms that create *edge-cuts* assign all outgoing (or incoming) edges of a vertex v to the same partition, and the proxy node for v in that partition is made the master. These partitions are known as outgoing (or incoming) edge-cuts. Conceptually, they can also be considered to be the result of partitioning *vertices* among hosts and assigning all outgoing (or incoming) edges of those vertices to those hosts. It is convenient to view these partitions in terms of the adjacency matrix of the graph (each non-zero entry in the matrix corresponds to one edge). Outgoing edge-cuts correspond to 1D row partitions of the matrix and incoming edge-cuts correspond to 1D column partitions of this matrix. Gemini's Edge-balanced Edge-Cut (EEC) [6], Linear Deterministic Greedy (LDG) [12], Fennel [13], Leopard [14], ADWISE [15], Metis [8], Spinner [10], and XtraPulp [9] produce edge-cuts.

TABLE I: Classification of partitioning policies with examples; *streaming* class of policies can be implemented in CuSP (Leopard [14] is omitted because it is for dynamic graphs).

Class	**Invariant**	**Examples**
Streaming	Edge-Cut	EEC [6], LDG [12], Fennel [13]
	Vertex-Cut	PowerGraph [4], HVC [5], Ginger [5], HDRF [16], DBH [17]
	2D-Cut	CVC [11], [1], BVC [18], [7], JVC [18], [7]
Streaming-Window	Edge-Cut	ADWISE [15]
Offline	Edge-Cut	Metis [8], Spinner [10], XtraPulp [9]

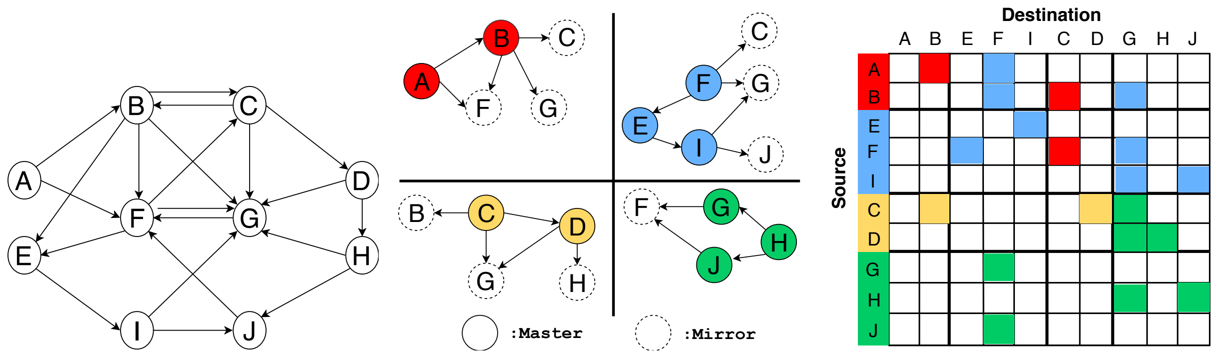
2) Vertex-Cut: General vertex-cuts, on the other hand, do not place any structural restriction on how edges are assigned to partitions, so both incoming and outgoing edges connected to a given vertex may be assigned to different partitions. PowerGraph [4], Hybrid Vertex-Cut (HVC) [5], Ginger [5], High Degree Replicated First (HDRF) [16], and Degree Based Hashing (DBH) [17] produce general vertex-cuts.

3) 2D-Cut: 2D block partitions are a special form of vertex-cuts in which the adjacency matrix of the graph is divided into blocks, and the blocks are assigned to partitions. One example is Cartesian vertex-cut (CVC) [11], [1], [7] as shown in Figure 1c using the adjacency matrix representation of the graph in Figure 1a. In CVC, rows of the adjacency matrix are partitioned among hosts using any 1D block partitioning policy, and masters are created on hosts for the vertices in the rows assigned to it. The columns are then partitioned into same sized blocks as the rows. These blocks of edges created can be distributed among hosts in different ways; one example is block distribution along the rows and a cyclic distribution along columns, as shown in Figure 1c. CheckerBoard Vertex-Cuts (BVC) [19], [18], [7] and Jagged Vertex-Cuts (JVC) [18], [7] are other 2D block partitioning policies that have been studied in the literature.

B. Streaming vs. offline partitioning algorithms

1) Streaming: These algorithms create graph partitions in a pass over the sequence of edges, so the decision to assign edges to hosts is made on the fly [12], [13], [4], [5]. These algorithms may differ in (i) the heuristics used to assign edges to hosts, (ii) *a priori* knowledge of graph properties exploited by the algorithm, and (iii) the *state* maintained during the partitioning. If decisions about edges and masters depend only on structural properties of the graph, no state needs to be maintained. In other algorithms, these decisions depend also on what has been done with previously processed edges; these algorithms must maintain a summary of the current state of the partitioning. Any streaming partitioning algorithm can be implemented using CuSP.

Many streaming algorithms try to assign roughly equal numbers of vertices and edges to the different subgraphs while minimizing the average replication factor. The hope is that this



(a) Original graph. (b) Edge-balanced Edge-Cut (EEC) partitions. (c) Cartesian Vertex-Cut (CVC) partitions.

Fig. 1: An example of partitioning a graph for four hosts using two different policies.

will balance computational load among hosts while reducing communication. LDG [12] and Fennel [13] try to co-locate vertices with their neighbors. Both LDG and Fennel need the total number of vertices in the graph to be computed beforehand, and they keep track of the edge assignment decisions made during partitioning. PowerGraph [4], HDRF [16], and DBH [17] do not require any graph properties to be pre-computed, but they maintain state to keep track of the load distribution among hosts as well as the partial degree of each vertex during partitioning. On the other hand, PowerLyra's HVC [5] does not keep any state during partitioning, but it needs the degree of each vertex to be pre-computed because it treats high-degree and low-degree vertices differently. PowerLyra's Ginger [5] uses a variant of the Fennel heuristic along with its degree-based hybrid-cut heuristic. Leopard [14] extends the Fennel heuristic to partition *dynamic* graphs. All other partitioning policies discussed in this paper, including those implemented in CuSP, partition only *static* graphs.

2) Streaming-Window: These algorithms also create graph partitions in a pass over the sequence of edges and decide to assign edges on the fly, but they maintain a window of edges scanned and use heuristics to pick and assign one of the edges in the window; the edge assigned need not be the last scanned edge. ADWISE [15] introduced such an algorithm. It may be possible to extend CuSP to handle this class of algorithms, but that is beyond the scope of this paper.

3) Offline: These algorithms take the complete graph as input and make multiple passes over the graph to create partitions [8], [20], [21], [22], [10], [9]. These algorithms typically use methods like community detection [23] and iterative clustering methods to compute high-quality partitions. Metis [8], Spinner [10], and XtraPulp [9] use offline partitioning algorithms. CuSP is not designed for such algorithms.

III. Customizable Streaming Edge Partitioning Framework

CuSP is a programmable framework for implementing streaming edge partitioners. This section describes how it is used by application programmers.

A. *Using CuSP*

The graph to be partitioned must be stored on disk in Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) format (CuSP provides converters between these and other graph formats like edge-lists). For explanation, we will assume the graph is read in CSR format (note outgoing edges in CSC correspond to incoming edges in CSR).

To specify a particular partitioning policy, the programmer must give CuSP the number of the partitions desired and provide the rules for (i) choosing master vertices and (ii) assigning edges to partitions. CuSP streams in the edges of the graph from disk and uses these rules to assign them to partitions and to choose masters.

To specify the partitioning rules, it is convenient to assume that there is a structure called `prop` that stores the number of desired partitions and the static properties of the graph such as the number of nodes and edges, the outgoing edges or neighbors of a node, and the out-degree of a node. This structure can be queried by the partitioning rule; for example, `prop.getNumNodes()` returns the number of nodes in the graph.

As mentioned in Section II, partitioning rules may be *history-sensitive*; for example, an edge may be assigned to a partition that currently has the smallest number of edges assigned to it. Each partitioning rule can define its own custom type to track the `state` that can be queried and updated by it. CuSP transparently synchronizes this `state` across hosts.

To specify the partitioning policy, users write two functions:

- **getMaster(prop, nodeID, mState, masters)**: returns the partition of the master proxy for the node `nodeID`; `masters` can be used to query the (previously) assigned partitions of the master proxies of the node's neighbors.
- **getEdgeOwner(prop, srcID, dstID, srcMaster, dstMaster, eState)**: returns the partition to which the edge `(srcID, dstID)` must be assigned; `srcMaster` and `dstMaster` are the partitions containing the master proxy of `srcID` and `dstID` nodes, respectively.

Users define the types of `mState` and `eState` tracked in `getMaster()` and `getEdgeOwner()` functions, re-

Algorithm 1 Examples of user-defined `getMaster`.

```
function CONTIGUOUS(prop, nodeID, mState)
    blockSize = ceil(prop.getNumNodes()/prop.getNumPartitions())
    return floor(nodeID / blockSize)

function CONTIGUOUSEB(prop, nodeID, mState)
    edgeBlockSize = ceil((prop.getNumEdges()+1)/prop.getNumPartitions())
    firstEdgeID = prop.getNodeOutEdge(nodeID, 0)
    return floor(firstEdgeID / edgeBlockSize)

function FENNEL(prop, nodeID, mState, masters)
    for $p = 0; p <$ prop.getNumPartitions(); $p++$ do
        score[p] = − ($\alpha$ * $\gamma$ * pow(mstate.numNodes[p], $\gamma - 1$))
    for $n \in$ prop.getNodeOutNeighbors(nodeID) do
        if n $\in$ masters then
            score[masters[n]]++
    part = p such that score[p] is highest
    mstate.numNodes[part]++
    return part

function FENNELEB(prop, nodeID, mState, masters)
    if prop.getNodeOutDegree(srcID) > constThreshold then
        return ContiguousEB(prop, mState, nodeID)
    $\mu$ = prop.getNumNodes() / prop.getNumEdges()
    for $p = 0; p <$ prop.getNumPartitions(); $p++$ do
        load = (mstate.numNodes[p] + ($\mu$ * mstate.numEdges[p])) / 2
        score[p] = − ($\alpha$ * $\gamma$ * pow(load, $\gamma - 1$))
    for $n \in$ prop.getNodeOutNeighbors(nodeID) do
        if n $\in$ masters then
            score[masters[n]]++
    part = p such that score[p] is highest
    mstate.numNodes[part]++
    mstate.numEdges[part]++
    return part
```

Algorithm 2 Examples of user-defined `getEdgeOwner`.

```
function SOURCE(prop, srcID, dstID, srcMaster, dstMaster, eState)
    return srcMaster

function HYBRID(prop, srcID, dstID, srcMaster, dstMaster, eState)
    if prop.getNodeOutDegree(srcID) > constThreshold then
        return dstMaster
    return srcMaster

function CARTESIAN(prop, srcID, dstID, srcMaster, dstMaster, eState)
    find $p_r$ and $p_c$ s.t. ($p_r \times p_c$) == prop.getNumPartitions()
    blockedRowOffset = floor(srcMaster / $p_c$) * $p_c$
    cyclicColumnOffset = dstMaster % $p_c$
    return (blockedRowOffset + cyclicColumnOffset)
```

spectively. Multiple calls to these functions with the same arguments must return the same value. CuSP permits the types of `mState` or `eState` to be `void` (when state is not required). It also permits `getMaster()` to be defined without the `masters` argument (when the master assignment of a node does not depend on those of its neighbors).

CuSP runs on the same number of distributed hosts as the number of desired partitions. After partitioning the graph, CuSP constructs a partition on each host's memory, in either CSR or CSC format (as desired by the user). These partitions can be written to disk if desired.

B. Examples of specifying partitioning policies

Algorithm 1 defines example functions for `getMaster()` that assign (the master proxy of) the given node to a partition.

TABLE II: Examples of specifying partitioning policies using CuSP; see Algorithms 1 and 2 for function definitions (Note: HVC, CVC, and FEC in [5], [11], and [13], respectively, did not use edge-balanced (EB) master assignment).

Policy	`getMaster`	`getEdgeOwner`
Edge-balanced Edge-Cut (EEC) [6]	ContiguousEB	Source
Hybrid Vertex-Cut (HVC) [5]	ContiguousEB	Hybrid
Cartesian Vertex-Cut (CVC) [11], [1]	ContiguousEB	Cartesian
Fennel Edge-Cut (FEC) [13]	FennelEB	Source
Ginger Vertex-Cut (GVC) [5]	FennelEB	Hybrid
Sugar Vertex-Cut (SVC)	FennelEB	Cartesian

`Contiguous` assigns a contiguous chunk of nodes to each partition such that the chunks assigned to different partitions are roughly equal-sized. `ContiguousEB` also assigns a contiguous chunk of nodes to each partition, but the number of outgoing edges of the nodes in each chunk are roughly equal-sized (edge-balanced). `Fennel` uses a heuristic [13] to calculate a score for each partition and assigns the node to the partition with the maximum score. The score tries to prefer partitions that have been assigned the neighbors of the node while avoiding partitions that have been assigned more nodes. `FennelEB` uses a similar heuristic [5], but the score avoids partitions that have been assigned more load, where the load is a combination of the nodes assigned to that partition and the number of outgoing edges of those nodes (edge-balanced).

Example functions for `getEdgeOwner()` are defined in Algorithm 2. `Source` assigns an edge to the partition containing the master proxy of its source. `Hybrid` [5] performs a similar assignment only if the degree of the source is below a threshold; otherwise, the edge is assigned to the partition containing the master proxy of its destination. To assign edges in `Cartesian`, the graph is viewed as a sparse matrix and the matrix is blocked in both dimensions; the number of blocks in each dimension match the number of partitions (block sizes can vary depending on the master assignment, as shown in Figure 1c). Let (i, j) be the block in the i^{th} row and j^{th} column. An edge from s to d belongs to the (m_s, m_d) block, where m_s and m_d are the masters of the source and the destination, respectively. The partitions p are considered to be in two dimensional grid of p_r and p_c. The row blocks are distributed among p_r partitions in a blocked way, and the column blocks are distributed among p_c partitions in a cyclic way (Figure 1c). This can be done using simple arithmetic.

To specify a partitioning policy, we can pick one function each from Algorithms 1 and 2. This would yield 12 different policies, and Table II lists 6 out of them (omitting `Contiguous` and `Fennel`). Each of these policies has two variants (24 policies in total) - one that reads the input graph in CSR format and another that reads it in CSC format. These policies correspond to those published in literature or variants of them, except Sugar Vertex Cut (SVC), which is a new policy that combines existing heuristics in a novel way. Several popular graph analytical frameworks use one of these policies; Gemini [6] uses EEC, PowerLyra [5] uses HVC or GVC,

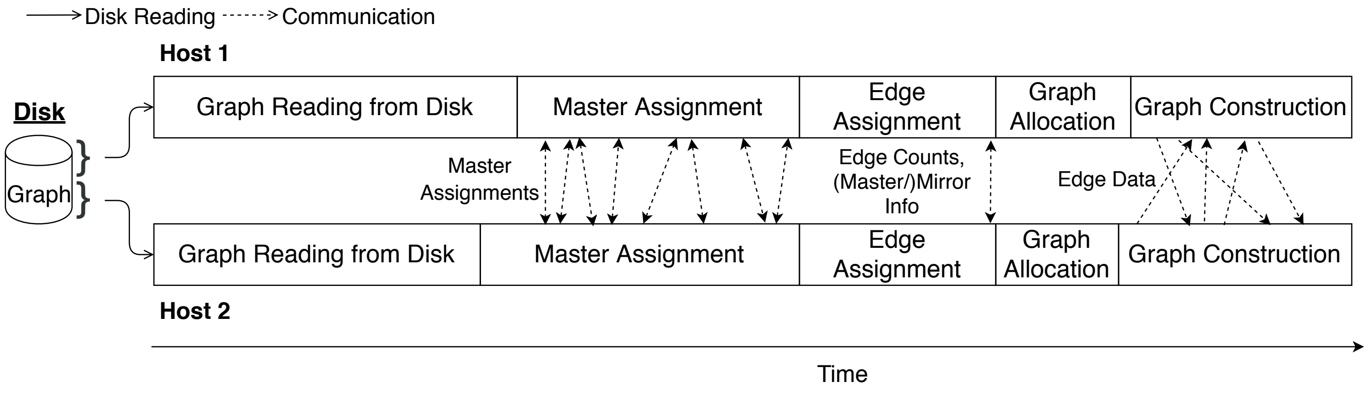


Fig. 2: Control and data flow of CuSP (partitioning state synchronization is omitted).

and D-Galois [1], [7] uses EEC, HVC, or CVC. PowerLyra introduced HVC and GVC considering incoming edges and in-degrees, so to use them in CuSP, the input graph would be read in CSC format. This illustrates the programmability and customizability of CuSP. Through the general interface of CuSP, the user can thus implement any streaming edge-cut or vertex-cut policy using only a few lines of code.

IV. Implementation of CuSP

This section describes the implementation of CuSP. Section IV-A gives an overview. Section IV-B explains the main phases of execution of the partitioner. Section IV-C explains how the implementation uses parallelism to efficiently partition graphs. Section IV-D presents optimizations used by CuSP to reduce communication overhead.

A. *CuSP Overview*

The input graph is stored on disk in CSR or CSC format. Without loss of generality, we assume the graph is stored in CSR format. Each host builds one partition of the graph in CSR or CSC format. If an edge should be assigned to the local partition, it is added to that partition; otherwise, it is forwarded to the appropriate host. CuSP handles a number of complications that arise in implementing this approach in parallel efficiently.

- Graphs in CSR or CSC format cannot be built incrementally since allocating the underlying arrays for CSR or CSC format requires knowing the number of nodes and outgoing or incoming edges for each node. Therefore, these counts must be determined before the graph can be constructed.
- Communication of edges between hosts is required since a host may process an edge and decide it belongs to a different partition than the one it is building.
- If the partitioning rules are history-sensitive, decisions made on one host will affect decisions made by other hosts. Therefore, partitioning state must be synchronized on all hosts, but doing this for every update made to the state by any host (as is done in cache-coherent systems) is expensive.

CuSP transparently handles these complications.

B. *Phases of Partitioning*

Partitioning in CuSP is split into five phases, as illustrated in Figure 2: graph reading, master assignment, edge assignment, graph allocation, and graph construction. We describe the phases below assuming partitioning state is not synchronized (its synchronization is explained in Section IV-D).

1) Graph reading: The edge array in the CSR format is divided more or less equally among hosts so that each host reads and processes a contiguous set of edges from this array. To reduce inter-host synchronization, this division is rounded off so that the outgoing edges of a given node are not divided between hosts. In effect, this approach assigns a contiguous set of vertices to each host so that each host has roughly the same number of edges, and the outgoing edges of those vertices are processed by that host. Users can change this initial assignment so that it takes nodes into consideration as well using command line arguments to assign importance to node and/or edge balancing. During this phase, each host loads its set of vertices and edges from disk into memory, so future phases will directly read them from memory.

2) Master assignment: Each host maintains its own local `masters` map from a vertex's global-ID to the partition assigned to contain its master proxy. Each host loops through the vertices whose edges it has read from disk. For each such vertex v, it determines the partition to assign the master of that vertex using `getMaster()` and stores it in `masters`. The `masters` map is periodically synchronized with that of the other hosts (more details in Section IV-D). It is also synchronized after all vertices have been assigned to partitions.

3) Edge assignment: As shown in Algorithm 3, each host h_i loops through the edges that it has read from disk. For each vertex v it is responsible for and for each host h_j, it determines (i) how many outgoing edges of v will be sent to h_j and (ii) whether the proxy vertices on h_j for the destinations of these edges will be mirrors. Once all edges have been processed, this information is sent from h_i to all other hosts. The phase concludes once a host has received data from all other hosts.

4) Graph allocation: When the edge assignment phase is complete, a host has a complete picture of how many vertices

Algorithm 3 Edge assignment phase of CuSP.

```
Input: $G_h = (V_h, E_h)$ where $V_h$ is the set of vertices read by host $h$, $E_h$ is
    the set of all outgoing edges of vertices $V_h$; $(a \neq b) \Rightarrow (V_a \cap V_b = \{\})$
Let $a$ be this host
Let outgoingEdgeCount[$h$][$s$] represent the number of outgoing edges of
    node $s$ that host $a$ will send to host $h$: initialized to 0
Let createMirror[$h$][$d$] being true represent host $h$ needs to create a mirror
    for node $d$: initialized to false
 1: for $s \in V_a$ do
 2:     for outgoing edges of $s$, $(s,d) \in E_a$ do
 3:         $h$ = GETEDGEOWNER(prop, $s, d$, masters[$s$], masters[$d$], eState)
 4:         outgoingEdgeCount[$h$][$s$]++
 5:         if $h \neq$ masters[$d$] then
 6:             createMirror[$h$][$d$] = true
 7: for all other hosts $h$ do
 8:     send outgoingEdgeCount[$h$] to $h$
 9:     send createMirror[$h$] to $h$
10: toReceive = 0
11: for all other hosts $h$ do
12:     save received outgoingEdgeCount into outgoingEdgeCount[$h$]
13:     $\forall s$, toReceive += outgoingEdgeCount[$h$][$s$]
14:     save received createMirror into createMirror[$h$]
```

and edges it will have in its partition. Each host allocates memory for its partition in CSR format and also creates a map from global-IDs to local-IDs for its vertices. Note that at this point, the host has not yet received its edges from other hosts, but by allocating memory for edges beforehand, it is possible to insert edges in parallel into the data structure as they are received from other hosts. Additionally, partitioning state is reset to initial values so calls to the user-specified functions will return the same value during the graph construction phase as they did in the edge assignment phase.

5) Graph construction: As described in Algorithm 4, each host will again loop over all of its read edges. Instead of compiling metadata like the assignment phase, it sends the edge to the appropriate host (using information returned by `getEdgeOwner()`). Edges are inserted in parallel into the CSR structure constructed in the previous phase whenever they are deserialized from an incoming message. Once all hosts have received the edges that they expect from all other hosts and have inserted their edges into the CSR structure, each host performs an in-memory transpose of their CSR graph to construct (without communication) their CSC graph if desired.

C. Exploiting Parallelism

CuSP exploits both multi-core and inter-host parallelism to perform graph partitioning quickly.

1) Parallel iteration over vertices and edges: The master assignment, edge assignment, and graph construction phases require hosts to iterate over vertices and edges while updating data structures. In CuSP, this is implemented using parallel constructs in the Galois system [24] and thread-safe data structures (user-defined functions are expected to update partitioning state using thread-safe atomics or locks provided by Galois). The system implements work-stealing among threads in a host (not among hosts), which is useful for load-balancing since threads can steal vertices that have been assigned to other threads if they finish their assigned work early.

Algorithm 4 Graph construction phase of CuSP.

```
Input: $G_h = (V_h, E_h)$ where $V_h$ is the set of vertices read by host $h$, $E_h$ is
    the set of all outgoing edges of vertices $V_h$; $(a \neq b) \Rightarrow (V_a \cap V_b = \{\})$
Let $a$ be this host
Let toReceive represent the number of edges this host expects to receive
    from all other hosts (determined in edge assignment phase)
 1: for $s \in V_a$ do
 2:     for outgoing edges of $s$, $(s,d) \in E_a$ do
 3:         $h$ = GETEDGEOWNER(prop, $s, d$, masters[$s$], masters[$d$], eState)
 4:         if $h == a$ then
 5:             construct $(s,d)$ in local CSR graph
 6:         else
 7:             send edge $(s,d)$ to host $h$
 8: while toReceive > 0 do
 9:     receive edge $(s,d)$ sent to this host
10:     construct $(s,d)$ in local CSR graph
11:     toReceive−−
12: if CSC format is desired then
13:     construct local CSC graph from CSR graph (in-memory transpose)
```

2) Parallel prefix sums: In some phases of partitioning, a vector is iterated over to determine a subset of elements that must be written into memory in the same order in which they appear in the original vector (e.g., a vector specifying how many edges to be received for each node from a particular host; some elements may be 0, which should be ignored in the write). CuSP uses prefix sums to parallelize this operation without requiring fine-grain synchronization. In the first pass, each thread is assigned a contiguous, disjoint portion of a vector to read, and it determines how many elements that it needs to write. A prefix sum is calculated from the amount of work each thread is assigned: from this prefix sum, each thread knows the location in memory that it should begin to do its writes at. Each thread reads its portion of the vector again in the second pass, at which point it will actually write data to a memory location using the prefix sum.

3) Serializing and deserializing messages in parallel: Serialization and deserialization of message buffers in the graph construction phase can be done in parallel. Each thread can serialize a node ID and all the node's edges to send to another host into its own thread-local buffer for sending. Similarly, each thread can receive buffers sent from other hosts and deserialize and process them in parallel to other threads. Since the edge assignment phase has already allocated memory for every node a host is expecting to receive and since messages from other hosts contain all the data for some particular node, threads can construct edges in parallel as no thread will interfere with other threads' writes.

D. Efficient Communication

Communication can be a bottleneck if not done efficiently. CuSP is designed to communicate efficiently using various methods of reducing communication overhead.

1) Dedicated communication thread: CuSP is typically run with as many threads as the number of cores on each host. CuSP uses a single additional hyperthread [25], [26] dedicated for communication and responsible for all sends and receives of data. This allows computation on the rest of threads to proceed without needing to pause for communication except when serializing and deserializing message buffers (which itself is done in parallel). The communication thread can use MPI or LCI [26] for message transport between hosts (LCI has been shown to perform well in graph analytics applications [26], [1]).

2) Reducing volume of communication: Since each host knows what vertices other hosts are responsible for reading, there is no need to send node ID metadata in the edge assignment phase. A host will send a vector (whose size is the number of vertices it was assigned for reading) where data at index i corresponds to the ith assigned node. The receiving end is able to map the data in this vector to a particular node based on its position.

Additionally, in edge assignment, it may be the case that a host does not have any edges to send to another host or does not need to inform the host to create an incoming node mirror. In such cases, it suffices to send a small message to that host telling it that there is no information to be sent.

3) Buffering large messages in graph construction: In graph construction, CuSP serializes a node ID and its edges in a buffer in preparation for sending them out to the owner. Instead of sending this buffer out immediately after serialization, CuSP waits until the buffer for this host is sufficiently full before sending it out. Larger buffers reduce the number of messages and the pressure on the underlying network resources [26], thereby improving the communication time significantly up to a certain buffer size. This is a tunable parameter; the evaluation uses a buffer threshold of 8MB.

4) Synchronizing partitioning state: If the partitioning rules are history-sensitive, decisions made on one host will affect decisions made by other hosts in general. Therefore, partitioning state must be synchronized on all hosts, but doing this for every update made to the state by any host, as is done in cache-coherent systems, is too expensive. Therefore, in the master assignment, edge assignment, and graph construction phases, CuSP uses periodic synchronization across all hosts to reconcile decisions made on different hosts. At a high level, this is similar to bulk-synchronous parallel execution. Execution is divided into rounds. At the beginning of each round, all hosts have the same partitioning state. During the execution of the round, they make independent updates to their copy of the partitioning state, and at the end of the round, global reductions are used to obtain a unified partitioning state (as an example, consider keeping track of the number of edges assigned to each partition). The correctness of partitioning does not depend on how often this reconciliation is done, but it can affect the quality of the partitioning. In CuSP, the number of such bulk-synchronous rounds is a parameter that can be specified at runtime. Note that if no partitioning state is used by a policy, then synchronization of that state is a no-op in CuSP.

5) Synchronizing `masters`*:* In the master assignment phase, the `masters` map is synchronized among all hosts along with the partitioning state. In other words, CuSP uses periodic synchronization for this too. Unlike the synchronization described previously, synchronization in the master assignment phase is not completely deterministic nor bulk-synchronous parallel: at the end of a round, if a host finds it has received no data, instead of waiting to receive data from other hosts, it will continue onto the next round. This lessens the effects of load imbalance in the synchronization rounds and adds a degree of non-deterministic asynchrony to the master assignment (it need not be deterministic since a master assignment decision need only be made once and saved). CuSP further optimizes its synchronization. If `getMaster()` is defined without `masters`, and partitioning state is not used, then the master assignment is a pure function, so CuSP does not synchronize `masters` among hosts. Instead, each host determines the master assignment for the nodes that need it (replicating computation instead of communication). On the other hand, if `getMaster()` is defined without the `masters` map but uses a partitioning state, then synchronization is not needed during the master assignment phase, so CuSP synchronizes it only after the phase ends.

The local `masters` map can become prohibitively big (running out-of-memory) if it stores the masters of all (global) nodes. CuSP, therefore, does not try to ensure that each host has the same local map after each synchronization round. Instead, the local map on a host is only updated with a master assignment for a node if it is going to be queried later on that host. This does not require complex analysis. During edge assignment and edge construction phases, a host only queries the master assignment of nodes whose edges it is reading and the neighbors of those nodes. After partitioning, a host would need to know the master assignment of all proxies in it, which is determined during the edge assignment phase. This information can be used to ignore master assignments for every other node. CuSP also elides communication of such assignments. Each host initially requests the master assignment of the neighbors of the nodes whose edges it is reading. In master assignment phase, assignments are sent only if a corresponding request had been received. In edge assignment phase, more master assignments are sent if the edge assigned to a host does not contain the master proxies of its endpoints. CuSP thus significantly reduces communication of `masters`.

V. Experimental Evaluation

In this section, we evaluate the policies implemented in CuSP and compare with XtraPulp [9], the state-of-the-art offline graph partitioner. Section V-A gives the experimental setup. Section V-B details the partitioning time experiments while Section V-C details the partitioning quality experiments. Section V-D examines the impact of some optimizations discussed in Section IV-D on partitioning time.

A. Experimental Setup

Experiments were conducted on the Stampede2 [27] cluster at the Texas Advanced Computing Center using up to 128 Intel Xeon Platinum 8160 ("Skylake") nodes with 48 cores with a clock rate of 2.1 Ghz, 192GB of DDR4 RAM, and 32KB L1 data cache. The cluster uses a Lustre file system, which distributes files across multiple disks to allow many

TABLE III: Input (directed) graphs and their properties.

	kron30	gsh15	clueweb12	uk14	wdc12
$\|V\|$	1,073M	988M	978M	788M	3,563M
$\|E\|$	17,091M	33,877M	42,574M	47,615M	128,736M
$\|E\|/\|V\|$	15.9	34.3	43.5	60.4	36.1
Max OutDegree	3.2M	32,114	7,447	16,365	55,931
Max InDegree	3.2M	59M	75M	8.6M	95M
Size on Disk (GB)	136	260	325	361	986

TABLE IV: Average speedup of CuSP partitioning policies over XtraPulp in partitioning and application execution time.

	Partitioning Time	Application Execution Time
EEC	21.9×	1.4×
HVC	10.2×	1.2×
CVC	11.9×	1.9×
FEC	2.4×	1.1×
GVC	2.4×	0.9×
SVC	2.3×	1.6×

nodes to read from files in parallel. Machines in the cluster are connected with a 100Gb/s Intel Omni-Path interconnect. Code is compiled with g++ 7.1.

The partitioning policies from Table II were implemented in CuSP: Edge-balanced Edge-Cut (EEC) [6], Hybrid Vertex-Cut (HVC) [5], Cartesian Vertex-Cut (CVC) [11], [1], Fennel Edge-Cut (FEC) [13], Ginger Vertex-Cut (GVC) [5], and Sugar Vertex-Cut (SVC). These policies are specified in CuSP using the functions defined in Algorithm 1 and 2; HVC, FEC, GVC, and SVC use a degree threshold of 1000 and, FEC, GVC, and SVC use $\gamma = 1.5$ and $\alpha = mh^{\gamma-1}/n^{\gamma}$, where n is the number of nodes, m is the number of edges, and h is the number of hosts. CuSP partitions read the CSR format from disk and produce partitions in the CSR format. In EEC, a host creates a partition from the nodes and edges it reads from the disk; therefore, no communication among hosts is required during partitioning. In HVC and CVC, the master assignment phase does not require communication, but edge assignment/construction phases involve communication. EEC, HVC, and CVC do not have partitioning state (so no synchronization of it is necessary). In contrast, during master assignment in FEC, GVC, and SVC, the partitioning state requires synchronization; we use 100 synchronization rounds unless otherwise specified. CuSP uses a message buffer size of 8MB unless otherwise specified.

We compare the policies implemented in CuSP with XtraPulp [9], the state-of-the-art offline partitioning system. Unlike CuSP, it only produces edge-cut partitions in which all edges of a node are assigned to the same host.

Experiments use CuSP and XtraPulp [9] with five different power-law graphs, whose properties are listed in Table III: kron30 is synthetically generated using a Kronecker generator [28] (we used weights of 0.57, 0.19, 0.19, and 0.05, as suggested by graph500 [29]) and gsh15, clueweb12, uk14 [30], [31], [32], [33], and wdc12 [34] are the largest publicly available web-crawls.

Partitioning times for CuSP for EEC, HVC, CVC, FEC,

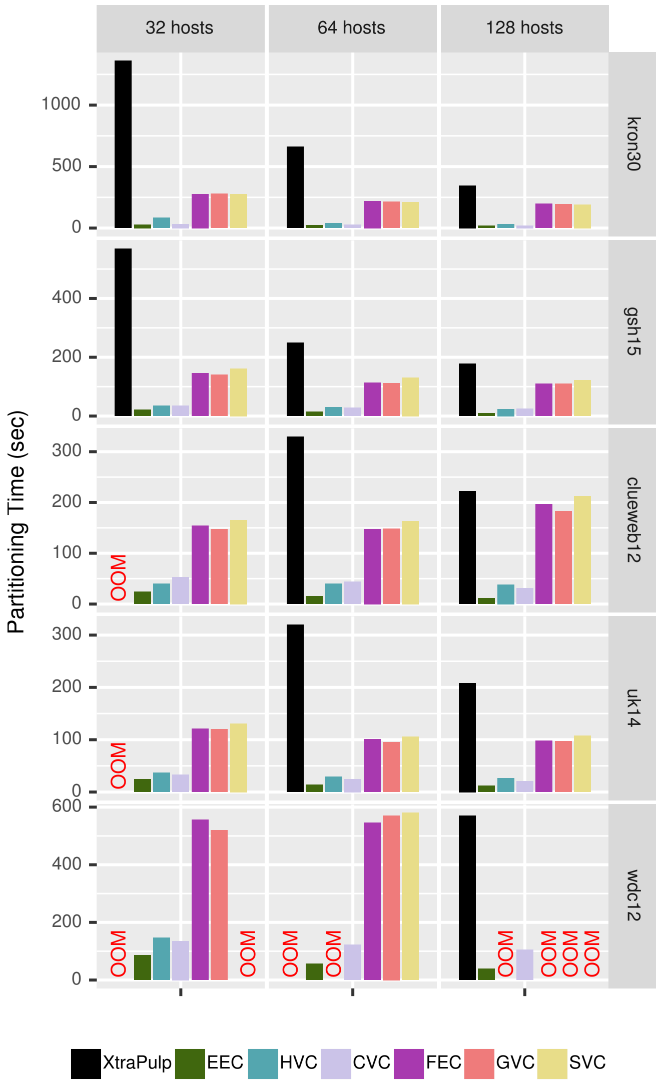


Fig. 3: Partitioning time for XtraPulp and policies in CuSP.

GVC, and SVC and XtraPulp's edge-cut are presented for 32, 64, and 128 Skylake hosts. All partitioners use directed versions of the graphs. Both systems create as many partitions as there are hosts in these experiments. Partitioning time for CuSP includes graph reading, partitioning, and graph construction, while partitioning time for XtraPulp only includes graph reading and master assignment (XtraPulp does not have built-in graph construction). The communication layer is MPI for both CuSP and XtraPulp for these experiments (CuSP can use LCI [26], but XtraPulp cannot, so for fairness we use MPI). All results presented are an average of three runs.

To evaluate quality, partitions generated by the policies are used in D-Galois [1], the state-of-the-art distributed graph analytics system, and run with breadth-first search (bfs), connected components (cc), pagerank (pr), and single-source shortest paths (sssp) on 64 and 128 Skylake nodes (cc uses partitions of the undirected or symmetric versions of the graphs and their partitioning time is omitted due to lack of

TABLE V: Data volume sent in edge assignment and graph construction phases of CuSP for 128 hosts.

		Assignment Data (GB)	Construction Data (GB)
kron30	**CVC**	71	129
	HVC	1016	76
gsh15	**CVC**	53	31
	HVC	933	0.9
clueweb12	**CVC**	52	68
	HVC	690	0.1
uk14	**CVC**	41	14
	HVC	637	1

space). The source node for bfs and sssp is the node with the highest out-degree. pr is run for a maximum of 100 iterations with a tolerance value of 10^{-6}. The communication layer is LCI [26] for all partitions (XtraPulp partitions are loaded into D-Galois, so XtraPulp being limited to MPI is not a factor). These results are also presented as an average of three runs.

B. Partitioning Experiments

Figure 3 shows the partitioning time of six different CuSP policies and XtraPulp for five different graphs at 32, 64, and 128 Skylake hosts. It is evident that CuSP partitions a graph faster than XtraPulp and Table IV shows the average speedup. Being a streaming partitioner, CuSP does not need to do extensive computation or communication on the graph when deciding how to partition, giving it an advantage over XtraPulp, which iteratively refines its partitions over time, incurring both computation and communication overhead. Additionally, XtraPulp fails to allocate memory for certain large inputs, making it unable to run for some of our experiments at 32 hosts and 64 hosts. CuSP also runs out of memory in cases where imbalance of data exists among hosts for partitioning.

EEC produces each partition from the nodes and edges read by the host from disk; therefore, no communication is required. EEC represents the minimum amount of time required by CuSP to partition the graph. On average, EEC is 4.7× faster than all other CuSP-implemented policies.

Communication in CuSP is also efficient. Table V shows the amount of data communicated in the edge assignment and graph construction phases for CVC and HVC on 128 hosts. CVC is designed to only communicate with its row or column hosts in the adjacency matrix while HVC may need to communicate with all hosts. Although HVC can communicate up to an order of magnitude more data in the worst case over what CVC communicates, HVC is only 1.2× slower than CVC on average, which shows that communication of large amounts of data does not cause a proportional degradation in partitioning time.

Figure 4 shows a breakdown of the partitioning time into time spent in the different phases of CuSP for clueweb12 and uk14 on 128 hosts. For EEC, disk reading time takes the majority of the time in graph partitioning as no inter-host communication is required. For HVC and CVC, edge assignment and graph construction involve communication, so disk bandwidth is not as major a bottleneck as it is in EEC. HVC takes more time in edge assignment because it communicates more data (as shown in Table V) and communicates with all hosts, unlike CVC. The master assignment phase in EEC, HVC, and CVC is negligible because it does not involve communication. In contrast, the main bottleneck of FEC, GVC, and SVC comes from the master assignment phase. Moreover, due to the increased likelihood of moving locally read nodes to other hosts compared to the other partitioning policies which keep their masters local, more time may be spent in the other phases as well.

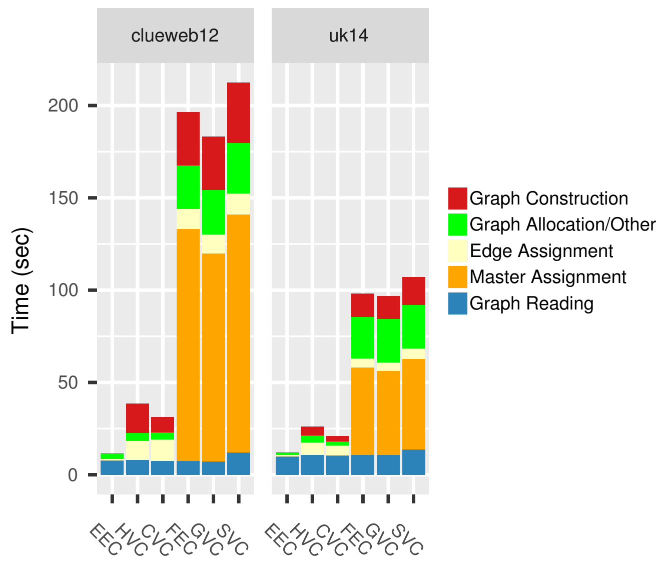


Fig. 4: Time spent by partitioning policies in different phases of CuSP for clueweb12 and uk14 on 128 hosts.

These results show that CuSP can produce partitions faster than XtraPulp can, in spite of producing more general partitions like vertex-cuts. CuSP partitions are 5.9× faster to create than XtraPulp partitions. Policies using `ContiguousEB` master assignment (EEC, HVC, CVC) are 14.0× faster than XtraPulp while policies using `FennelEB` master assignment (FEC, GVC, SVC) (and hence have a non-trivial master assignment phase) are 2.4× faster.

C. Quality Experiments

While the speed and generality of partitioning are important, quality of the partitions is paramount. Partitions may be evaluated using structural metrics such as replication factor of nodes and load balancing of nodes or edges. However, these are not necessarily correlated to execution time of applications [7]. Therefore, our evaluation focuses on the runtime of graph applications using different partitioning strategies.

Figure 5 and 6 present execution time of four graph applications for four inputs on 64 and 128 hosts, respectively, using D-Galois [1] with partitions from XtraPulp and CuSP. Table IV

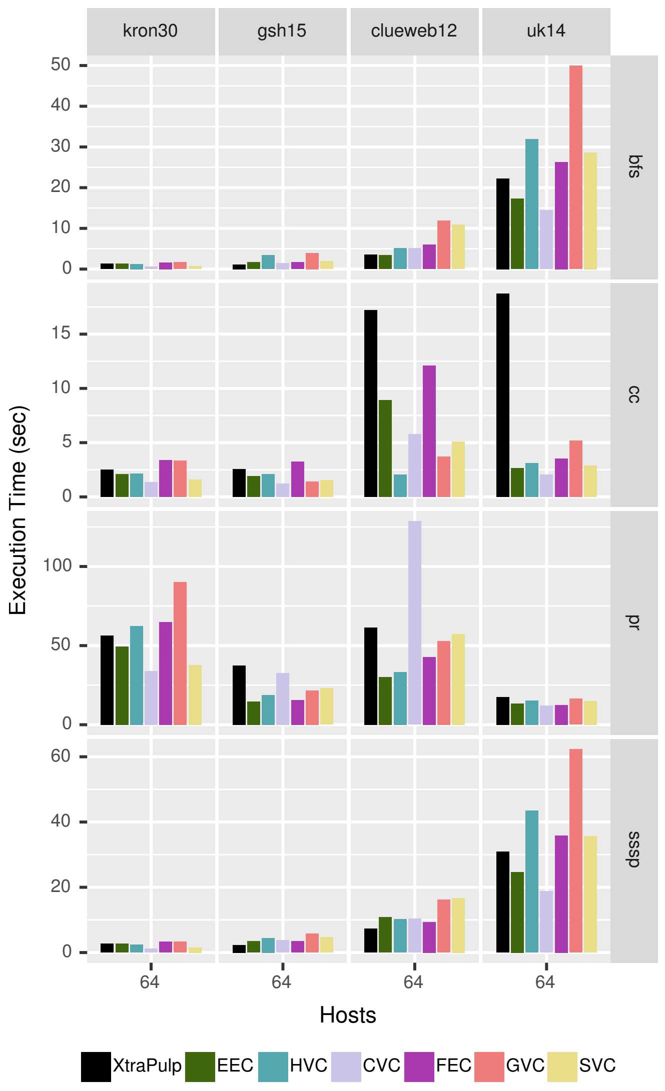


Fig. 5: Execution time of different benchmarks and inputs using partitions generated by all policies on 64 hosts.

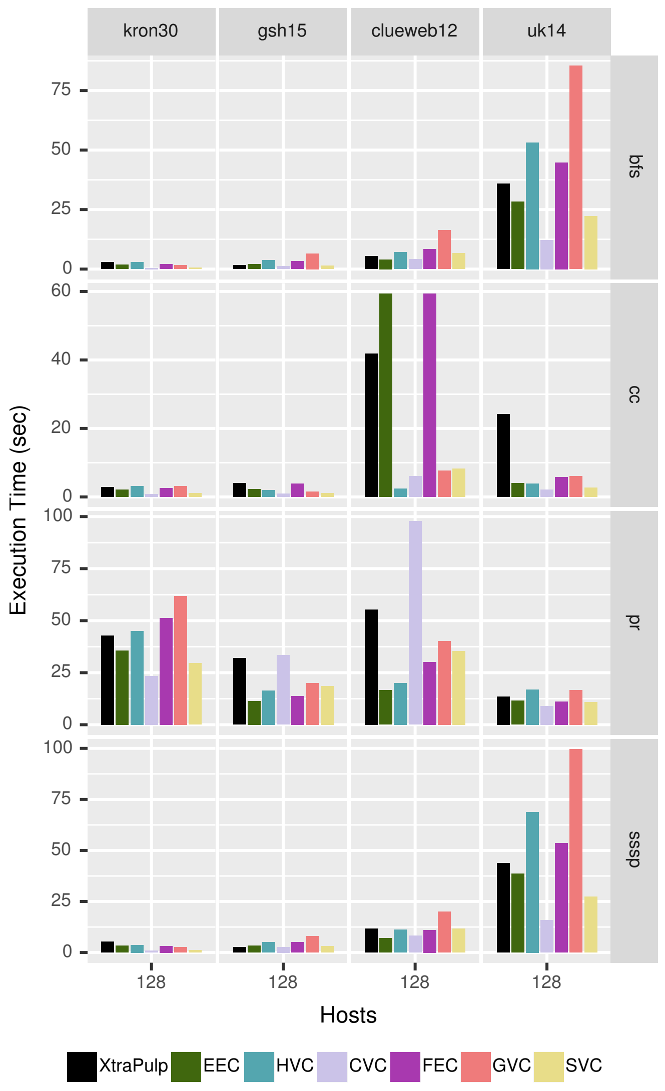


Fig. 6: Execution time of different benchmarks and inputs using partitions generated by all policies on 128 hosts.

shows the average execution time speedup of applications using CuSP partitions over those using XtraPulp partitions.

XtraPulp, EEC, and FEC are edge-cuts and are comparable in quality for many applications and inputs. They take advantage of D-Galois edge-cut communication optimizations. The difference among them is a difference of master vertex assignment, which may change performance due to graph locality and the communication required. EEC has the advantage over XtraPulp and FEC in that it is simpler and faster to generate. CVC and SVC perform better than EEC and XtraPulp in several cases as they are designed to optimize communication among hosts by reducing communication partners to hosts in the same row or column in the adjacency matrix, and D-Galois takes advantage of this aspect to improve communication. CVC does not necessarily perform the best; previous studies [7] have shown that it can perform better at a higher number of hosts. SVC may differ from CVC due to different master assignment that focuses on more balanced edges: this can lead to improved runtime or degraded runtime depending on how the partitioning ends up. For example, SVC has better runtime than CVC for pr on clueweb12 at 128 hosts because computational load balance is better for SVC due to the different master assignment used.

Finally, since both HVC and GVC are general vertex-cuts, they lack structural invariants that can be exploited by communication optimizations in D-Galois [1]. Due to this, they generally perform worse than the other partitioning policies evaluated in this paper. It is important to note that this is an artifact of the partitioning policies and not a reflection on the quality of the partitions produced by CuSP.

These results show that partitions generated from CuSP perform just as well or better than XtraPulp-generated partitions even though CuSP is more general and faster than XtraPulp. Partitions generated by CuSP perform 1.3× better on average

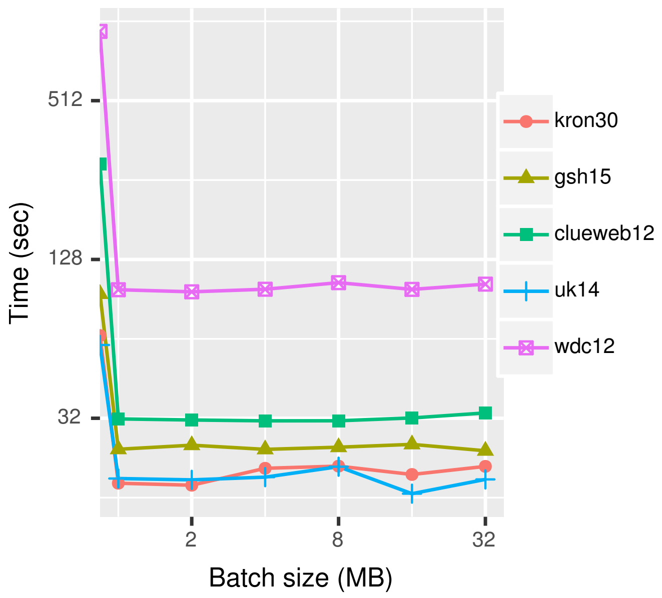


Fig. 7: Partitioning time of CVC in CuSP on 128 hosts with varying message batch sizes (log-log scale).

than XtraPulp partitions on our evaluated applications and graphs. Partitions using `ContiguousEB` master assignment (EEC, HVC, CVC) are 1.5× faster than XtraPulp while policies using `FennelEB` master assignment (FEC, GVC, SVC) are 1.1× faster.

D. *Impact of Optimizations*

In this section, we briefly examine the impact of buffering messages as well as the impact of synchronizing partitioning state during graph partitioning.

1) Message Buffering: Figure 7 shows the effect of message buffer size on partitioning time for CVC on 128 hosts. We vary the message size from 0MB (in which a message is sent out immediately rather than buffered until a size threshold) to 32MB. Increasing the buffer size to a larger value past a certain point neither degrades nor improves performance. However, sending a message immediately has negative effects on the total time it takes to partition a graph. Even buffering messages to a small amount like 4MB is enough to benefit partitioning time greatly: on average, a message buffer size of 4MB is 4.6× faster in partitioning than a message buffer size of 0MB. Therefore, buffering messages during partitioning is a key optimization in CuSP that is critical for performance.

2) Frequency of State Synchronization: We explore the effect of changing the frequency of partitioning state synchronization on the partitioning time and the partitioning quality of SVC.

Table VI shows the time it takes to do partitioning on clueweb12 and uk14 with varying rounds in the master assignment phase that perform synchronization of master assignment and partitioning state: 1 means that synchronization occurs after all hosts have assigned their nodes to partitions while 1000 means synchronization checks occur 1000 times in total

TABLE VI: Partitioning time (sec) of SVC in CuSP on 128 hosts with different number of synchronization rounds.

	Synchronization Rounds			
	1	**10**	**100**	**1000**
clueweb12	205.8	200.8	212.3	498.1
uk14	108.1	101.5	107.2	189.9

TABLE VII: Execution time of different benchmarks (sec) using partitions generated by SVC with different number of synchronization rounds in CuSP on 128 hosts.

		Synchronization Rounds			
		1	**10**	**100**	**1000**
clueweb12	**bfs**	7.2	7.5	6.6	6.2
	cc	7.5	6.7	8.3	8.5
	pagerank	26.8	31.4	35.4	30.9
	sssp	11.0	12.1	11.7	10.1
uk14	**bfs**	35.8	35.4	22.3	23.5
	cc	3.4	2.9	2.7	3.0
	pagerank	16.5	16.4	11.0	11.6
	sssp	42.6	39.9	27.3	26.8

during the phase (giving hosts a more recent global view of current partitioning state to better inform master assignment decisions). As the number of synchronizations goes ups, the time taken does not change significantly until it reaches a particularly high number of rounds such as 1000. This can be attributed to the lack of barriers between rounds in the master assignment phase: at the end of a round, if there is no data to receive from another host, execution simply continues in order to avoid blocking on other hosts.

Table VII shows the runtime of the graph analytics applications with the SVC partitions generated by the different round counts during the master assignment phase. Allowing CuSP to synchronize partitioning state more often can allow a policy to make better decisions when assigning masters, and this translates into runtime improvements to a certain point as shown with uk14. However, it may not necessarily lead to runtime improvements as shown with clueweb12, where different number of rounds leads to different behavior for the different benchmarks.

Increasing the number of synchronizations can improve performance for the application, but such gains may be offset by the increase in partitioning time. There may be a sweet spot for the number of synchronizations that differ for each policy and application: CuSP allows users to experiment with the number of synchronization rounds as a runtime parameter.

VI. Related Work

Graph partitioning is an important pre-processing step for various distributed graph analytics systems. Distributed graph analytics systems [1], [2], [3], [4], [5], [6] support different partitioning policies that trade-off factors like partitioning time, load balancing across hosts, efficient communication, etc.

a) Distributed streaming graph partitioning: It is known that graph partitioning for existing distributed graph systems such as PowerGraph [4], PowerLyra [5], Gemini [6], and D-Galois [1] can take almost as long or longer than running the application itself. CuSP is designed to reduce the partitioning time significantly. In addition, systems like Fennel [13], LDG [12], and Gemini [6] only support streaming edge-cut partitioning policies; others such as PowerGraph [4] and PowerLyra [5] only support general vertex-cut policies. Gluon [1], [7] has shown that it is important for a distributed graph analytics framework to support various partitioning policies in order to get good performance for various algorithms and inputs at different scales. CuSP is a generic and fast streaming graph partitioning framework enabling users to easily specify the desired policy that is up to an order of magnitude faster than past distributed graph systems (e.g., it takes roughly 40 seconds to partition clueweb12 on 128 Skylake nodes with CuSP compared to nearly 400 seconds D-Galois takes on 256 KNL nodes [1]).

b) Shared-memory and distributed offline graph partitioning: Systems like Metis [8] and PuLP [20] create offline partitions on a single host in shared-memory. To partition large graphs that do not fit in the memory of a single host, systems like Ugander et al. [21], Wang et al. [22], Spinner [10], and XtraPulp [9] distribute partitioning across multiple hosts. Both shared-memory and distributed offline systems can use community detection [23] and iterative clustering methods to compute high-quality partitions.

XtraPulp [9] is the state-of-the-art fast distributed offline partitioning framework. It only supports edge-cuts. In contrast, CuSP is a streaming graph partitioning framework which is roughly 6× faster than XtraPulp and is more general as it can support edge-cuts, vertex-cuts (like HVC), 2D blocking partitioning policies (like CVC), etc. It also provides an easy to use framework for users to explore various policies.

VII. Conclusion

This paper presents CuSP, a fast streaming graph partitioning framework which (i) permits users to specify the desired partitioning policy at a high level of abstraction and (ii) generates high-quality graph partitions fast. For example, it can partition wdc12, one of the largest publicly available web-crawl graphs with 4 billion vertices and 129 billion edges, in under 2 minutes with 128 machines. Our experiments show that it can produce quality partitions 6× faster than stand-alone partitioners in the literature while producing high-quality partitions and supporting a wide range of partitioning policies.

References

[1] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A Communication Optimizing Framework for Distributed Heterogeneous Graph Analytics," ser. PLDI, 2018.

[2] 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 *Proc. ACM SIGMOD Intl Conf. on Management of Data*, ser. SIGMOD '10, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[3] "Apache Giraph," http://giraph.apache.org/, 2013.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," ser. OSDI'12, 2012.

[5] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs," ser. EuroSys '15, 2015.

[6] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A Computation-centric Distributed Graph Processing System," ser. OSDI'16, 2016.

[7] G. Gill, R. Dathathri, L. Hoang, and K. Pingali, "A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms," ser. PVLDB, vol. 12, no. 4, 2018.

[8] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Sci. Comput.*

[9] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning Trillion-Edge Graphs in Minutes," in *IPDPS*, 2017.

[10] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, April 2017, pp. 1083–1094.

[11] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2D graph partitioning," in *SC*, 2013.

[12] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," ser. KDD, 2012.

[13] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming Graph Partitioning for Massive Scale Graphs," ser. WSDM '14, 2014.

[14] J. Huang and D. J. Abadi, "Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs," *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 540–551, Mar. 2016. [Online]. Available: http://dx.doi.org/10.14778/2904483.2904486

[15] C. Mayer, R. Mayer, M. A. Tariq, H. Geppert, L. Laich, L. Rieger, and K. Rothermel, "Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing," 07 2018, pp. 685–695.

[16] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "HDRF: Stream-Based Partitioning for Power-Law Graphs," ser. CIKM '15, 2015.

[17] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed Power-law Graph Computing: Theoretical and Empirical Analysis," in *NIPS*, 2014.

[18] U. V. Çatalyürek, C. Aykanat, and B. Uçar, "On Two-Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe," *SIAM J. Sci. Comput.*, 2010.

[19] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., 1994.

[20] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *IEEE Big Data*, 2014.

[21] J. Ugander and L. Backstrom, "Balanced Label Propagation for Partitioning Massive Graphs," ser. WSDM '13, 2013.

[22] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *ICDE*, 2014.

[23] X. Que, F. Checconi, F. Petrini, and J. A. Gunnels, "Scalable community detection with the louvain algorithm," in *IPDPS*, 2015.

[24] D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," ser. SOSP '13.

[25] D. Buono, T. D. Matteis, G. Mencagli, and M. Vanneschi, "Optimizing message-passing on multicore architectures using hardware multi-threading," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2014, pp. 262–270.

[26] H.-V. Dang, R. Dathathri, G. Gill, A. Brooks, N. Dryden, A. Lenharth, L. Hoang, K. Pingali, and M. Snir, "A Lightweight Communication Runtime for Distributed Graph Analytics," in *IPDPS*, 2018.

[27] D. Stanzione, B. Barth, N. Gaffney, K. Gaither, C. Hempel, T. Minyard, S. Mehringer, E. Wernert, H. Tufo, D. Panda, and P. Teller, "Stampede 2: The Evolution of an XSEDE Supercomputer," ser. PEARC17, 2017.

[28] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker Graphs: An Approach to Modeling Networks," *J. Mach. Learn. Res.*, 2010.

[29] "http://www.graph500.org."

[30] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *WWW*, 2004.

[31] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *WWW*, 2011.

[32] P. Boldi, A. Marino, M. Santini, and S. Vigna, "BUbiNG: Massive Crawling for the Masses," in *WWW*, 2014.

[33] T. L. Project, "The ClueWeb12 Dataset," 2013. [Online]. Available: http://lemurproject.org/clueweb12/

[34] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Web Data Commons," 2012. [Online]. Available: http://webdatacommons.org/hyperlinkgraph/