Streaming Hypergraph Partitioning Algorithms on Limited Memory Environments

Fatih Taşyaran, Berkay Demireller, Kamer Kaya, Bora Uçar

Streaming Hypergraph Partitioning Algorithms on Limited Memory Environments

Fatih Taşyaran*, Berkay Demireller*, Kamer Kaya*, and Bora Uçar†
*Computer Science and Engineering, Sabancı University, İstanbul, Turkey
{fatihtasyaran, bdemireller, kaya}@sabanciuniv.edu
†CNRS and LIP (UMR5668 Univ. de Lyon, ENS Lyon, Inria, UCBL 1), France
bora.ucar@ens-lyon.fr

***Abstract*—Many well-known, real-world problems involve dynamic, interrelated data items. Hypergraphs are powerful combinatorial structures that are frequently used to model such data. Many of today's data-centric have streaming data; new items arrive continuously, and the data grow over time. With paradigms such as Internet of Things and Edge Computing, such applications become more natural and more practical. In this work, we assume a streaming model where the data items and their relations are modeled as a hypergraph, which is generated at the edge. This hypergraph is then partitioned, and the parts are sent to remote nodes via an algorithm running on a memory-restricted device, such as a single board computer. Such a partitioning is usually performed by taking a connectivity metric into account to minimize the communication cost of later analyses that will be performed in a distributed fashion. Although there are many offline tools that can partition static hypergraphs effectively, algorithms for the streaming settings are rare. We analyze a well-known algorithm from the literature and significantly improve its run time by altering its inner data structure. On a medium-scale hypergraph, the new algorithm reduces the run time from 17800 seconds to 10 seconds. We then propose sketch- and hash-based algorithms, as well as ones that can leverage extra memory to store a small portion of the data to enable the refinement of partitioning when possible. We experimentally analyze the performance of these algorithms and report their run times, connectivity metric scores, and memory uses on a high-end server and four different single-board computer architectures.**

Keywords: Hypergraph partitioning, streaming hypergraphs, single-board computers.

I. Introduction

Real-world data can be complex, multi-model and multi-dimensional with irregular relations among the data items. Most of the models such as column- or row-oriented tabular representation fail in capturing the essence of knowledge contained in such data. Hypergraphs, which are generalizations of graphs, are highly flexible and appropriate for modeling and analysing such data. Therefore, they are used in various areas: DNA sequencing [1], scientific computing [2], VLSI design [3], citation recommendation [4], finding semantic similarities between documents [5], finding descriptor similarities between images [6], and classification [7].

Distributed graph and hypergraph stores became popular in today's applications. A good partitioning of the data among the compute nodes in a distributed framework is necessary to reduce the communication in the upstream applications. With the increasing popularity of data-centric paradigms such as Internet of Things and Edge Computing, the data that is fed to these stores started to have a streaming fashion. In this work, we assume that the data is generated/processed at the edge of a network and partitioned on a memory-restricted device, such as a single-board computer (SBC). There are popular algorithms to partition streaming graphs [8], [9], [10], and two recent benchmarks to evaluate the performance of such algorithms [11], [12]. Although hypergraphs have a more expressive power capability, and there are fine-tuned, optimized, offline hypergraph partitioning tools, e.g., [13], [14], [15], [3], the hypergraph partitioning problem in the streaming setting is not addressed thoroughly.

Hypergraphs generalize graphs. In a graph, the edges represent pairwise connections, whereas in a hypergraph hyperedges (or nets for short) represent multi-way connections. That is, a net connects more than two vertices. In a streaming setting, this difference makes the hypergraph partitioning problem much harder than the graph partitioning problem. For graph partitioning, when a vertex appears with its edges, the endpoint vertex IDs are implicit; hence, just the part information of the vertices is sufficient to judiciously decide on the part of the vertex at hand. However, in a hypergraph, a vertex appears with its nets and the neighbor vertices are not implicit. Hence, one needs to keep track of the connectivity among the nets and the parts to effectively assign the current vertex to a part.

We assume a streaming setting where the vertices of a hypergraph appear in some order along with the id's of their nets. The contribution of the study is three-fold:

- We take one of the existing and popular algorithms from the literature [16] and make it significantly faster by modifying its inner data structure used to store the part-to-net connectivity.
- We propose techniques to refine the existing partitioning at hand with the help of some extra memory to store some portion of the hypergraph.
- We propose and experiment with various algorithms and benchmark their run times, memory requirements, and partitioning quality on a high-end server and multiple SBCs.

The rest of the paper is organized as follows. Section II presents the notation and background on streaming hypergraph partitioning. The proposed algorithms are presented in Section III. The related work is summarized in Section IV.

Section V presents the experiments, and Section VI concludes the paper.

II. Notation and Background

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ consists of a set $\mathcal{V}$ of vertices and a set $\mathcal{N}$ of nets. A net $n \in \mathcal{N}$ contains a set of vertices, where the vertices in n are called its *pins*. The *size* of $n \in \mathcal{N}$ is the number its pins, and the *degree* of $v \in V$ is the number of nets containing v. The notation $\texttt{pins}[n]$ and $\texttt{nets}[v]$ represent the pin set of a net n, and the set of nets containing a vertex v, respectively. We assume unit weighted vertices and nets.

A *K-way partition* of a hypergraph $\mathcal{H}$, which is denoted as $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$, is a vertex partition where

- parts are pairwise disjoint, i.e., $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \leq k < \ell \leq K$,
- each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, i.e., $\mathcal{V}_k \subseteq \mathcal{V}$ and $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$,
- the union of K parts is equal to $\mathcal{V}$, i.e., $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$.

In the streaming setting, the vertices in $\mathcal{V}$ appear one after another. The elements of the stream are pairs $(v, \texttt{nets}[v])$. For each stream element, v will be partitioned, i.e., the part vector entry $\texttt{part}[v]$ will be set by the partitioning algorithm. In the strict streaming setting, each stream element $(v, \texttt{nets}[v])$ is forgotten after $\texttt{part}[v]$ is decided. Besides, none of the partitioning decisions can be revoked. In a more flexible streaming setting, a buffer with a capacity B is reserved to store some of the net sets. These vertices can then be re-processed and re-partitioned. In this setting, the cost of storing $(v, \texttt{nets}[v])$ in the buffer is $|\texttt{nets}[v]|$.

At any time point of the partitioning, the partition must be kept *balanced* by limiting the difference between the number of vertices in the most loaded and the least loaded parts. Let s be the slack denoting this difference. We say that the partition is balanced if and only if

$$abs(|\mathcal{V}_i| - |\mathcal{V}_j|) \leq s \text{ for all } 1 \leq i < j \leq K \ , \qquad (1)$$

where $abs(x)$ is the absolute value of x. For a K-way partition Π, a net that has at least one pin (vertex) in a part is said to *connect* that part. The number of parts connected by a net n is called its *connectivity* and denoted as λ_n. A net n is said to be *uncut* (*internal*) if it connects only one part (i.e., $\lambda_n = 1$), and *cut* (*external*), otherwise (i.e., $\lambda_n > 1$). Given a partition Π, if a vertex is in the pin set of at least one cut net, it is called a *boundary vertex*.

We use $\texttt{parts}[n]$ to denote the set of parts net n is connected to. Let $\Lambda(n, p) = |\texttt{pins}[n] \cap \mathcal{V}_p|$ be the number of pins of net n in part p. Hence, $\Lambda(n, p) > 0$ if and only if $p \in \texttt{parts}[n]$. There are various metrics to measure the quality of a partitioning in terms of the connectivity of the nets [17]. The one which is widely used in the literature and shown to accurately model the total communication volume of many data-processing kernels is called the *connectivity-1* metric. This cutsize metric is defined as:

$$\chi(\Pi) = \sum_{n \in \mathcal{N}} (\lambda_n - 1) \ . \qquad (2)$$

In this metric, each cut net n contributes $(\lambda_n - 1)$ to the cut size. The hypergraph partitioning problem can be defined as the task of finding a balanced partition Π with K parts such that $\chi(\Pi)$ is minimized. This problem is NP-hard even in the offline setting [17].

III. Partitioning Streaming Hypergraphs

In describing the algorithms, we assume that the stream elements come from a hypergraph $\mathcal{H}$, where a vertex of $\mathcal{H}$ is revealed with its nets. We use the notation $|\mathcal{H}|$ to denote the number of pins in this hypergraph while analyzing algorithms.

The simplest partitioning algorithm one can use in the streaming setting is random partitioning, RANDOM, which assigns the vertex in a stream element to a random part, while keeping the partitioning always balanced as shown in Algorithm 1. In the algorithm, p is the candidate part, p_{min} is the part ID having the least number of vertices, and $rand(1, K)$ chooses a random integer in between $[1, K]$. When the difference between the number of vertices is equal to s, v cannot be assigned to p since this decision makes the partitioning unbalanced.

Algorithm 1: RANDOM

```
Input: K, s
Output: part[·]
for all (v, nets[v]) in streaming order do
    p ← rand(1, K)
    while |V_p| − |V_{p_min}| = s do
        p ← rand(1, K)
    part[v] ← p
    if p = p_min then
        Update p_min
return part
```

A. Min-Max Partitioning

MINMAX is a well-known approach proposed for streaming hypergraph partitioning [16]. The approach, whose pseudocode is given in Algorithm 2, keeps track of net connectivity, i.e., which net is connected to which part, by keeping a net set $\texttt{p2n}[i]$ for each $1 \leq i \leq K$. Each streaming vertex v is assigned to the part p with the largest intersection set $\texttt{p2n}[p] \cap \texttt{nets}[v]$ that does not violate the balance constraint. After setting $\texttt{part}[v]$ to p, there will not be an additional cost for each net in $\texttt{p2n}[p] \cap \texttt{nets}[v]$. Hence, the maximum intersection cardinality is a good greedy decision. The downside of this approach is that there is no way of knowing if any of v's nets are connected to a part i without checking $\texttt{p2n}[i]$. This approach requires many unnecessary checks since all parts need to be considered even if most of them are not connected to the vertex. This problem is exacerbated when K is large, which is the case for many real-life applications. Overall, even when the cost of the intersection computation is $\mathcal{O}(1)$ per net, the algorithm takes $\mathcal{O}(K \times |\mathcal{H}|)$. This is not acceptable, since K can be in the order of thousands, and $|\mathcal{H}|$ can be huge for streaming, massive-scale hypergraphs.

Algorithm 2: MINMAX

Input: K, s
Output: $\texttt{part}[\cdot]$
for i *from* 1 *to* K **do**
  $\texttt{p2n}[i] \leftarrow \emptyset$
for *all* $(v, \texttt{nets}[v])$ *in streaming order* **do**
  $saved \leftarrow -1$
  for i *from* 1 *to* K **do**
    if $|\mathcal{V}_i| - |\mathcal{V}_{p_{min}}| < s$ **then**
      if $|\texttt{p2n}[i] \cap \texttt{nets}[v]| > saved$ **then**
        $saved \leftarrow |\texttt{p2n}[i] \cap \texttt{nets}[v]|$
        $p \leftarrow i$
  $\texttt{part}[v] \leftarrow p$
  $\texttt{p2n}[p] \leftarrow \texttt{p2n}[p] \cup \texttt{nets}[v]$
  if $p = p_{min}$ **then**
    Update p_{min}
return part

Algorithm 3: MINMAX-N2P

Input: K, s
Output: $\texttt{part}[\cdot]$
$save[\cdot] \leftarrow$ an array of size K
$mark[\cdot] \leftarrow$ an array of size K with all -1s
$pids[\cdot] \leftarrow$ an array of size K
$indx[\cdot] \leftarrow$ an array of size K
for *all* $(v, \texttt{nets}[v])$ *in streaming order* **do**
  $active \leftarrow 0$
  for $n \in \texttt{nets}[v]$ **do**
    if n *appears for the first time* **then**
      $\texttt{n2p}[n] \leftarrow \emptyset$
    for $i \in \texttt{n2p}[n]$ **do**
      if $mark[i] \neq v$ **then**
        $mark[i] \leftarrow v$
        $active \leftarrow active + 1$
        $pids[active] \leftarrow i$
        $save[active] \leftarrow 1$
        $indx[i] \leftarrow active$
      else
        $save[indx[i]] \leftarrow save[indx[i]] + 1$
  $saved \leftarrow -1$
  for j *from* 1 *to* $active$ **do**
    $i \leftarrow pids[j]$
    if $|\mathcal{V}_i| - |\mathcal{V}_{p_{min}}| < s$ **then**
      if $save[j] > saved$ **then**
        $saved \leftarrow save[j]$
        $p \leftarrow i$
  $\texttt{part}[v] \leftarrow p$
  for $n \in \texttt{nets}[v]$ **do** $\texttt{n2p}[n] \leftarrow \texttt{n2p}[n] \cup \{p\}$;
  if $p = p_{min}$ **then** Update p_{min} ;
return part

B. Using Net-to-Part Information and Finding Active Parts

We observe that the performance problem in Algorithm 2 arises, because the connectivity among the nets and the parts is stored from the parts' perspective. However, if this information had been organized from the perspective of the nets it would be possible to identify the *active* parts that are connected to at least one net of the current vertex v at hand. Algorithm 3 describes the pseudocode of this approach. For an efficient computation, it uses four auxiliary arrays, $save$, $mark$, $pids$, and $indx$. Each of these arrays is of size K. However, they are only initialized once, and no expensive reset operation with complexity $\Theta(K)$ is performed after a vertex is processed. Thanks to these arrays, when a part is first identified to be connected to one of the nets in $\texttt{nets}[v]$, it is marked to save a single net and placed into the active part array. Once it is placed, the next access to the same part (but for a different net) will only increase the savings of this part. Both these operations can be performed in constant time and no loop over all the parts is required.

MINMAX as proposed in [16], and as described in Algorithm 2, has been used in the literature to benchmark novel algorithms for scalable hypergraph partitioning [18], [19]. For instance, it is reported that a hypergraph with 0.43M vertices and 180M pins is partitioned into $K = 128$ parts in around 1000 seconds [18]. On the other hand, the variant in Algorithm 3 can partition a hypergraph with 1.1M vertices and 228M pins into $K = 2048$ parts in around 200 seconds.

Considering that K is at most in the order of tens of thousands, the extra memory due to the four additional arrays is not prohibitive. On the other hand, both MINMAX and MINMAX-N2P use approximately the same amount of memory to store the connectivity information. That is the total number of entries in $\texttt{n2p}[\cdot]$ and $\texttt{p2n}[\cdot]$ arrays are the same, and exactly n more than the (*connectivity* - 1) metric given in Eq. 2. This being said, MINMAX-N2P uses slightly more memory since unlike K, $|\mathcal{N}|$ is not known beforehand in the streaming setting, and a dynamic data structure with more overhead is required to organize the connectivity as in $\texttt{n2p}$.

C. MINMAX *Variants Using Less Memory*

Depending on the hypergraph and the number of parts, the memory requirements of the two previous approaches can be too large. For streaming data, there is no upper limit, which is obviously a problem for SBCs. In this subsection, we propose variants of MINMAX that can use a fixed amount of memory.

1) MINMAX-Lℓ*:* As explained above, the total memory spent for $\texttt{n2p}$ grows as long as the data is streaming. To avoid this, while working similar to Algorithm 3, MINMAX-Lℓ restricts the maximum *length* of each $\texttt{n2p}[\cdot]$ to ℓ. When a new part p is being added to a $\texttt{n2p}[n]$ for a net $n \in \mathcal{N}$, if $|\texttt{n2p}[n]| = \ell$, a random part id from $\texttt{n2p}[n]$ is chosen and replaced with p. Although the connectivity information is only kept in a lossy fashion, we expect it to guide the partitioning decisions for sufficiently large ℓ values.

2) MINMAX-BF*:* Bloom filters (BF) are memory-efficient and probabilistic data structures used to answer set membership queries [20]. Compared to the traditional data structures

Algorithm 4: MINMAX-Lℓ

Input: K, s, ℓ
Output: part$[\cdot]$
$\cdots$ same as Algorithm 3
for *all* $(v, \texttt{nets}[v])$ *in streaming order* **do**
  $\cdots$ same as Algorithm 3
  for $n \in \texttt{nets}[v]$ **do**
    if $|\texttt{n2p}[n]| < \ell$ **then**
      $\texttt{n2p}[n] \leftarrow \texttt{n2p}[n] \cup \{p\}$
    else if $p \notin \texttt{n2p}$ **then**
      $idx \leftarrow rand(1, \ell)$
      $\texttt{n2p}[idx] \leftarrow p$
  if $p = p_{min}$ **then**
    Update p_{min}
return part

Algorithm 5: MINMAX-BF

Input: K, s, m
Output: part$[\cdot]$
BF $\leftarrow \emptyset$ (creates an m-bit, all zero sequence)
for *all* $(v, \texttt{nets}[v])$ *in streaming order* **do**
  $saved \leftarrow -1$
  for i *from* 1 *to* K **do**
    if $|\mathcal{V}_i| - |\mathcal{V}_{p_{min}}| < s$ **then**
      $saved_i \leftarrow 0$
      for $n \in \texttt{nets}[v]$ **do**
        if BF.query$((n, i))$ **then**
          $saved_i \leftarrow saved_i + 1$
      if $saved_v > saved$ **then**
        $saved \leftarrow saved_i$
        $p \leftarrow i$
  part$[v] \leftarrow p$
  for $n \in \texttt{nets}[v]$ **do**
    BF.insert$((n, p))$
  if $p = p_{min}$ **then**
    Update p_{min}
return part

such as arrays, sets, or hash tables used for the same task, a BF occupies much less space while allowing false positives. If the item is a member of the set, BF always returns true. If the item is not in the set, BF most likely answers correctly but can also return a false positive.

A Bloom Filter, which employs an m-bit sequence, uses k hash functions to find the indices of bits and sets them to 1 to mark the existence of a new. To answer a query for an item x, it simply checks the corresponding k hash functions on x, and answers positively if each of the corresponding bits is 1. An important parameter for a BF is its false positive probability. Assuming the hash functions are independent of each other, when n items are inserted into a BF, kn bits are altered. Hence, the probability of a bit staying zero is $(1 - 1/m)^{kn} \approx e^{-kn/m}$, and the false-positive probability can be computed as $\left(1 - e^{-kn/m}\right)^k$.

The BF variant of MINMAX is similar to Algorithm 2. However, instead of p2n, it leverages a Bloom Filter BF to store connectivity tuples (n, p) which means that the net n has a pin at part p. For a given $(v, \texttt{nets}[v])$, it goes over all the parts, and for each net in $\texttt{nets}[v]$, it queries the corresponding tuple within the BF. Then it chooses the part with the most number of positive answers. The pseudocode of this approach is given in Algorithm 5. As in MINMAX-Lℓ, using a BF limits and fixes the amount of memory that will be used to store the connectivity among the nets and the parts.

3) MINMAX-MH*:* An approach fundamentally different from MINMAX-Lℓ and MINMAX-BF is to completely discard the connectivity information and try to cluster similar vertices with similar $\texttt{nets}[\cdot]$ sets into the same parts. A natural tool for this task is hashing. We use a MinHash-based approach [21], MINMAX-MH, to find the part id for a given vertex. For implementation, we use k hash functions in the form of $h_i(x) = a_i x + b_i \bmod q$ where $1 \leq i \leq k$, q is a prime number, a_i and b_i are random integers chosen from $[0, q)$ per hash function. Given $(v, \texttt{nets}[v])$, this approach first computes $(\alpha_1, \alpha_2, \ldots, \alpha_k)$ where $\alpha_i = \min_{n \in \texttt{nets}[v]}\{h_i(n)\}$. Then the part id for v is computed as

$$p = \left(\prod_{i=1}^{k} \alpha_i\right) \bmod K.$$

If v cannot be assigned to p due to the balance constraint, the approach tries the next part in the natural order, $(p+1 \bmod K)$ until a suitable part is found.

It is intuitive to think that vertices with similar net sets will end up with closer hash values which will result in assigning them in the same part. The fact that there is no need for additional memory to keep net-part connectivity unlike the previous algorithms makes this approach suitable for low memory environments.

D. Buffering and Refining

Revoking the partitioning decisions is impossible for the strict streaming setting. This is so as the net sets are forgotten after the corresponding vertex is put to a part. We can overcome this by using an additional buffer to keep the $\texttt{nets}[\cdot]$ sets. With such a buffer, one can revisit the buffered vertices and reassign them to a different part if such a reassignment improves the cutsize. A high-level description of this approach is given in Algorithm 6.

Algorithm 6 works along the same lines of MINMAX-N2P. In addition to finding the part id for a vertex v, after v is processed it can be chosen to be inserted to the buffer BUF. Once the buffer is full, the algorithm goes over all the vertices in the buffer $passes$ times. For each vertex u, first the $leaveGain$ of u is computed which is the change in the (*connectivity* - 1) metric when u is removed from $\texttt{part}[u]$. Then if u is decided to be movable, it is removed from $\texttt{part}[u]$. A new part p is then found and u is moved to p. We desgined three strategies,

Algorithm 6: MINMAX-N2P-REF

```
Input: K, s, B, passes
Output: part[·]
··· same as Algorithm 3
BUF ← ∅ (an empty buffer that can store B pins)
for all (v, nets[v]) in streaming order do
    ··· same as Algorithm 3
    if isBufferable(v) then
        BUF.insert(v)
        if BUF.isFull() then
            for i from 1 to passes do
                for u ∈ BUF do
                    Compute leaveGain for u
                    if isMoveable(u) then
                        Remove u from part[u]
                        Find the best part p for u
                        part[u] ← p
                        for n ∈ nets[u] do
                            n2p[n] ← n2p[n] ∪ {p}
                        if p = p_min then
                            Update p_min
            BUF ← ∅
return part
```

namely REF, REF_RLX and REF_RLX_SV, for MINMAX-N2P-REF, which differ on how they behave for the functions isBufferable(·) and isMoveable(·):

- The first strategy REF buffers every vertex but finds a new best part p if and only if the corresponding $leaveGain > 0$. That is it only modifies part when it is probable to reduce the *connectivity* - 1 metric. Hence, it is a restricted strategy while exploring the search space.
- The second strategy REF_RLX buffers every vertex and finds a new best part p for all the vertices in BUF. Hence, compared to the previous one it is a *relaxed* strategy.
- The third strategy REF_RLX_SV only buffers the vertices with small net sets and finds a new best part p for all the vertices in BUF. It aims to reduce the overhead of refining while keeping its gains still on the table.

To compute $leaveGain$, one also needs to keep track of the number of pins of each net residing at each part. This almost doubles the memory requirement of the refinement heuristics compared to MINMAX, since for every connectivity entry stored in n2p , an additional positive integer is required. That is the original entry shows that "*net n is connected to part p*", and the additional entry required for refinement shows that "*net n has k pins in part p*". The refinement-based algorithms require this information since when $k = 1$, they can detect a gain on the connectivity.

IV. RELATED WORK

There exist excellent offline hypergraph partitioners in the literature such as PaToH [14] and HMetis [3]. Recently, more tools are developed focusing on different aspects and using different approaches. Deveci et al. [15] focus on handling multiple communication metrics at once, Mayer et al. [18] focus on the speed, and Schlag et al. [22] focus more on the quality by using a more advanced refinement mechanism.

Faraj et al. [23] have recently proposed a streaming graph partitioning algorithm which yields high quality partitioning on streaming graphs utilizing buffering model. Jafari et al. [24] have proposed a fast parallel algorithm which processes the given graph in batches.

The streaming setting has not been analyzed thoroughly on hypergraphs except the work by Alisarth et al. [16] which proposes the original MINMAX algorithm at once. We improved this algorithm significantly by altering its inner data structure used to store the part-to-net connectivity. Furthermore, we proposed techniques to refine the existing partitioning at hand with the help of some extra memory.

TABLE I: Hypergraphs used for the experiments.

Matrix	Size	Pins	Max. Deg	Avg. Deg	Deg. Var.
coPapersDBLP	0.5M	30.5M	3.2K	56.41	66.23
hollywood2009	1.1M	227.8M	11.4K	99.91	271.69
soc-LiveJournal1	4.8M	68.9M	13.9K	14.23	42.30
com-Orkut	3.0M	234.3M	33.3K	76.28	153.92
uk-2005	39.4M	936.3M	1.7M	23.72	1654.56
webbase2001	118.1M	1.0B	816.1K	8.63	141.79

V. EXPERIMENTAL RESULTS

We tested the proposed algorithms on hypergraphs created from six matrices downloaded from the SuiteSparse Matrix Collection (*https://sparse.tamu.edu/*). For each $n \times n$ matrix, we create a column-net hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ where the vertices (nets) in $\mathcal{V}$ (in $\mathcal{N}$) correspond to the rows (columns) of the matrix. The properties of the resulting hypergraphs are given in Table I. Moreover, we tested the algorithms on a cutting-edge server and multiple SBCs. Since the variants studied in this work have different time, memory, and quality tradeoff characteristics, we used a set of single-board computers to observe their performance. These SBCs have different number of cores and different amounts of memory. The specifications of these architectures are as follows:

- **Server:** Intel Xeon Gold 6140, 2.30GHz, 256GB RAM, gcc 5.4.0, Ubuntu 16.04.6.
- **LattePanda:** Intel Atom x5-Z8350, 1.44GHz, 4096MB DDR3L @ 1066 MHz RAM, gcc 5.4.0, Ubuntu 16.04.2
- **Pi:** Broadcom BCM2837, 1.2GHz, 1024MB LPDDR2 @ 400 MHz RAM, gcc 6.3.0, Raspbian 9
- **Odroid:** ARM Cortex-A15, 2GHz and Cortex-A7 @1.3GHz, 2048MB LPDDR3 @ 933 MHz, 2048MB LPDDR3 @ 933 MHz, gcc 7.5.0, Ubuntu 18.04.1

We implemented algorithms in C++ and compiled on each device separately with the gcc version available in each SBC. For each hypergraph, we created three random streams with different vertex orderings. Although it is an offline partitioner and is not suitable for the streaming setting, we used PaToH v3.3 [14] to evaluate the quality and performance of the streaming algorithms. As balance constraint, we used a dynamic value s in Eq. 1 computed by using a constant *allowed imbalance ratio* $\beta = 0.1$, for which values in the range 5%-10% are common in the partitioning literature. That is, while processing the rth stream element, the allowed slack

is set to $s = \texttt{max}(1, \beta \times (r/K))$. This translates to obtaining balance by settign the slack s to β times the average part weight at any time during partitioning.

TABLE II: Comparison of the proposed streaming hypergraph partitioning algorithms and the state-of-the-art on the **Server** and coPapersDBLP.

Parts				
256	Algorithm	Run Time(sec)	Cut($\times 10^6$)	Memory(MB)
	MinMax	13952.12	1.731	18.60
	MinMax-n2p	7.40	1.731	36.78
	MinMax-L3	2.20	2.543	15.00
	MinMax-L5	2.60	1.913	25.41
	MinMax-BF(4)	496.90	3.280	2.64
	MinMax-BF(16)	1479.19	3.273	2.54
	MinMax-MH(4)	1.49	11.214	0.18
	MinMax-MH(16)	5.37	13.464	0.19
	Random	0.04	23.817	0.00
2048	Algorithm	Run Time(sec)	Cut($\times 10^6$)	Memory(MB)
	MinMax	17823.45	3.096	28.65
	MinMax-n2p	10.64	3.096	44.10
	MinMax-L3	2.61	5.217	15.00
	MinMax-L5	3.25	4.168	25.10
	MinMax-BF(4)	3664.48	4.508	2.59
	MinMax-BF(16)	11882.30	4.655	2.57
	MinMax-MH(4)	1.51	15.134	0.12
	MinMax-MH(16)	5.38	18.101	0.15
	Random	0.07	28.992	0.00

Table II reports the run times, cut, and memory usage of the proposed MinMax variants on the hypergraph CoPapersDBLP. The first two rows show the effectiveness of the modification on MinMax. The modified version MinMax-n2p is $\approx 1700x$ faster on this hypergraph while using 16–18MB more memory. This is due to the reduced number of unnecessary operations on unrelated parts while computing the part savings. The MinMax-L3 and MinMax-L5 variants obtain partitions comparable to MinMax in terms of quality (i.e., with respect to *connectivity*-1) while using less memory.

For MinMax-BF and MinMax-MH, we use 4 and 16 hash functions, and for the former, we use 20M bits. Although being fast, the partitioning quality of the MinHash variant is half of the Random. On the contrary, the Bloom Filter variant seems to work well with a comparable partitioning quality. However, it is slow since when BF is used instead of n2P, one needs to go over all the parts to compute their savings. Still, it is very promising in terms of memory/quality trade-off and enables a scenario in which a device with a small memory is used in partitioning on a network edge.

Figure 1 compares MinMax-n2p's performance to those of a hypothetical streaming tool based on PaToH. That is the hypergraphs are partitioned by PaToH (with SPEED and DEFAULT configurations) and the run times, cuts (connectivities), and memory requirements are normalized with respect to those of MinMax-n2P for hypergraphs coPapersDBLP, holywood2009, socLiveJournal1, com-Orkut, and uk2005. The experiments show that on these hypergraphs, the DEFAULT configuration can be 10–30% better in terms of partitioning quality. However, it can also be more than $100\times$ slower (see the run time ratio bar for SocLiveJournal1). Furthermore, PaToH uses 12–18$\times$ more memory compared to MinMax-n2P. Note that PaToH, or any other offline partitioner, is not suitable for the streaming setting. The results are only given to demonstrate that there is room for improvement on MinMax-n2p especially in terms of partitioning quality. This in turn justifies the attempts for refining the partitioning throughout streaming.

TABLE III: Effect of the number of $passes$ on refinement algorithms; the results are averaged for $K = 256$ and $K = 2048$. The experiments are executed on the **Server**. All of the values are normalized with respect to those of MinMax on the same experiment.

			Run Time			Cut		
Matrix	Buf.	Passes	R	RR	RRS	R	RR	RRS
coPapers	0.15	2	2.8	4.5	2.1	0.84	0.82	0.87
		4	3.8	6.6	2.8	0.83	0.81	0.87
		8	6.0	10.8	4.2	0.83	0.79	0.86
hollywood	0.15	2	4.0	4.4	3.8	0.97	0.96	0.97
		4	5.8	6.4	5.6	0.97	0.95	0.96
		8	9.6	10.2	8.8	0.96	0.95	0.97
soc-Live	0.15	2	3.8	3.8	3.4	0.96	0.94	0.95
		4	5.5	5.7	4.9	0.96	0.94	0.94
		8	9.0	9.3	7.8	0.96	0.94	0.94

To find a good value for $passes$ in the refinement-based algorithms, we performed 2, 4, and 8 passes over the buffered vertices and measured the run time and partitioning quality of REF, REF_RLX, and REF_RLX_SV. Table III presents the results of these experiments with a buffer capacity $B = 0.15\times |\mathcal{H}|$ for each hypergraph $|\mathcal{H}|$. The results show that although refinement can be useful for reducing the connectivity, its overhead is significant. Furthermore, after 2 passes, there is only a minor improvement on the partitioning quality. Using $passes = 4$ reduces the cut size for a large number of experiments, and therefore we use this setting in the rest of the experiments.

Figure 2 shows the run times (in seconds), connectivity scores (normalized with respect to MinMax-n2p), and memory usages (in MBs) of the refinement heuristics, MinMax-n2p, and MinMax-L5 on all hypergraphs in Table I. The experiments are executed on the **Server**. An imbalance ratio of 0.1 and $passes = 4$ are used for the experiments. As the results show, the refinement based heuristics improve the partitioning quality in between 5–20% depending on the hypergraph. Furthermore, when the buffer size is increased, these heuristics tend to improve the quality better. Besides, for the two largest hypergraphs in our experiments, REF_RLX_SV is much faster than the other two refinement heuristics REF and REF_RLX with a similar improvement over the partitioning quality and the same memory requirement. Hence, it can be a good replacement to the original MinMax-n2p heuristic with no refinement, if the partitioning quality has the utmost importance.

Table IV shows the run time performance of the proposed algorithms on different architectures and for $K = 32, 256, 2048$ and 16384. The hypergraph socLiveJournal1 is used for

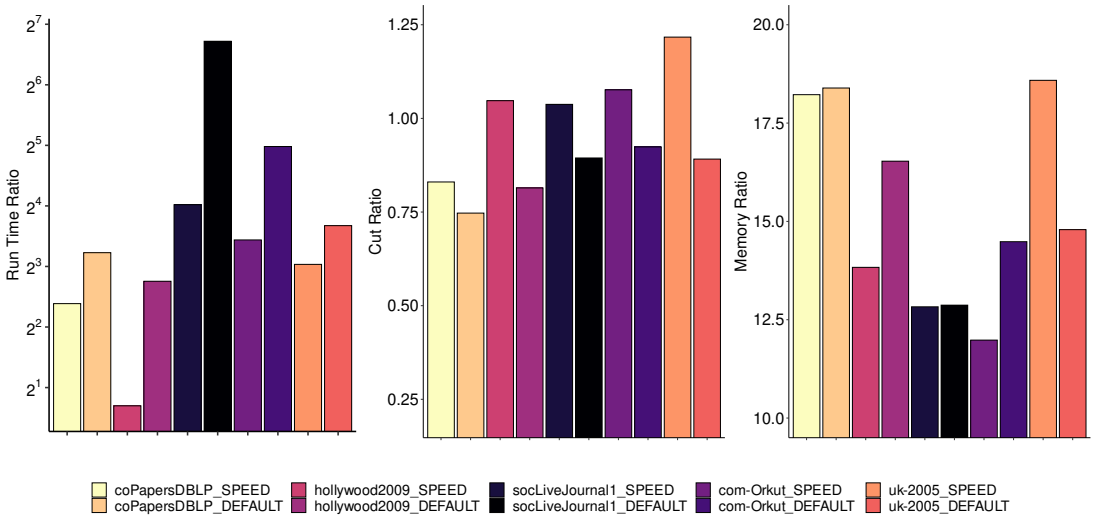

Fig. 1: Run times, cuts and memory usages of PaToH normalized with respect to those of MINMAX-N2P. The experiments are executed on the **Server**.

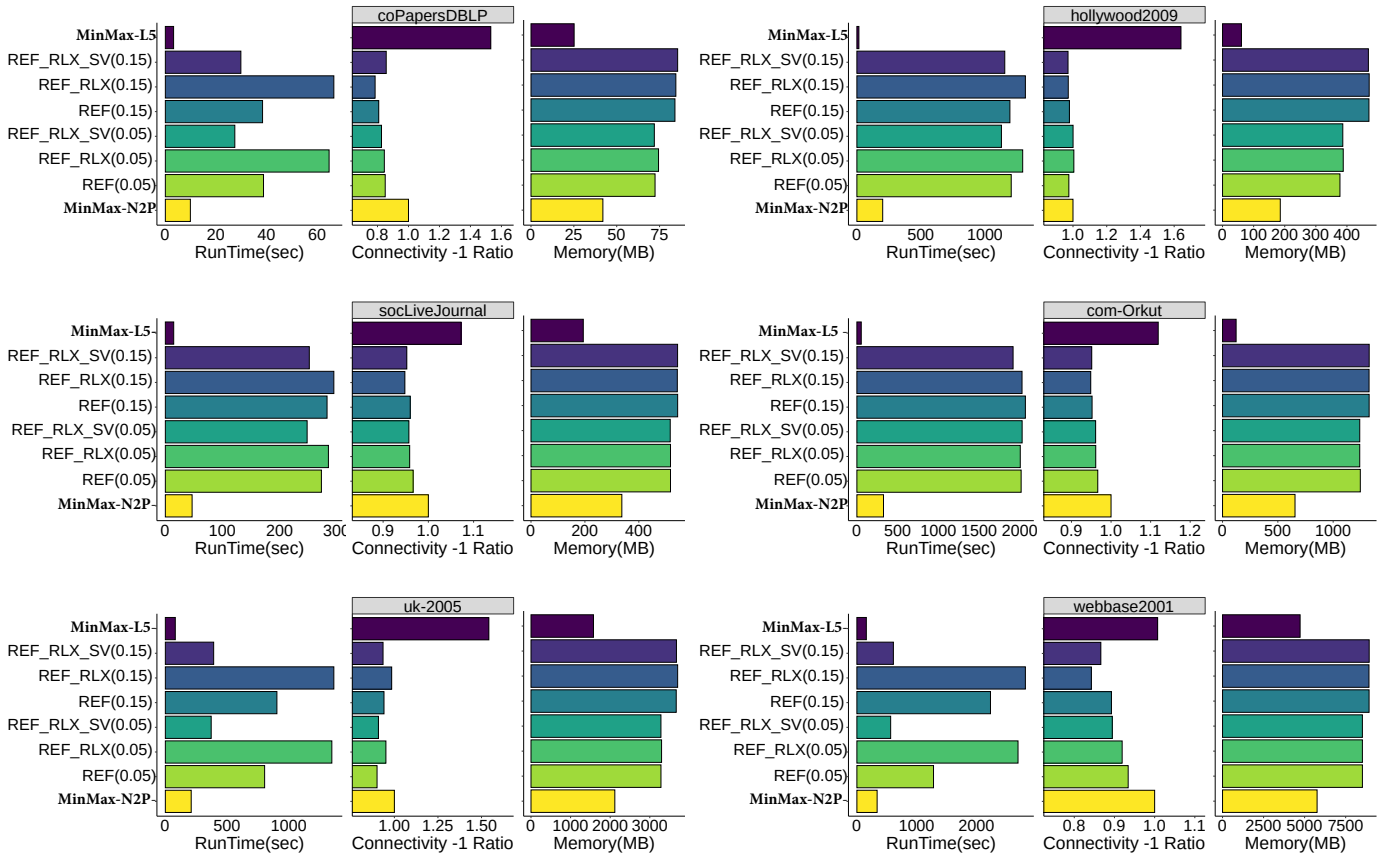

Fig. 2: The run times (in seconds), connectivity scores (normalized with respect to MINMAX-N2P), and memory requirements (in MBs) of the refinement heuristics, MINMAX-N2P, and MINMAX-L5 on all hypergraphs. The experiments are executed on the **Server** for $K = 2048$. An imbalance ratio of $\beta = 0.1$ is used for all experiments and $passes = 4$ is used for the refinement heuristics. In the figure, the parameter for these heuristics shows the parameter θ to find the buffer size $B = \theta \times |\mathcal{H}|$ for each hypergraph $\mathcal{H}$.

TABLE IV: Run times of the proposed algorithms on single board devices and `socLiveJournal1` for different K values. The allowed imbalance is $\beta = 0.1$, and the buffer capacity $B = 0.15 \times |\mathcal{H}|$ is used. The results on the $K = 32$ column are given in seconds, and for $K = 256, 2048$ and 16384, the results are normalized with respect to $K = 32$. That is the results in the last three columns show the decrease in the run time when K is increased from 32 to the corresponding value.

		Parts			
Device	Algorithm	32	256	2048	16384
Pi-1GB	MINMAX-N2P	142.6	1.43×	2.16×	3.73×
	REF	593.1	1.64×	2.93×	4.38×
	REF_RLX	650.0	1.50×	2.89×	4.16×
	REF_RLX_SV	532.3	1.53×	2.89×	4.73×
	MINMAX-L5	55.6	1.17×	1.30×	1.74×
Odroid-2GB	MINMAX-N2P	72.4	1.32×	1.81×	2.80×
	REF	335.2	1.41×	2.32×	3.62×
	REF_RLX	362.8	1.33×	2.27×	3.60×
	REF_RLX_SV	287.5	1.47×	2.37×	3.92×
	MINMAX-L5	36.1	1.11×	1.19×	1.40×
LattePanda-4GB	MINMAX-N2P	71.5	1.34×	1.89×	2.85×
	REF	308.3	1.45×	2.46×	3.73×
	REF_RLX	339.8	1.35×	2.43×	3.62×
	REF_RLX_SV	264.8	1.52×	2.57×	4.05×
	MINMAX-L5	30.4	1.11×	1.18×	1.33×

these experiments. The results are similar to the ones in the **Server**. Yet additionally, the slow-down values in the last three columns show that using much less memory, MINMAX-L5 stays more scalable compared to other algorithms when K is increased. Furthermore, the overhead of the refinement heuristics degrades the scaling behavior when they are added on top of the MINMAX-N2P. However, their negative impact tends to decrease when an SBC with more memory is used. This also shows the importance of streaming hypergraph algorithms with low-memory requirements in practice.

VI. Conclusion and Future Work

We focused on the streaming hypergraph partitioning problem. The problem has unique challenges compared to similar problem of streaming graph partitioning. We significantly improved the run time performance of a well-known streaming algorithm and proposed several variants on top of it to reduce the memory footprint and improve the partitioning quality. The experiments show that there is still room for improvement for these algorithms. As future work, we plan to devise more advanced techniques that can overcome the trade-off among the run time, memory requirements, and partitioning quality.

Acknowledgements

This work is funded by The Scientific and Technological Research Council of Turkey (TÜBITAK) under the grant number 117E249.

References

[1] S. Venkatraman, G. Rajaram, and K. Krithivasan, "Unimodular hypergraph for DNA sequencing: A polynomial time algorithm," *Proceedings of the National Academy of Sciences, India Section A: Physical Sciences*, nov 2018.

[2] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[3] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[4] O. Küçüktunç, K. Kaya, E. Saule, and U. V. Çatalyürek, "Fast recommendation on bibliographic networks," in *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2012, pp. 480–487.

[5] T. Menezes and C. Roth, "Semantic hypergraphs," *arXiv*, vol. cs.IR, no. 1908.10784, 2019.

[6] K. Skiker and M. Maouene, "The representation of semantic similarities between object concepts in the brain: a hypergraph-based model," *BMC Neuroscience*, vol. 15, no. Suppl 1, pp. P84–P84, Jul 2014.

[7] L. Sun, S. Ji, and J. Ye, "Hypergraph spectral learning for multi-label classification," in *Proc. of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '08. New York, NY, USA: ACM, 2008, p. 668–676.

[8] W. Zhang, Y. Chen, and D. Dai, "Akin: A streaming graph partitioning algorithm for distributed graph storage systems," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 183–192.

[9] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proc. of the 7th ACM Int. Conf. on Web Search and Data Mining*, ser. WSDM '14. New York, NY, USA: ACM, 2014, p. 333–342.

[10] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. of the 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, ser. KDD '12. New York, NY, USA: ACM, 2012, p. 1222–1230.

[11] A. Pacaci and M. T. Özsu, "Experimental analysis of streaming algorithms for graph partitioning," in *Proc. of the 2019 Int. Conf. on Management of Data*, ser. SIGMOD '19. New York, NY, USA: ACM, 2019, p. 1375–1392.

[12] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: An experimental study," *Proc. VLDB Endow.*, vol. 11, no. 11, p. 1590–1603, Jul. 2018.

[13] R. Andre, S. Schlag, and C. Schulz, "Memetic multilevel hypergraph partitioning," *arXiv*, vol. cs.DS, no. 1710.01968, 2017.

[14] Ü. Çatalyürek and C. Aykanat, *PaToH (Partitioning Tool for Hypergraphs)*. Boston, MA: Springer US, 2011, pp. 1479–1487.

[15] M. Deveci, K. Kaya, B. Uçar, and Ümit V. Çatalyürek, "Hypergraph partitioning for multiple communication cost metrics: Model and methods," *Journal of Parallel and Distributed Computing*, vol. 77, pp. 69 – 83, 2015.

[16] D. Alistarh, J. Iglesias, and M. Vojnovic, "Streaming min-max hypergraph partitioning," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 1900–1908.

[17] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. Chichester, U.K.: Wiley–Teubner, 1990.

[18] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, and K. Rothermel, "Hype: Massive hypergraph partitioning with neighborhood expansion," *arXiv*, vol. cs.DC, no. 1810.11319, 2018.

[19] L. Epple, "Partitioning Billionscale Hypergraphs," Master's thesis, Institute of Parallel and Distributed Systems, University of Stuttgart, Universitätsstraße 38 D–70569 Stuttgart, 2018.

[20] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970.

[21] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, 1997, pp. 21–29.

[22] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz, "k-way hypergraph partitioning via *n*-level recursive bisection," in *18th Workshop on Algorithm Eng. and Exp.*, 2016, pp. 53–67.

[23] M. F. Faraj and C. Schulz, "Buffered streaming graph partitioning," 2021, arXiv, cs.DS, 2102.09384.

[24] N. Jafari, O. Selvitopi, and C. Aykanat, "Fast shared-memory streaming multilevel graph partitioning," *Journal of Parallel and Distributed Computing*, vol. 147, p. 140–151, 2021.