Shared-Memory Scalable k-Core Maintenance on Dynamic Graphs and Hypergraphs

Kasimir Gabert
Georgia Institute of Technology
Atlanta, Georgia
Email: kasimir@gatech.edu

Ali Pinar
Sandia National Laboratories
Livermore, California
Email: apinar@sandia.gov

Ümit V. Çatalyürek
Georgia Institute of Technology
Atlanta, Georgia
Email: umit@gatech.edu

***Abstract*—Computing k-cores on graphs is an important graph mining target as it provides an efficient means of identifying a graph's dense and cohesive regions. Computing k-cores on hypergraphs has seen recent interest, as many datasets naturally produce hypergraphs. Maintaining k-cores as the underlying data changes is important as graphs are large, growing, and continuously modified. In many practical applications, the graph updates are bursty, both with periods of significant activity and periods of relative calm. Existing maintenance algorithms fail to handle large bursts, and prior parallel approaches on both graphs and hypergraphs fail to scale as available cores increase.**

We address these problems by presenting two parallel and scalable fully-dynamic batch algorithms for maintaining k-cores on both graphs and hypergraphs. Both algorithms take advantage of the connection between k-cores and h-indices. One algorithm is well suited for large batches and the other for small. We provide the first algorithms that experimentally demonstrate scalability as the number of threads increase while sustaining high change rates in graphs and hypergraphs.

I. Introduction

An important problem in graph analysis is finding locally dense regions in globally sparse graphs. In this work we consider the problem of finding k-cores [1], [2], which are maximal connected subgraphs with minimum degree at least k. This problem has seen significant attention due to its computational efficiency [2] and usefulness on a large number of problems [3]–[9]. We address computing k-core *values*, the largest k such that a vertex is in a k-core. Cores themselves can then be efficiently computed from the values [10].

Many practically important graphs today, from web data, social networks, and related fields, are both large and continuously changing. Finding dense regions as quickly as possible after a change is important, for example to quickly initiate a response to rapidly spreading false information about vaccines or to urgently address new pandemic super-spreading events. We focus on *maintaining* k-core values over a dynamic graph with batches containing both edge insertions and deletions. The maintained k-core values can then be directly queried. The goal of maintenance algorithms is to drive down the *latency* of a query, or the algorithm runtime for processing a single edge change. This typically comes at a cost of *throughput*, or the number of edge changes processed by the total runtime. A sequential, single-edge maintenance algorithm typically has both a low latency and throughput, whereas re-computing from scratch will have both a high latency and throughput.

There are two main approaches for maintaining cores values on dynamic graphs, traversal [11], [12] and order [13]. When bursts of large activity come in, they *cannot keep up* with the data stream. There has been a recent focus on parallel batch algorithms to address this problem [14]–[17]. Such algorithms operate on a *batch* of edge changes at once, enabling more parallelism at the cost of latency for small batches. They provide a middle ground between computing from scratch and single-edge maintenance algorithms. There are three known parallel batch algorithms for cores, all based on the idea of finding *independent* edges and processing them with traditional, sequential techniques [18]–[20]. Unfortunately, this approach does not show much scalability as additional processors are added (e.g., see Fig. 11 in [19], Fig. 6 in [20].) Given the increasing rate of data from social and web applications, there is a strong need for completely new approaches that can scale as the number of threads grow. We present two new algorithms to address this gap.

Our algorithms use the connection between h-indices [21] and k-cores, first identified by Lu et. al [22], which has been used as a local, distributed algorithm [23], [24] for computing from scratch. In this algorithm, each vertex has a local value that is initialized to inifinity. At each step (either synchronously or asynchronously), the algorithm computes the h-index of its neighbors' values. This process will converge within the number of degree levels of the graph [24]. The advantage of this process is that, after initialization, each vertex can operate independently. Building on this, we provide two scalable maintenance algorithms that maintain k-core values on batches of graph insertions and deletions. The first, mod, is based on modifying local values and then "continuing" convergence. The challenge with this approach is to increment local values as little as possible. The second, set, follows the h-index iterations, but keeps track of each edge insertion made to the graph. The increases happen locally, based on the given set of insertions and together with convergence. The mod algorithm provides consistent improvements over re-computing from scratch for large batches, and the set algorithm provides improvements for small batches.

In many cases, real-world data is naturally modeled as hypergraphs instead of graphs [25]. For example, consider purchasing relationships that consist of users and items. Hyperedges naturally model multiple users purchasing the

same item. As another example, people may be vertices, and hyperedges would indicate that they were close enough to each other to spread diseases during a time period. Here, a hypergraph k-core would represent a group of people that are likely to internally spread disease. We want to ensure that our approaches both apply to hypergraphs and have scalability when running on hypergraphs. For dynamic hypergraphs, there are two main models. One model treats each hyperedge as a single, immutable unit, and operates on a stream of hyperedge changes. This is the approach taken by [26]. However, this model cannot capture the dynamic nature of many existing hypergraphs. For example, consider a hypergraph consisting of users and topics that the user likes. In real networks, both new topics are created *and* users' preferences change. To model this behavior, each hyperedge itself can also have internal changes. We address this more general model.

We provide the following contributions.

- We provide two shared-memory parallel batch algorithms, mod and set, that maintain k-core decompositions using the connection between cores and h-indices on both graphs and hypergraphs
- We introduce strategies to handle parallelism in the more challenging case of changing hyperedge *pins*
- We demonstrate our algorithms' scalability empirically

The remainder of this paper is structured as follows. In § II we provide background, including notation used, the problem we address, and related work. In § III we present the static algorithms for k-core decompositions. In § IV we introduce our parallel batch algorithms. In § V we present our experiments and results, and finally in § VI we conclude.

II. Background

A. Notation and Preliminaries

Here we describe the notation used throughout the paper. We are concerned with both graphs and hypergraphs. Let $G = (V, E)$ be a graph, where V is a set of vertices and E a set of edges. We focus on simple, undirected graphs, so each edge is a set containing two distinct vertices. A hypergraph $H = (V, E)$ is a generalization of graphs, where V is a set of vertices and E is a set of hyperedges. In hypergraphs, a hyperedge is a subset of vertices, $e \in E$ such that $e \subseteq V$. This means that a hyperedge may contain one or more participating vertices, compared with exactly two as in a graph. We call the relationship between a vertex and a hyperedge a *pin*.

The set of neighbors of a vertex in both cases is given by $\Gamma(v) = \{u \in V : \exists e \in E, \{u, v\} \subseteq e\}$. The *degree* of a vertex is the number of neighbors it has, $\deg(v) = |\Gamma(v)|$. The maximum degree in a graph or hypergraph is $\Delta(G)$ ($\Delta(H)$, resp.). Induced subgraphs in hypergraphs cannot split hyperedges, and so every pin in a hyperedge remains. So, when an induced subgraph is taken (for example, in a k-core), if *any* vertex in a hyperedge has a degree less than k then the hyperedge is effectively "peeled" from the hypergraph.

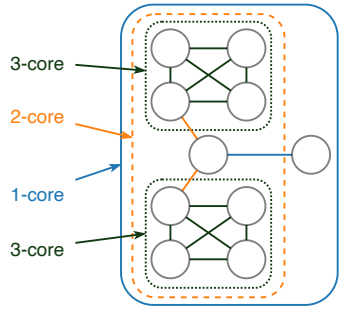

Fig. 1. An example graph and its cores.

B. Computing k-Cores on Static Graphs and Hypergraphs

Cores were introduced concurrently and independently by [1] and [2]. While the notation differed, in both cases a core was defined as the following.

Definition 1: Given a graph G and a parameter k, a k-core of G is a maximal connected subgraph with minimum degree at least k.

However, over time implementations and approaches dropped the connectivity constraint [27]. In particular, the focus has been on computing *k-core values*, or coreness, the process of which is called a k-core decomposition. An example graph with the k-core values shown is given in Figure 1.

Definition 2: Given a graph G and vertex v, the k-core value of v, denoted $\kappa[v]$, is the value k such that v is in a k-core but not in a $(k+1)$-core.

Using disjoint-set forests, cores can be maintained from k-core values quickly [10]. In the remainder, we consider k-core values when we refer to cores.

The most straightforward way of computing cores is through "peeling." [2] In this approach, vertices are iteratively removed—typically by keeping track of whether they are active or not, not by modifying the graph—if their degree is less than k. Any vertex that is removed has its k-core value assigned as k. When all vertices are removed, the process stops. If vertices are organized in buckets based on their degree, this process can be done in $O(|V|+|E|)$ [2]. Variants of this approach are used for processing in parallel [28]–[30]. Parallelization in these approaches largely occurs by taking advantage of parallelism within levels.

A separate approach follows the connection between h-indices [21] and k-cores [22]. This strategy was first developed as a distributed algorithm [23] and later shown to be close to state-of-the-art in a shared-memory setting [24]. The idea of this approach, presented in detail in Section III, is to iteratively update a *local value* associated with each vertex. The local value on vertices is initialized high, and in each iteration the h-index of each vertices' neighbors' local values is computed. The process terminates when changes are made. The advantage of this approach is that each vertex can operate independently, asynchronously.

In hypergraphs a core is defined exactly the same way: it is a maximal connected subgraph with minimum degree at least

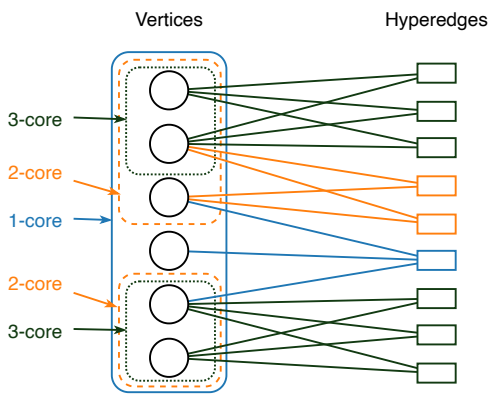

Fig. 2. An example hypergraph and its cores.

k. Similarly, a k-core value for a vertex in a hypergraph is the largest value k such that a vertex is in a k-core but not in any $(k+1)$-core. An example hypergraph and its k-core decomposition is shown in Figure 2.

In hypergraphs, the standard peeling approach works as well, including in parallel [25]. In Section III we develop an h-index static computation approach for hypergraphs.

C. Dynamic Graph and Hypergraph Models

We address graphs and hypergraphs that change over time, which are called dynamic. Dynamic graphs and hypergraphs can be viewed as an infinite sequence of *changes*, where each change is either an edge insertion or deletion. A *batch* is an interval on the infinite stream, meaning a collection of changes close in time that are processed together. Large batches naturally occur when the arrival of graph changes is faster than the latency of processing the prior batch. Our goal is to address large batches in a scalable, parallel manner.

For hypergraphs, the stream can be either at the level of *hyperedge changes* or *pin changes*. In either case, the notion of a k-core still requires inducing a full hyperedge, separating this problem from that of bipartite cores [31]. While [26] addresses hypergraph streams with hyperedge changes, we address the more general model of *pin changes*. It is straightforward to simulate hyperedge changes by setting batch boundaries at full hyperedges.

D. Related Work

There are two main approaches for *maintaining* cores on dynamic graphs, both of which are sequential. The first is the traversal algorithm [11], [12]. Given a new edge, it performs a depth-first graph traversal from the endpoint with the lowest coreness. The traversal remains within a *subcore*, which is a connected region with the same coreness value. If it comes across a vertex unable to increase, it stops searching that path. The second is the order algorithm [13]. A valid decomposition order is an ordering of vertices that could arise from peeling. On an edge insertion, this algorithm corrects the *order* by moving vertices that change coreness, keeping their relative prior order, to the beginning of the next core. The sequential order based approach was extended in the *truss* setting to handle batches [32].

There are other parallel core maintenance approaches [18]–[20]. These approaches are traversal-based but identify opportunities for parallelism by using matchings among edge sets or independent tree structures. Unfortunately, on real-world graphs such opportunities are limited, and these approaches suffer scalability problems as the number of cores grow. Concurrent to our work, [33] considers h-index based methods for dynamic core maintenance. Unfortunately, this work is not parallel and the approach is not competitive against state-of-the-art sequential methods [13], [32].

Handling k-cores in dynamic hypergraphs has seen recent attention [26]. The authors identified the need and demonstrated the importance of solving hypergraph k-cores. However, [26] presents only an approximate sequential solution, again based on peeling.

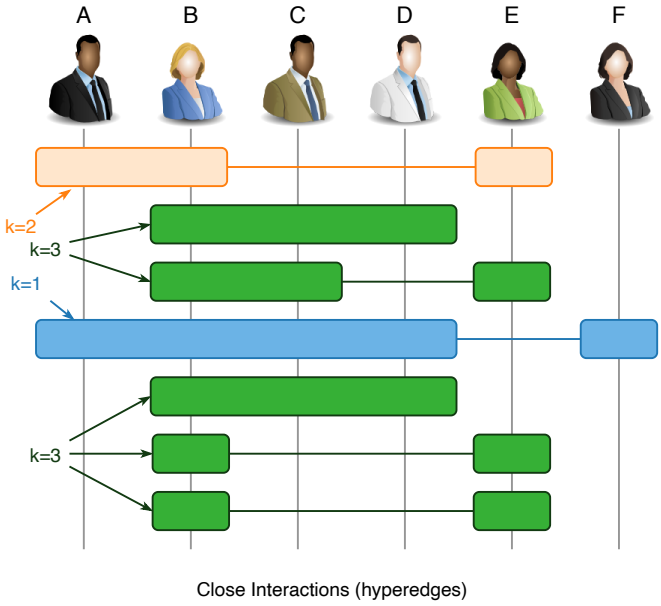

Fig. 3. Co-occurrence hypergraphs may be useful for identifying groups susceptible to pandemics. Boxes are hyperedges. Dark green indicates all members have $\kappa = 3$, orange indicates $\kappa = 2$, and blue indicates $\kappa = 1$. A graph representation gives F a high degree and a high core value, even though it likely has low exposure. In a co-occurrence hypergraph, F would have a core value of 1 whereas B, C, D, and E would have a value of 3, capturing that they have close, on-going, and intimate interactions.

E. Hypergraph k-Cores to Address Pandemics

Computing hypergraph k-cores may be beneficial for identifying groups of individuals to monitor or address for spreading of diseases, for example to assist in addressing the COVID-19 pandemic. Consider Figure 3. Here, a hyperedge (a rectangle) is created between any individuals that have close contact within a time period. In the figure, person A is in a meeting with person B and E, and so they have close contact and the first line is a hyperedge connecting them. We call such a hypergraph a *co-occurrence* hypergraph. A k-core in this hypergraph could then identify individuals with significant

Algorithm 1: Asynchronous local algorithm to compute κ [22].

```
Data: graph G = (V, E)
∀v ∈ V, τ[v] ← deg(v)
repeat
    for v ∈ V do in parallel
        L ← ⟨⟩
        for w ∈ Γ[v] do
            L ← L.extend(τ[w])
        τ[v] ← H-INDEX(L)
until τ no longer changes, converging to κ
return τ
```

close contact with others, in a way that captures deeper relationships than simply looking at a graph perspective. For example, for person F, a graph representation would show them connected with every other individual, with both a high degree and a high core value. However, they may have been present only at one meeting, and so would be less likely to transmit the disease.

The hypergraph k-core instead would place B, C, D, and E in a 3-core together, as they are each part of 3 hyperedges where all members are in a 3 hyperedge as well.

This paper provides computational methods to address hypergraph k-cores in parallel, and is motivated by the possible application to address pandemics; however, the usefulness and applicability for pandemics remains an open question.

III. Static h-index Algorithms

In this section we describe h-index based approaches to compute cores, which are the foundation of our maintenance algorithms.

Definition 3: Let $S = \langle s_1, s_2, \ldots, s_n \rangle$ be a tuple of values, with $s_i \in \mathbb{Z}$ for $i \in [1, n]$. The h-index of S is the largest value h such that $s_i \geq h$ for $i \in H$, where $|H| = h$ and $H \subseteq [1, n]$.

In this section we describe the prior foundational algorithm, highlight the challenges in converting them to dynamic *maintenance* algorithms, and describe our final algorithms.

A. h-index Coreness Computation

To understand our algorithms, it is first useful to understand the asynchronous local h-index algorithm proposed by [22]. This algorithm is re-presented here in Algorithm 1 for completeness. The local variable τ is initialized to the degree of each vertex in H. Then, each vertex is iteratively processed and τ is updated based on the h-index of neighbor's τ values. When no further changes occur $\tau = \kappa$.

This was a breakthrough for computing k-cores as it is well suited for parallelization: each vertex can update its own local values, given access to its neighbors' local values, in parallel. In the synchronous version each vertex considers its neighbor's values from the previous time step. In the asynchronous version, each vertex takes the latest available value for each neighbor.

B. Key Problem: How To Reinitialize

Consider Algorithm 1. There are many different possible initializations for τ. In fact, τ can be initialized to any value equal to or larger than κ—the degree is chosen simply because it is an upper bound on κ. It would be possible to use ∞, or $\kappa[v] + 1$, as shown by the convergence of the asynchronous version [22]. If τ is initialized to ∞, then after only one iteration τ will recover the degree initialization. The constraint is that τ cannot be initialized *too low*, that is, if some vertex d, $\tau[d]$ is initialized below $\kappa[d]$, τ may fail to converge to κ.

It may seem like simply re-using the prior output, incrementing the edge that changed, and continuing the computation will work. Unfortunately, this is not the case.

Lemma 1: If a τ value is below κ, then Algorithm 1 may never converge to κ.

Proof: Consider P_n, a path of length n. Note that $\forall v \in P_n$, $\kappa[v] = 1$. Let the endpoints be v_1 and v_n, such that $\deg(v_1) = \deg(v_n) = 1$. Suppose $\{v_1, v_n\}$ is an inserted edge. Then, suppose that *any two consecutive* vertices v_j, v_k have $\tau[v_j] = \tau[v_k] = 1$. Now, running Algorithm 1 to convergence will result in $\kappa[v_i] = 1 \forall i \in [1, n]$, which is incorrect. ■

As tempting as it is, given Lemma 1 a simple memoization algorithm *will not work.*

Furthermore, if only part of the hypergraph is initialized above κ, then the vertices already at κ do not need to recompute each iteration allowing the problem to remain local to part of the graph. The key problem is that we need to know the smallest set to increment, and increment those vertices, before we can run to convergence.

Building on this observation we present two dynamic algorithms along with several variants. In the first algorithm, we initialize τ values once for each batch, trying to increase τ as little as possible while ensuring convergence. We keep track of which vertices in H are already at convergence and do not perform any computation on them, allowing for batches to run in $o(|H|)$. In the second, we combine initialization and convergence, allowing for initialization to spread concurrently with convergence. We keep track of changes to H and propagate them through the graph, increasing τ values for neighbors that have not seen the change yet and would be affected by it.

C. Extension To Hypergraphs

We extend Algorithm 1 to hypergraphs. Our extension is shown in Algorithm 2. In particular, we build the neighbor list L using the *minimum value* after excluding the source vertex. This allows for coreness values to remain correct, as any vertex with too low of a coreness value will cause the entire hyperedge to stop contributing.

Theorem 1: Algorithm 2 will correctly return the coreness values κ.

Proof: The proof follows Thm. 1 in [22]. Induction is used to bound the h-index sequence both from above and below by coreness, and convergence is shown as progress is made every iteration. This results in exactly the coreness. ■

In the remainder, when we reference graphs, we mean both graphs and hypergraphs. Graphs can be viewed as a special

Algorithm 2: hhcLocal, extending [22] to hypergraphs.

Data: hypergraph $H = (V, E)$
Input: optional τ initialization, frontier A
1 **if** τ *is not given* **then** $\forall v \in V,\ \tau[v] \leftarrow \deg(v)$
2 **if** A *is not given* **then** $A \leftarrow V$
3 **repeat**
4 **for** $v \in A$ **do in parallel**
5 $A' \leftarrow \emptyset$
6 $L \leftarrow \langle\rangle$
7 **for** $e \in E : v \in e$ **do**
8 $L \leftarrow L.\mathsf{extend}\left(\min_{w \in e, w \neq v} \tau[w]\right)$
9 $\tau[v] \leftarrow \text{H-INDEX}(L)$
10 **if** $\tau[v]$ *changed* **then** $A' \leftarrow A' \cup \{v\} \cup \Gamma(v)$
11 $A \leftarrow A'$
12 **until** $A = \emptyset$
13 **return** τ

Algorithm 3: A simple variant of mod that operates only on a single *hyperedge* change and maintains κ. hhcLocal extends Algorithm 2 with active vertices that, on a local change, make neighbors active and otherwise go dormant.

Data: hypergraph $H = (V, E)$, local values τ
Input: single hyperedge change $y = \{d_1, \ldots, d_s\}$, c
▷ update the hypergraph
1 **for** $d_i \in y$ **do**
2 **if** $c = +$ **then** $H[y, d_i] \leftarrow 1$
3 **else** $H[y, d_i] \leftarrow 0$
▷ maps are zero-initialized
4 $R \leftarrow \{\}$ ▷ map τ values to num. resolved ins.
5 $D \leftarrow \{\}$ ▷ map τ values to num. of deletions
▷ find a vertex with min τ value
6 $d_m \leftarrow \arg\min_{d_i \in y} \tau[d_i]$
7 **if** $c = +$ **then** $R[\tau[d_m]] \leftarrow 1$
8 **else** $D[\tau[d_m]] \leftarrow 1$
9 $A \leftarrow \emptyset$ ▷ active vertices to process
10 **for** $d \in V$ **do in parallel**
 ▷ apply the resolved count
11 $\tau[d] \leftarrow \tau[d] + R[\tau[d]]$
 ▷ mark changed vertices as active
12 **if** $R[\tau[d]] \geq 0$ *or* $D[\tau[d]] > 0$ **then**
13 $A \leftarrow A \cup \{d\}$
14 hhcLocal(A, τ), using vertices A and initialized to τ

case of hypergraphs, where each hyperedge has exactly two endpoints. This is easy to handle in an implementation. When we explicitly mention hypergraphs, we address problems that do not apply to graphs.

IV. h-Index Based Core Maintenance

We now present two dynamic algorithms, both of which build on the local h-index algorithm presented in [22]: the first involves incrementing τ across the graph, attempting to increment as few times as possible. Convergence then occurs similarly to in Algorithm 1. The second involves combining initialization and convergence. We keep track of each change to H and run a modification of the h-index algorithm. As the τ computation iterates, updates are propagated outwards, causing vertices to increment their own τ as appropriate. We refer to this algorithm as set, with setmb additionally optimized with mini-batches.

These two algorithms come with different tradeoffs. The re-initialization is useful to provide consistent latencies lower than a static recomputation, but on many graphs fails to capture really low latencies. The combined initialization and convergence provides a different advantage: it can have significant latency improvements, reaching over $10^4\times$ static computation on real-world graph instances, but with high variability: with some batches there are far more iterations as increments propagate, and there is a computational overhead for checking whether an update has been processed.

Our algorithms are presented with *callback* functions, which are designed to be run when a vertex is inserted into or removed from a hyperedge. The change value c is the direction, indicating either an insertion $(+)$ or deletion $(-)$.

A. Re-initialization Based Algorithms

In this section we describe our algorithms that are based on initializing τ and then running Algorithm 2, which result in fewer iterations than static computation in many instances. This problem was shown to be unbounded in the locally persistent model [32], and so we cannot expect to do better than re-running from scratch in all cases. This result also means that we need to ensure our worst-case complexity matches computing from scratch, but we cannot expect to have a better complexity.

Differing significantly from state-of-the-art core and truss maintenance algorithms [13], [32], we do not maintain an order of the vertices. Instead, we make larger increments than necessary and let the parallel local h-index approach resolve them.

We begin by introducing a simple, non-batch variant of our algorithm, presented in Algorithm 3. Our batch approach naturally extends from there to the full mod algorithm shown in Algorithm 4.

We know that for a single edge insertion in H, the κ value will only change for the involved minimum κ valued vertex (by [11]). However, we will be performing multiple updates in a batch. This means that the τ values are, at this point in time, potentially *smaller* than κ. So, we cannot identify which vertex has the minimum κ and instead increment all κ values that participate in the modified hyperedge. In lines 5-12 we deal with the situation where the subcore, or part of it, has *moved* after some other insertion or deletion. We conservatively increment in as many potential parallel cases as possible, such as when the subcore is broken, part of it

Algorithm 4: The mod algorithm.

Data: hypergraph $H = (V, E)$, local values τ

▷ Insertion callback

1 **Function** `f-mod`$(e_a,\ v_b,\ c)$**:**
2 **if** $\exists v_i \in e_a$ *s.t.* $\tau[v_b] > \tau[v_i]$ **then return**
3 **if** $c = +$ **then** $I[\tau[v_b]] \leftarrow I[\tau[v_b]] + 1$
4 **else** $D[\tau[v_b]] \leftarrow D[\tau[v_b]] + 1$

Input: batch edge set B

1 $I \leftarrow \{\}$ ▷ map τ values to num. of insertions
2 $D \leftarrow \{\}$ ▷ map τ values to num. of deletions
3 $R \leftarrow \{\}$ ▷ map τ values to num. resolved ins.
4 MaintainH(`f-mod`, B)
▷ increment as many possible subcore levels and values that could arise from concurrent execution
5 **for** $k \in$ keys(I) **do in parallel**
▷ increment as if some subcore at $\kappa = k$ decreased and merged with another
6 **for** $t = k - D[k]$ *to* $k - 1$ **do**
7 $R[t] \leftarrow R[t] + I[k]$
8 $R[k] \leftarrow R[k] + I[t]$
9 $R[k] \leftarrow I[k]$ ▷ increment if stayed at level
▷ increment as if some subcore at $\kappa = k$ increased and merged with another
10 **for** $t = k + 1$ *to* $k + I[k]$ **do**
11 $R[t] \leftarrow R[t] + k + I[k] - t$
12 $R[k] \leftarrow R[k] + I[t]$
13 $A \leftarrow \emptyset$ ▷ active vertices to process
14 **for** $d \in V$ **do in parallel**
▷ apply the resolved counts
15 $\tau[d] \leftarrow \tau[d] + R[\tau[d]]$
▷ mark changed vertices as active
16 **if** $R[\tau[d]] \geq 0$ *or* $D[\tau[d]] > 0$ **then**
17 $A \leftarrow A \cup \{d\}$
18 hhcLocal(A, τ), using vertices A and initialized to τ

merged with smaller subcores, and more.

If a κ value changes for a vertex, two properties must hold: that vertex must have a specific starting κ value and they must be connected to the hyperedge change (again by [11]). The mod algorithm exclusively uses the κ *value* and ignores the connectivity.

Additionally, we perform an important optimization: the minimums on hyperedges are cached. It is possible to only store a single minimum, as this will not have a negative impact on the convergence or correctness.

B. Processing in Parallel with Pin Changes

The problem becomes more complicated for re-initialization algorithms as we deal with a stream of *pin changes* instead of hyperedge changes. For each pin deletion, this can result in both *an increase* and *a decrease* in κ values for different nodes. The hyperedge that *loses* the pin may in fact *gain* a new κ value for all other vertices. This will happen if the pin is exactly the lowest valued vertex in the hyperedge.

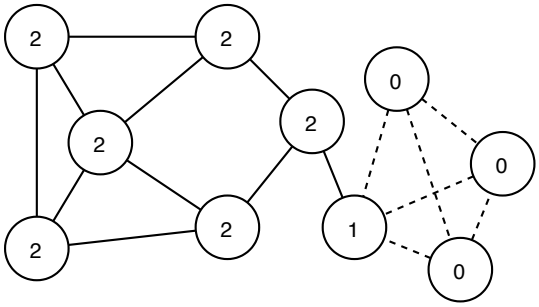

Fig. 4. A notional example showing why increments need to be sufficiently high. The dotted lines are new edges and the solid lines are existing edges. Even though edges are only added to the $\kappa = 1$ vertex, after the batch is processed all vertices need to increase to $\kappa = 3$.

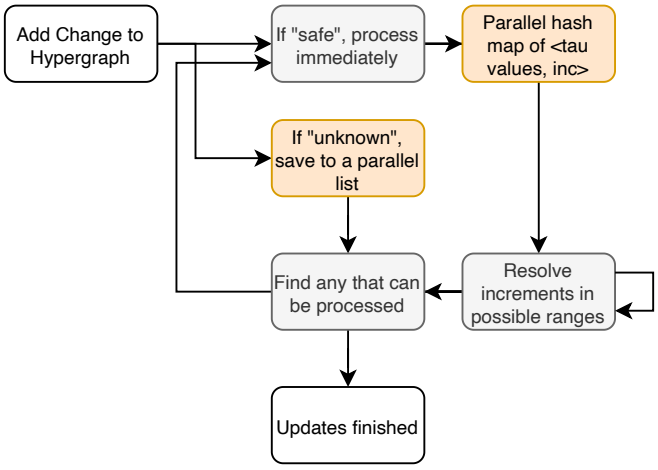

Fig. 5. The process of performing increments based on a hyperedge change. Safe means the change can be concretely resolved, that is the *possible positions* of the pin in the hyperedge *range* is known.

Additionally, the vertex that the pin connects to may drop its k-core value. With insertions, both an *increase* in the core value for the vertex with the pin and a *decrease* in the core value for every other vertex in the hyperedge can occur.

This complicates the decision for resolving increments and increases the number of increments and decrements that have to happen. An example showing the complication is given in Figure 4. This results in significantly more book-keeping for the process in lines 5-12. The full, parallel process in the implemented system can be seen in Figure 5. Here, given a change to the hypergraph, we need to keep track of whether the change can be processed immediately or needs to be resolved later. This process iterates until all changes have been resolved.

Concretely, we break the problem down into four cases. Here, we present the cases dealing with deletions. For insertions, the deletions and insertion changes are swapped.

- Case 1: the hyperedge no longer exists. Find the min range within the previous hyperedge vertices' ranges and decrement this range.
- Case 2: the min range of the deleted pins can be smaller than the prior min range. Decrement within the min range

Algorithm 5: The set algorithm, which mixes incrementing τ and converging τ. The id function resets each batch and increments on distinct e_a inputs.

```
Data: hypergraph H = (V, E), local values τ
▷ Insertion callback
1  Function f-set(e_a, v_b, c):
2      A[v_b] ← 2                                   ▷ maximum time-to-live
3      if c = + then U[v_b] ← U[v_b] ∪ {id(e_a)}
Input: batch edge set B
1  A, U, P ← {}, {}, {}
2  MaintainH(f-set, B)
3  repeat
4      c ← false
5      for x ∈ V do in parallel
6          if A[x] = 0 then continue
7          U_x ← U[x]                               ▷ U[x] may change
8          L ← ∅                                    ▷ list of neighbor values
9          for e ∈ E : x ∈ e do
10             m ← ∞
11             for n ∈ e : n ≠ x do
                   ▷ Consider un- or processed hypergraph
                     changes for n
12                 t ← τ[n] + |U[n] ∪ (U_x \ P[n])|
13                 if t < m then m ← t
14             L ← L ∪ m
15         τ' ← H-INDEX(L)
           ▷ Determine if our τ changed
16         if τ' ≠ τ[x] then
               ▷ Update the neighbors
17             for n ∈ e such that x ∈ e and n ≠ x do
18                 U[n] ← U[n] ∪ (U_x \ P[n])
19                 A[n] ← 2
20             τ[x] ← τ'
21             A[x] ← 2
22         else
23             A[x] ← A[x] − 1
24         P[x] ← P[x] ∪ U_x
25         U[x] ← U[x] \ U_x
26         c ← true
27 until c = false
```

of the deleted pins and increment within the min range of the existing hyperedge.

- Case 3: no deleted pins had vertices within the min range. This is marked *unknown* and will be revisited.
- Case 4: the deleted pins min range may overlap the prior min range. Decrement the middle range.

Note that everywhere we use *ranges* instead of concrete κ values, as they will be processed in a loop until convergence.

C. Mixing Initialization and Convergence

In this section we describe our set algorithm, which mixes initialization of τ and convergence concurrently. This algorithm deviates internally from Algorithm 2. Each hyperedge change is recorded and its history is remembered for the course of the batch. During each h-index computation, the neighbors of a vertex are considered but instead of reading τ directly, τ is read and each potential graph change is applied. If this will result in a change to that vertex, then the change is propagated. Otherwise, the change stops.

This allows for a small part of the graph to be visited, but in the worst case will result in additional iterations and additional work, as each change may slowly propagate to the whole graph *before* regular τ convergence, increasing the number of iterations by the diameter of the graph. We present this approach in Algorithm 5. In the callback, each vertex is marked as having the modification "unprocessed." Then, lines 7-25 contain the mixed convergence with increments. First each neighbor is considered. Instead of setting the new τ value based on the h-index of the neighbor's τ values, it sets τ based on the h-index the neighbor's currently unprocessed modifications, plus the modifications neither the vertex nor the neighbor has processed. Differing from the re-initialization algorithms, vertices stay active for one extra iteration after convergence. This allows for convergence to occur when $U[x]$ is updated while x is currently processing and covers cases where a propagation changes from incrementing to converging.

While the updates are propagating the memory may continue to grow across the hypergraph, as more U and P entries are being set. There are a variety of ways to realize this in implementation, some of which come with more expensive memory requirements. We considered using boolean vectors, dynamic bit vectors, and fixed-size pre-allocated bit vectors coupled with mini-batches. Our experiments are all performed with mini-batches (setmb), with batch sizes of 64. Mini-batches stopped iterating when P became empty for all vertices with a final batch iteration to converge τ.

The correctness of this algorithm comes from the observation that if the frontier of an update will not cause any further increase in τ values, then it is not necessary to further propagate the update.

D. Runtime and Memory Complexity

Prior work has shown that the worst-case runtime complexity of a dynamic core maintenance algorithm is the same as re-computing from scratch [32]. As such, our h-index method has the same runtime as recomputing from scratch, which is formally studied in [24] and is the same as peeling, $O(n+m)$. We stress that the *variance* between graphs and even batches is not well understood and remains an open question. For our memory complexity we store κ values for each vertex and, with setmb, we store set lists as single 64-bit integers. Finally, we store additional data for each batch bound by the number of edges in a batch. So, the memory complexity is $O(n+m)$.

V. Experiments and Results

In this section we perform experiments to empirically demonstrate the scalability of our two algorithms, mod and setmb, as graphs become bursty.

TABLE I
GRAPHS USED FOR OUR EXPERIMENTS.

Name	Vertices	Edges
OrkutLinks	3.07 M	240 M
LiveJ	3.99 M	37.4 M
Pokec	1.63 M	22.3 M
Patents	3.77 M	16.5 M
DBLP	1.82 M	8.34 M
WikiTalk	2.39 M	4.66 M
Google	0.88 M	4.32 M
YouTube	3.22 M	9.38 M

TABLE II
HYPERGRAPHS USED FOR OUR EXPERIMENTS.

Name	Vertices	Hyperedges	Pins
OrkutGroup	2.8 M	8.7 M	327 M
WebTrackers	27 M	13 M	141 M
LiveJGroup	3.2 M	7.5 M	11.M

We implemented our algorithms in C++17 and compiled with GCC 10.2.0 and `-O3`. We use Intel TBB to provide parallel hash maps to store the graph (with the edge lists stored as vectors) and both TBB and OpenMP to parallelize execution. We ran on Intel Xeon E5-2683 v4 CPUs with 512 GB RAM and dual sockets and used `numactl` to set thread counts. We checked correctness against Ligra [25].

Unfortunately we were not able to compare against alternative parallel approaches due to a lack of available implementations [18]–[20]. We note that against the *reported runtimes*, we are around $4\times$ faster, however we are using different hardware and systems. We stress that none of the prior systems have demonstrated shared-memory scalability.

While we do not test vertex changes directly, in all of our experiments vertices are deleted when their degree drops to zero and created when their degree increases from zero. Our implementation does not require vertex labels to be contiguous, and so it supports hypersparse graphs where many implied vertices have a degree of zero. We use 64-bit unsigned integers to store vertex IDs.

A. Datasets

We chose a variety of graphs from domains representing social networks, citation networks, and web data. The graphs are shown in Table I and the hypergraphs are shown in Table II. The graph datasets were download from SNAP [34] and the hypergraphs from KONECT [35]. As these graphs are not ordered temporally and do not have deletions, we simulated edge insertions and deletions as follows. First, we uniformly randomly select pins or edges and remove them from the graph. We then insert them back again, and time both the removal and insert. To test mixed insertion and removal times, we set our removal and insert size to be $3/2$ the full batch size. The number of edges or pins in the graph is a major factor in runtime, and the maximum coreness and complexity of core hierarchy additionally impact runtime.

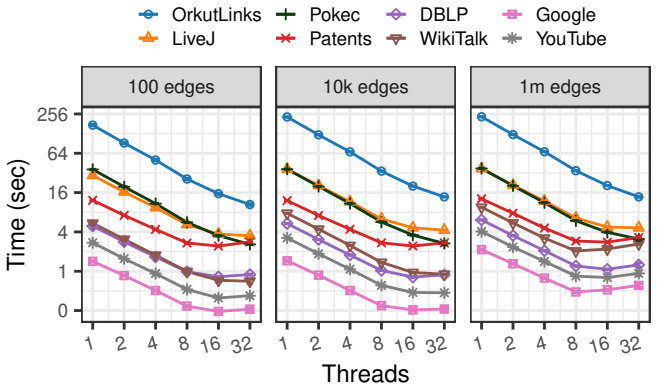

Fig. 6. The mod algorithm's scalability for processing edge insertions at different batch sizes. Insertion-only edge batches with mod.

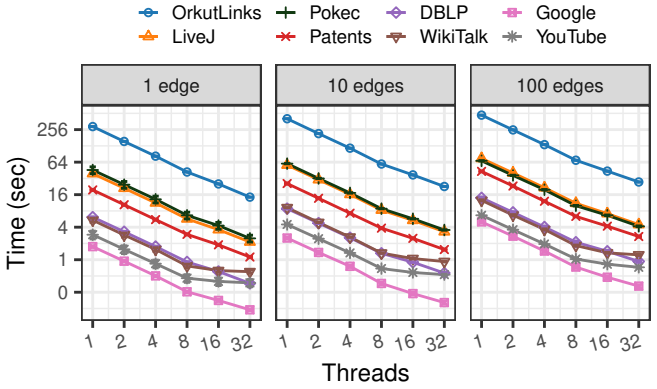

Fig. 7. Insertion-only edge batches with setmb.

Future work includes characterizing graphs and batches to determine runtime behavior.

In each experiment, batches were removed and then re-inserted 50 times. Error bars show one standard deviation from the mean. We chose batch sizes based on expected real-world ranges for each algorithm. We do not show the results for setmb for hypergraphs, as it will require caching values on hyperedges to be competitive against mod.

B. Insertion Scalability

First, we measure the scalability for handling insertions for both mod and setmb. The results can be seen in Figure 6–7. In both cases, as the cores increase the total runtime decreases. Choosing to run setmb for very small batches and mod for larger batches would be effective for a wide range of insertions. Additionally, setmb has a very high variance on the larger graphs. While it provides the smallest runtimes on small batches, it also has high outliers that significantly increase the average. Future work can address reducing the variance and the maximum cost. For some graphs, moving from 16 to 32 threads decreases performance slightly.

Note that the total runtime for small batches with mod is only slightly less than large batches, although we show results in log log plots. In many cases moving from 100 to 1 million results in only around a $1.5\times$ increase in runtime. This is

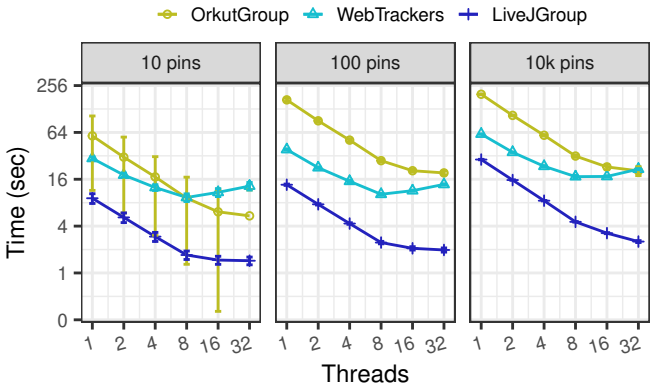

Fig. 8. Insertion-only pin batches with mod.

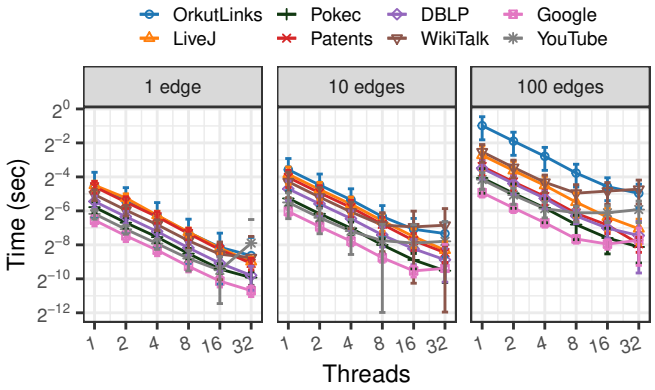

Fig. 10. Deletion-only edge batches with setmb.

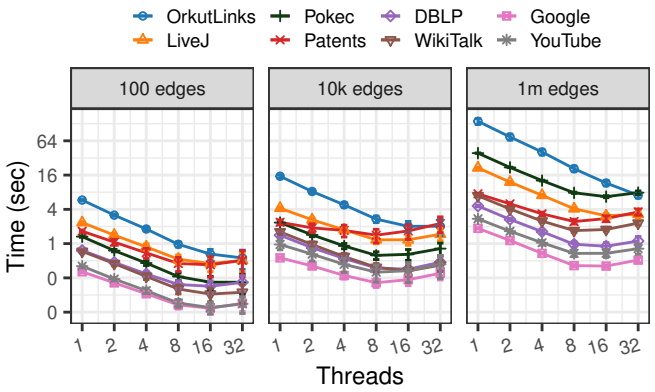

Fig. 9. Deletion-only edge batches with mod.

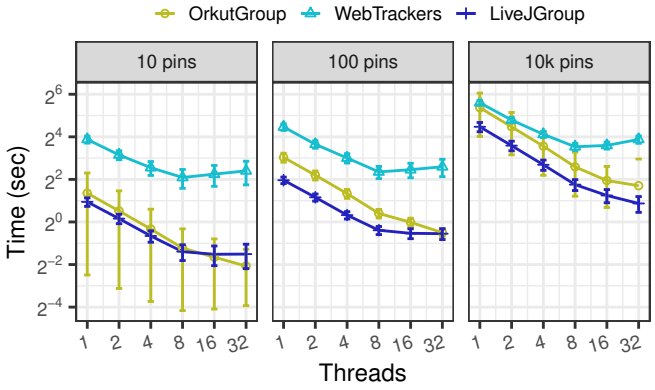

Fig. 11. Deletion-only pin batches with mod.

due to incrementing some edges that have a small coreness value, causing large parts of the graph to be impacted. This reduces the variance between batches but introduces additional work. Future work could find how to identify changes that result in mod incrementing many values and then process those separately with setmb, forming a hybrid approach that improves the overall runtime while retaining scalability and reducing long tails.

In Figure 8 we show the scalability for running mod on the hypergraphs. For WebTrackers, the performance decreases in all cases after 8 threads, however for OrkutGroup and LiveJGroup the performance continues to decrease after the NUMA boundary. With up to 8 threads on those graphs the scalability is close to linear.

C. Deletion Scalability

We measure the scalability for performing just edge deletions with both mod and setmb. The results can be seen in Figures 9–11. For both mod and setmb, the performance tends to decrease as the batch sizes get larger and increase as the number of threads increasing, showing that this approach similarly scales on deletions. For setmb, even with large batches the latency for deletions is low.

When deleting pins in hypergraphs, the variance can become large. For example, see OrkutGroup with 10k pins. Both the insertion and deletion variance for a small number of pin changes is high.

D. Mixed Insertions and Deletions

One advantage of our proposed algorithms is that they do not require pre-processing on the stream to separate deletions and insertions. Instead, insertions and deletions can be handled concurrently. Our results show similar scalability for mixed batches as with insertions. For example, in Figure 12 we show the mixed batches for mod. Note the similarity to Figure 6.

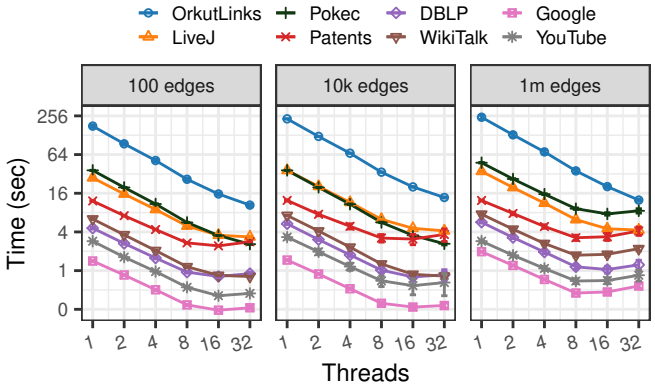

Fig. 12. Mixed batches with mod.

VI. Conclusion

We present two scalable, parallel batch k-core maintenance algorithms that operate on fully dynamic graph and hypergraph streams. These algorithms differ from prior approaches in that they build on the connection between h-indices and k-cores. We address two models for dynamic hypergraphs, one with hyperedge changes and one with pin changes. Our implementations empirically scale well on shared-memory systems, exceeding the scaling performance of prior algorithms.

Future work includes combining the two approaches into a hybrid approach that can provide both low latencies for small batches but addresses high variance, introducing approximate results during very high batch rates, and implementing these algorithms in distributed systems to further explore scalability.

Acknowledgment

This work was funded in part by the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[2] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *Journal of the ACM (JACM)*, vol. 30, no. 3, pp. 417–427, 1983.

[3] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tompkins, and E. Upfal, "The web as a graph," in *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2000, pp. 1–10.

[4] I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "k-core decomposition: a tool for the analysis of large scale internet graphs," *arXiv preprint cs.NI/0511007*, 2005.

[5] P. Hagmann, L. Cammoun, X. Gigandet, R. Meuli, C. J. Honey, V. J. Wedeen, and O. Sporns, "Mapping the structural core of human cerebral cortex," *PLoS Biol*, vol. 6, no. 7, p. e159, 2008.

[6] M. P. Van Den Heuvel and O. Sporns, "Rich-club organization of the human connectome," *Journal of Neuroscience*, vol. 31, no. 44, pp. 15 775–15 786, 2011.

[7] J. García-Algarra, J. M. Pastor, J. M. Iriondo, and J. Galeano, "Ranking of critical species to preserve the functionality of mutualistic networks using the k-core decomposition," *PeerJ*, vol. 5, p. e3321, 2017.

[8] H. A. Filho, J. Machicao, and O. M. Bruno, "A hierarchical model of metabolic machinery based on the k core decomposition of plant metabolic networks," *PloS one*, vol. 13, no. 5, p. e0195843, 2018.

[9] Y.-X. Kong, G.-Y. Shi, R.-J. Wu, and Y.-C. Zhang, "K-core: Theories and applications," *Physics Reports*, vol. 832, pp. 1–32, 2019.

[10] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu, "Effective and efficient attributed community search," *The VLDB Journal*, vol. 26, no. 6, pp. 803–828, 2017.

[11] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.

[12] R.-H. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 10, pp. 2453–2465, 2013.

[14] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.

[13] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 337–348.

[15] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Work-efficient parallel union-find with applications to incremental graph connectivity," in *European Conference on Parallel Processing*. Springer, 2016, pp. 561–573.

[16] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala, "Parallel batch-dynamic graph connectivity," in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019, pp. 381–392.

[17] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 918–934.

[18] N. Wang, D. Yu, H. Jin, C. Qian, X. Xie, and Q.-S. Hua, "Parallel algorithm for core maintenance in dynamic graphs," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2366–2371.

[19] H. Jin, N. Wang, D. Yu, Q.-S. Hua, X. Shi, and X. Xie, "Core maintenance in dynamic graphs: A parallel approach based on matching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2416–2428, 2018.

[20] Q.-S. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen, "Faster parallel core maintenance algorithms in dynamic graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1287–1300, 2019.

[21] J. E. Hirsch, "An index to quantify an individual's scientific research output," *Proceedings of the National academy of Sciences*, vol. 102, no. 46, pp. 16 569–16 572, 2005.

[22] L. Lü, T. Zhou, Q.-M. Zhang, and H. E. Stanley, "The h-index of a network node and its relation to degree and coreness," *Nature communications*, vol. 7, p. 10168, 2016.

[23] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 2, pp. 288–300, 2012.

[24] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *Proceedings of the VLDB Endowment*, vol. 12, no. 1, pp. 43–56, 2018.

[25] J. Shun, "Practical parallel hypergraph algorithms," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 232–249.

[26] B. Sun, T.-H. H. Chan, and M. Sozio, "Fully dynamic approximate k-core decomposition in hypergraphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 14, no. 4, pp. 1–21, 2020.

[27] A. E. Sariyüce and A. Pinar, "Fast hierarchy construction for dense subgraphs," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 97–108, 2016.

[28] N. S. Dasari, R. Desh, and M. Zubair, "Park: An efficient algorithm for k-core decomposition on multicore processors," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 9–16.

[29] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 1482–1491.

[30] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, 2017, pp. 293–304.

[31] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation: An index-based approach," in *The World Wide Web Conference*, 2019, pp. 1130–1141.

[32] Y. Zhang and J. X. Yu, "Unboundedness and efficiency of truss maintenance in evolving graphs," in *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 1024–1041.

[33] L. Gao, G. Gao, D. Ma, and L. Xu, "Coreness variation rule and fast updating algorithm for dynamic networks," *Symmetry*, vol. 11, no. 4, p. 477, 2019.

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

[35] J. Kunegis, "KONECT – The Koblenz Network Collection," in *Proc. Int. Conf. on World Wide Web Companion*, 2013, pp. 1343–1350. [Online]. Available: http://userpages.uni-koblenz.de/~kunegis/paper/kunegis-koblenz-network-collection.pdf