Decentralized and Self-adaptive Core Maintenance on Temporal Graphs

Davide Rucci[1], Emanuele Carlini[1], Patrizio Dazzi[2], Hanna Kavalionak[1], and Matteo Mordacchini[3]

[1] ISTI-CNR, Pisa, Italy
davide.rucci@isti.cnr.it, emanuele.carlini@isti.cnr.it,
hanna.kavalionak@isti.cnr.it
[2] University of Pisa, Pisa, Italy
patrizio.dazzi@unipi.it
[3] IIT-CNR, Pisa, Italy
matteo.mordacchini@iit.cnr.it

Abstract. Key graph-based problems play a central role in understanding network topology and uncovering patterns of similarity in homogeneous and temporal data. Such patterns can be revealed by analyzing communities formed by nodes, which in turn can be effectively modeled through temporal k-cores. This paper introduces a novel decentralized and incremental algorithm for computing the core decomposition of temporal networks. Decentralized solutions leverage the ability of network nodes to communicate and coordinate locally, addressing complex problems in a scalable, adaptive, and timely manner. By leveraging previously computed coreness values, our approach significantly reduces the activation of nodes and the volume of message exchanges when the network changes over time. This enables scalability with only a minimal trade-off in precision. Experimental evaluations on large real-world networks under varying levels of dynamism demonstrate the efficiency of our solution compared to a state-of-the-art approach, particularly in terms of active nodes, communication overhead, and convergence speed.

1 Introduction

Graphs are a fundamental data model in computer science, capable of representing virtually any relationship among a set of entities. Their structure allows many key problems to be addressed in a graph-based form, such as detecting connected components [13, 20], computing node centrality [2], solving vertex cover problems [4,26], and performing *core decomposition* [3,16,18]. These problems are closely related because their focus is on understanding the underlying organization of the graph, particularly in identifying communities—subsets of nodes that are densely connected internally but sparsely connected to the rest of the network. The detection and analysis of such community structures are crucial in many real-world applications, as proven by the many research efforts in the field. However, as graph data becomes increasingly large and available,

traditional centralized algorithms face significant limitations in terms of scalability, latency, and resource consumption. In response, decentralized computing models have emerged as a viable alternative. In fact, by distributing the computational workload across the network, decentralized systems enhance scalability and enable more efficient processing of large-scale graphs [21, 24].

A further layer of complexity arises when considering dynamic or temporal graphs, where nodes and edges change over time. This temporal dimension introduces new challenges for graph analysis, particularly for centralized systems that rely on global knowledge and batch processing. Centralized methods often struggle to accommodate rapid or frequent changes, resulting in inefficiencies and delayed responses. In contrast, decentralized approaches can naturally adapt to dynamic environments by leveraging localized communication and incremental updates. This adaptability allows nodes to recompute relevant metrics in response to changes without restarting from scratch, thereby maintaining efficiency in continuously evolving networks.

Within this context, this paper focuses on the decentralized *coreness* (or core number) computation and its evolution over time. Since their introduction in the 1970s [18], k-cores and their analysis have gained increasing popularity. Knowing a node's coreness – i.e., the largest k such that it belongs to the k-core – offers valuable insight into the network structure, supporting tasks like community detection and node ranking [6, 8, 9, 14]. Coreness reflects how densely connected a node is within its local neighborhood and indicates membership in cohesive subgraphs, making it a useful indicator of structural relevance.

In social networks, the coreness helps identify influential users that facilitate information spread [6]; it is also applied, among other uses, to anomaly detection [19] and routing optimization in blockchain networks [27]. Tracking coreness over time further aids in understanding the evolution of core structures and identifying stable, central components [5]. While some decentralized algorithms have been proposed for static k-core computation (e.g., Montresor et al. [16]), they do not handle temporal changes natively. Conversely, several centralized approaches exist for temporal k-core decomposition [7], but they lack the scalability of decentralized methods.

We introduce a decentralized, iterative, message-passing, and incremental algorithm for computing the k-core composition of temporal graphs. Our approach leverages previously computed coreness values to reduce the number of active nodes and exchanged messages, accepting a slight trade-off in accuracy. In fact, our experiments on large, real-world graphs show a reduction of 50%-90% in the number of total messages exchanged during the execution of our algorithm with respect to a state-of-the-art competitor.

To summarize, the main contributions of this paper are:

- A novel decentralized algorithm for coreness computation in temporal graphs;
- Simulation over large, heterogeneous real-world networks with varying temporal parameters;
- A comparative evaluation with a state-of-the-art approach, analyzing active nodes, message counts, and convergence;

– Public release of the code to foster reproducibility and comparison.

The remainder of this paper is structured as follows. Section 2 reviews the related work. Section 3 outlines the algorithm, notation, and definitions. Section 4 details the experimental setup and presents the main results. Finally, Section 5 concludes with a summary of key contributions and future research directions.

2 Related Work

Several studies have explored core decomposition in temporal networks, developing various definitions and centralized algorithms [15]. Yang et al. [25] introduce the Temporal k-core Query problem and propose the Temporal Core Decomposition algorithm, which efficiently computes k-cores across time intervals while minimizing redundant calculations. Li et al. [11] define a k-core model that ensures stability over time, using graph reduction techniques coupled with a branch-and-bound approach. Wu et al. [23] present a temporal core decomposition method that incorporates edge-based constraints to identify frequently interacting subgraphs.

Galimberti et al. [7] propose a method for analyzing temporal networks using temporal core decomposition. In their framework, each core is characterized by two parameters: the minimum degree and the span, which indicates the time interval over which the core persists. They develop efficient algorithms to compute these *span-cores* and demonstrate their effectiveness in real-world applications, such as analyzing contact networks and studying the evolution of social interactions. Although the approach of Galimberti et al. aligns with our philosophy of intersection-based decomposition (see Section 3.1), their definition of temporal core identification does not match our definition, since they have tailored it to compute a slightly different concept. A recent work by Conte et al. [5] offers a comprehensive summary of the various definitions of temporal k core that have been proposed in the above articles, highlighting the key differences and comparing the results that can be obtained by exploiting those definitions. This work also drives our algorithm design later in Section 3.

These contributions mostly focus on centralized algorithms for the temporal core decomposition task, assuming that a single computational entity processes the entire graph. Our work shifts the focus to decentralized solutions, aiming to distribute the computation across multiple nodes rather than relying on a central processing unit.

An influential solution to decentralize the computation of the k-core for static graphs was introduced by Montresor et al. [16]. They proposed a message-exchange algorithm based on the locality property of the k-core decomposition, which states that the coreness of a node is the highest number k for which the node has at least k neighbors in a k-core or higher. While their algorithm offers a fast convergence rate, it cannot be directly translated into a temporal scenario, as it would require recomputing everything from scratch each time the graph changes.

Aridhi et al. [1] propose distributed algorithms for efficient core decomposition and maintenance in large-scale dynamic graphs. Their approach addresses the computational challenges of massive and evolving graph structures by incrementally updating the core decomposition as the graph changes. However, such a solution relies on a hybrid model with a master node that orchestrates the update process, while different worker nodes handle the partitions. In contrast, our work proposes a fully decentralized solution, focusing on reducing the number of activated nodes and minimizing message exchanges, further optimizing performance in dynamic graph settings. Weng et al. [22] leverage a Pregel-like graph computing framework and focus on efficient algorithms to update core decompositions as the graph evolves. Their proposed methods aim to enhance the performance and scalability of core maintenance tasks within distributed computing environments, addressing challenges in large-scale graph processing. Liu and Zhang [12] introduce core maintenance algorithms for edge insertion and deletion, which are validated through experiments on real-world graphs, in dynamic, edge-weighted graphs. This is different from our approach, which is focused on unweighted temporal graphs. Yu et al. [28] present efficient algorithms for maintaining core numbers in dynamic graphs. They introduce the concept of a superior edge set to handle multiple edge insertions and deletions simultaneously, reducing redundant vertex visits. Their incremental and decremental core maintenance algorithms support parallel implementations, but remain centralized and are primarily designed for dynamic graphs, unlike our approach, which emphasizes decentralized solutions on temporal graphs.

In summary, while these studies offer valuable contributions to the core decomposition and maintenance task in dynamic and temporal networks, they predominantly rely on centralized or semi-centralized methods. In contrast, our approach is decentralized, focusing on minimizing the number of activated nodes and reducing redundant message exchanges. By distributing the computational load across multiple nodes and optimizing communication, we aim to achieve greater efficiency and scalability in dynamic environments, without depending on a central processing entity. Another key distinction lies in the concepts of temporal and dynamic graphs. Temporal graphs are inherently dynamic, but organized differently. Firstly, temporal graphs typically allow for the reconstruction of changes over time by providing snapshot- or epoch-based organization. On the other hand, dynamic graphs are usually designed to react to instant changes, also providing algorithmic methods to update necessary data structures whenever a single edge or node is inserted or deleted. Furthermore, in temporal graph algorithms, we can decide how significant an interaction is within a specific time interval. For example, we can prioritize a connection that lasts longer over many connections that exist only for a single snapshot.

3 Algorithm

This Section outlines our decentralized approach for determining the coreness of nodes in a temporal graph. We begin with fundamental notation and definitions

that will be referenced throughout the paper, followed by a presentation and analysis of our algorithm's pseudocode.

Algorithm 1: Distributed Algorithm for k-core computation in a temporal graph G_τ, run by each node $u \in V$.

```
1  on initialization do // when a node joins G_τ
2      changed ← false
3      core ← d(u)
4      foreach v ∈ N(u) do est[v] ← ∞
5  on epoch change do
6      oldCore ← core
7      if there is at least one new neighbor v then
8          core ← d(u)
9          changed ← true
10         est[v] ← ∞
11     else if a neighbor is lost then
12         core ← computeCoreness (est,u,core)
13         if oldCore ≠ core then changed ← true
14 on receive ⟨v, k⟩ do
15     est[v] ← k
16     t ← computeCoreness (est,u,core)
17     if t ≠ core then
18         if t < core then wait one iteration before sending update
19         core ← t
20         changed ← true
21 repeat
22     if changed then
23         send ⟨u, core⟩ to N(u)
24         changed ← false
25 until convergence
26 Function ComputeCoreness(est, u, k)
27     foreach e ∈ est[] do
28         if e < ∞ then
29             if e < d(u) then
30                 count[e] ← count[e] + 1
31             else
32                 count[d(u)] ← count[d(u)] + 1
33         else return min{core, d(u)}
34     total ← 0
35     for i = count.length down to 0 do
36         total ← total + count[i]
37         if total ≥ i then return i
38     return d(u)
```

3.1 Notation and Definitions

Formally, we define a temporal graph G_τ as a pair of sets of vertices and temporal edges (V, E_τ), where $\tau \in \mathbb{N}$ is called the *lifespan* of G, and the set of edges of G_τ is defined as $E_\tau = \{(u, v, t) \mid u, v \in V, 1 \leq t \leq \tau\}$, where t is the *timestamp* of an edge. If we consider only the edges of E_τ with a fixed value for $1 \leq t \leq \tau$, we obtain the static graph $G_t = (V, E_t)$ that is called *snapshot* or *epoch* of G at time t. A static *k-core* K is defined as the inclusion-maximal subset of vertices $K \subseteq V$ such that every vertex of K has degree at least k in the *vertex-induced subgraph* $G[K] = (K, E_K = \{(u, v) \in E \mid u, v \in K\})$. The maximum k for which a node v belongs to the k-core of a graph is called *coreness* (also *core number* in the literature [18]) of v. The main issue to be addressed when translating the concept of k-cores from static to temporal graphs is the selection of the most appropriate aggregation function, i.e., which edges are to be considered for the computation of cores for a given time interval. A recent study by Conte et al. [5], empirically shows that there exists no one-for-all solution to this problem; instead, a set of graphs may be analyzed with a set of distinct *aggregation functions* $af : E_\tau \times [a, b] \to E_\tau$ that, given a time interval $[a, b]$ tell us which edges to consider as present in the graph for that interval. We informally summarize here the possible functions that can be adopted, referring the reader to the paper by Conte et al. [5] for a more formal overview of those and when to use which. Given a time interval $[a, b], 1 \leq a \leq b \leq \tau$, we can consider the edge $(u, v, t) \in E_\tau$ to exist in that interval if:

- *(intersection)*: the edge $(u, v, t) \in E_\tau$ for *all* $t = a, \ldots, b$;
- *(union)*: the edge $(u, v, t) \in E_\tau$ for *any* $t = a, \ldots, b$;
- *(union-h)*: the edge $(u, v, t) \in E_\tau$ for *at least* h distinct values of $t = a, \ldots, b$. We will denote this function by $\cup_h$ later in the paper.

We denote the *neighborhood at time* t of $v \in V$ as $N_t(v)$, or just $N(v)$ whenever t is clear from the context. Similarly, the *degree at time* t of $v \in V$ is $d_t(v) = |N_t(v)|$, omitting the t when clear from the context. A key question to address each time we use intervals on temporal graphs is: how do we choose both the extremes and the length of intervals? Our answer is to introduce a parameter, called *memory size*, to serve as the length of every interval; then, we use a sliding window approach, spanning the entire lifetime of the graph. For example, setting the memory size to 5 implicitly allows nodes to "recall" the last 5 snapshots in their neighborhood. Then, we use the edge aggregation function to decide which neighbors are actually part of the graph in the interval.

3.2 Our Decentralized Algorithm

Our strategy is formalized in the pseudocode of Algorithm 1. It is a message-exchange approach in which the computation is organized in rounds or *iterations*. For each iteration, the nodes communicate estimates of their coreness value to their neighbors, which, in turn, adjust theirs based on what they received. We note that Algorithm 1 is fully decentralized and can run for an indefinite period

of time, as it can adapt to every change in the graph. There are three main parts into which we can subdivide the algorithm that runs from every node present in the graph: *initialization*, *new epoch*, and *reception of a message*, and we analyze them separately.

Initialization. When a new node entity is generated, the initial step is performed: it assigns its coreness estimate to its degree, as this is the sole information available at creation, sets its neighbors' estimates to infinity since they are unknown, and then communicates its degree to its neighbors.

Epoch Change. When the event of a new epoch in the temporal graph occurs, there are three possibilities for each node: *(a)* the node gains a new neighbor, *(b)* the node does not gain any new neighbor but loses at least one neighbor instead, or *(c)* the neighborhood of the node is not affected. Only the first two possibilities trigger a reaction of the node:

(a) the node acquires at least one new neighbor, prompting it to update its coreness estimate to reflect its new degree, as this ensures accuracy.
(b) the node experiences a loss of one or more neighbors without gaining any new ones; under these circumstances, the node attempts to recalculate its coreness using knowledge of its remaining neighbors. This capability is crucial to the algorithm, allowing us to use preexisting graph information and bypass the need to transmit potentially unnecessary messages. This approach also helps in identifying whether the departure of neighbors affects coreness, without the need to wait for a complete message cycle.

Note that case *(a)* takes precedence over case *(b)*, since the simultaneous occurrence of these events results in at least one infinite value in the node's estimate table, potentially raising its coreness value. Therefore, we opt for a cautious strategy by resetting the coreness to the degree.

Message received. The last event that we consider is the reception of a message from a neighbor, which triggers an update in the table of estimates of the recipient and possibly a change in the coreness value. The main aspect is the check on line 18, which postpones sending an update message to the following iteration if our coreness estimate is less than before. This helps reduce potential errors originating from delayed change propagation across the graph, since a neighbor might experience an increase in its coreness when a new node joins its neighborhood. Indeed, it requires two iterations for a node to detect a change in the coreness of its neighbors if those neighbors have acquired new connections within the same epoch. We will address this issue later.

3.3 Example

Fig. 1 illustrates our algorithm in action. The initial graph consists of three nodes, **A**, **B**, and **C**, all interconnected. In the new epoch, a new node **D** arrives,

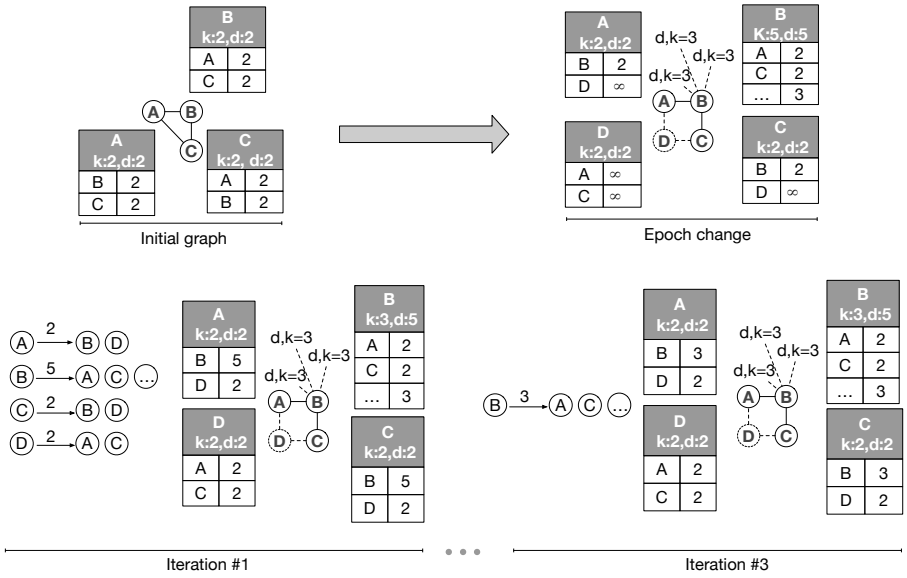

Fig. 1. An example of Algorithm 1 in action. Dashed nodes and edges are inserted in the new epoch.

connecting with nodes **A** and **C**, while the link between **A** and **C** disappears. Furthermore, three new neighbors of **B** appear, each with a degree and coreness of 3. For the sake of clarity and simplicity, in this example we ignore the updates sent by the new neighbors of **B**, and show only the local effects on the initial nodes. Since each node has acquired at least one new neighbor, they all revert their coreness to match their degree and adjust the table entry for any new neighbors to infinity. In iteration # 1, all nodes that received a new neighbor (and thus have their `changed` variable now set to `true`) send a new message with their core estimate (set to their degree) to their neighbors. Nodes **A**, **C**, and **D** send a value of 2, while **B** sends a value of 5. As a result, all nodes get messages and revise their coreness. Nodes **A**, **C**, and **D** have unchanged coreness and therefore cease sending messages. Node **B** alters its coreness to 3, down from 5. Consequently, it delays messaging its neighbors for one iteration to account for possible propagation delays due to changes occurring in its two-hop neighbors. The message is finally sent at Iteration #3. Since this new information received does not change the coreness of any neighbors, the process stops.

3.4 Efficiency/errors trade off

Since Algorithm 1 is the result of empirical and iterative refinements, its strategy may lead to errors in the computed coreness for some nodes. This primarily results from each graph node consistently treating the estimates it receives from

its neighbors as valid, rather than resetting them to infinity with each epoch change. Thus, while executing the function on line 26, a node's coreness may abruptly shift from a high value, such as its degree, to a minimal coreness value, bypassing potential intermediate values. Generally, this isn't a problem since the coreness can vary; however, this sometimes causes nodes to compute incorrect values. Our empirical analysis revealed that this phenomenon typically occurs when new nodes enter the graph. Neighbors, located two hops from these new nodes, tend to calculate an incorrect value compared based on the competitor we considered, which also serves as the ground truth for coreness values. This explains why we implemented a delay before dispatching updates when a node calculates a coreness lower than its prior value: this allows the node to potentially determine a new and accurate coreness in the next cycle if it gets additional updates from its neighbors. Adopting a method that allows for some errors involves a compromise between strategy efficiency and error count: indeed, to completely eliminate errors, we would need to reset all estimates whenever an epoch changes, requiring each node to send at least one message for every epoch in the graph. Conversely, by extensively reusing our existing estimates, we risk calculating an incorrect value, but gain a notably faster algorithm.

As we verified (see Section 4) that the offset of the errors produced by Algorithm 1 is always in the range ± 1 and that, most importantly, it affects a very small portion of nodes in the whole graph, we accept to have some small errors to get a big saving in the efficiency metrics in return.

Table 1. Overview of our dataset.

Dataset	Kind	Max. Nodes	Total Edges
AS-733	Autonomous System	7716	11410810
Email-EU-Core	Email Exchange	986	332334
Rec-Amazon-Ratings	Amazon Product Reviews	2146057	5838027
sx-mathoverflow	Q&A	24759	390441
reddit	Hyperlinks between subreddits	54075	858488

4 Experimental evaluation

This section provides an overview of our comprehensive experimental phase, describes the algorithm selected as the competitor for validating our analysis, and presents a discussion of the key results achieved.

4.1 Competitor

To compare and verify our results, we take advantage of a revised version of the algorithm originally proposed by Montresor et al. [16], which was designed

for static graphs. The structure of this algorithm mirrors that of Algorithm 1, meaning each node monitors updates to its own coreness estimate and maintains an estimate of the coreness of its neighboring nodes. Montresor et al. provide a formal proof demonstrating that their method consistently converges to the accurate coreness value for every node in a graph [16]. This algorithm can be modified to accommodate the temporal scenario by recalculating completely whenever there is an epoch transition: each node discards all estimates about its neighbors and assigns its coreness equal to its degree. We use this adaptation to compute ground truth values for the coreness of nodes, and to show the amount of savings we can achieve by adopting our fine-tuned strategy instead of a direct translation of an existing algorithm to the temporal context.

Table 2. Summary of results obtained in our experiments. $\cup_h$ is the union-h function (see Section 3.1). Results are averaged over all the executions of algorithms for all epochs. The memory size parameter is set to 5. Error percentage is relative to the number of nodes in the graph.

Dataset	Aggr. Function	Epochs (length)	Avg. Activated Nodes			Avg. Total Messages			Avg. Iterations			Avg. Errors per Epoch
			Alg. 1	Comp.	Ratio	Alg.1	Comp.	Ratio	Alg.1	Comp.	Ratio	
AS-733	$\cap$	105 (7 days)	372	3968	0.09	582	4950	0.11	10	7	1.39	13.7 (0.2%)
AS-733	$\cup$	105 (7 days)	431	4630	0.09	741	5912	0.12	14.1	9.35	1.51	16.6 (0.3%)
email-Eu-core	$\cap$	75 (7 days)	91	249	0.38	135	331	0.40	7.5	5.5	1.37	4.21 (0.4%)
sx-mathoverflow	$\cup$	334 (7 days)	701	1481	0.47	1401	2530	0.55	24.8	13.8	1.79	9.92 (0.1%)
reddit	$\cup_2$	174 (7 days)	813	2103	0.38	1291	2799	0.46	15.5	8.5	1.81	10.04 (0.02%)
reddit	$\cup_2$	87 (14 days)	813	2103	0.38	1291	2799	0.46	15.5	8.5	1.81	10.1 (0.02%)
reddit	$\cap$	174 (7 days)	62	186	0.33	82	219	0.38	6.15	4	1.52	0.5 (0.001%)
rec-amazon-ratings	$\cup$	124 (28 days)	69029	220240	0.31	105124	285357	0.36	45.9	24.7	1.85	1344.5 (0.1%)

4.2 Experimental Setup and Dataset

We implemented both algorithms (Algorithm 1 and the aforementioned competitor by Montresor et al. [16]) in the Rust programming language, and carried out our experiments on the following architecture: Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz, 8 physical cores, 16 logical cores with 64 GB of RAM, and 16 MB of shared L3 cache. Our code is publicly available on GitHub[4].

To simulate the distributed environment, we used two queues for gathering nodes and messages sent by the nodes, to dispatch them to the appropriate recipients. Initially, we gather the indices of nodes required to transmit a message as a vector. Subsequently, each node sends its message to a separate message queue, and ultimately, each message is sent to the appropriate recipient for processing. A node that receives a message and needs to send a new one will postpone its transmission until the next iteration of the algorithm. To streamline deployment and evaluation, our implementation includes a central entity called the *Graph* that verifies the convergence of the algorithm and the coreness computation of each node. However, we remark that Algorithm 1 is entirely decentralized and works correctly without an orchestrator.

[4] https://github.com/DavideR95/temporal_distributed_kcores

We run the experiments on a dataset of real-world temporal graphs obtained from the SNAP Repository [10] and Network Repository [17]. The dataset, summarized in in Table 1, offers a varying level of dynamism, i.e. how much and how fast a graph changes over time, that can be noticed by looking at the plots in Figs. 2(d) and 3(d), which show how much a graph changes between two consecutive epochs. For each graph, we only give the maximum number of distinct nodes that have been active at least once throughout the lifespan of the graph. For any given epoch, the number of nodes that are part of the graph, i.e., that have at least one incident edge, is upper-bounded by this value. The number of edges reported follows a similar rationale, representing the total count of edges present in the graph during at least one epoch. We manually chose the epoch length for each graph by grouping edges according to their timestamp, in a way such that each epoch is sufficiently populated with nodes and edges so that the results obtained are not trivial (e.g. nodes are not isolated and there is enough change in active edges for any two consecutive epochs). We run both algorithms on the whole dataset, collecting the following metrics:

- *Activated nodes*: nodes that sent at least one message during the execution of the algorithms.
- *Number of iterations*: number of iterations of the main loop of the algorithms needed at a certain epoch to terminate their execution.
- *Number of messages*: the number of messages sent by all nodes for each iteration of the algorithms.
- *Errors*: number of nodes that computed a different coreness value in Algorithm 1 with respect to the ones computed by our competitor.

Different values for the *memory size* of each node were tested, spanning from a minimal size (1) to a larger size (10). Given space constraints, we present results solely for a memory size of 5, which represents a compromise between the extremes. Moreover, as we pointed out in Section 1, we tested different edge aggregation functions, namely intersection, union and union-2 (or *half*), i.e., every edge must appear at least half of the memory size times (for a memory size of 5, *half* corresponds to 2, $= \lfloor 5/2 \rfloor$).

4.3 Results and Discussion

Table 2 summarizes the results we obtained for Algorithm 1, compared to our competitor, highlighting the ratio between the two. We can immediately notice that the ratio of both the activated nodes and the total messages is always well below 1, showing a big savings on these two critical measures.

Another thing we notice is the average number of iterations performed by Algorithm 1 against the competitor: this is due to our strategy of delaying the action of sending an update to the next iteration, to reduce the error rate. Although this may appear as a step back with respect to the Montresor et al. strategy [16], it is important to note that despite this, the number of activated nodes and the total number of messages exchanged do not increase. Thus, the

savings in resources are still consistent even if the number of iterations to reach convergence increases. Moreover, in our experiments, this number never exceeded twice the amount of the competitor. Fig. 2(c) shows the values of the number of iterations for the AS-733 dataset, where we can further verify this statement.

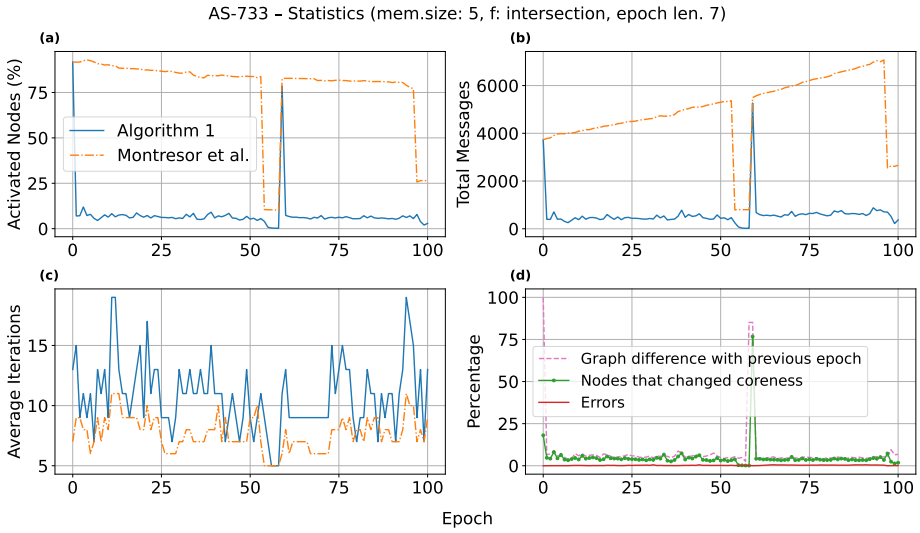

Fig. 2. Visualization of the results obtained on the *AS-733* dataset, for the edge aggregation function *intersection*.

We now turn to the error rate. Recall that we count one error for each node that, at the end of an execution of Algorithm 1, has a coreness value different from the one computed by our competitor algorithm. Table 2 shows the average number of errors per epoch, both in absolute and relative terms, relative to the number of nodes in the graph. We consistently obtained less than 1% of errors in every configuration for every dataset, proving we can achieve a significant speedup without losing too much accuracy. Additionally, in our experiments, we verified that all errors are always ± 1 with respect to the correct value of coreness computed by our competitor. It is interesting to see how the algorithm reacts to different epoch lengths when the aggregation function and memory size are fixed. Indeed, there is basically no difference between the results for the Reddit dataset when the epoch length is enlarged to 14 days instead of 7. On the other hand, if we change the function and keep the same epoch length, things change drastically (e.g. Reddit dataset with intersection).

Fig. 2 and Fig. 3 go more in depth on the statistics we gathered for two specific datasets and configurations, namely AS-733 with intersection as the edge aggregation function, and sx-mathoverflow with union. In particular, we plotted the percentage of activated nodes for our algorithm and our competitor by

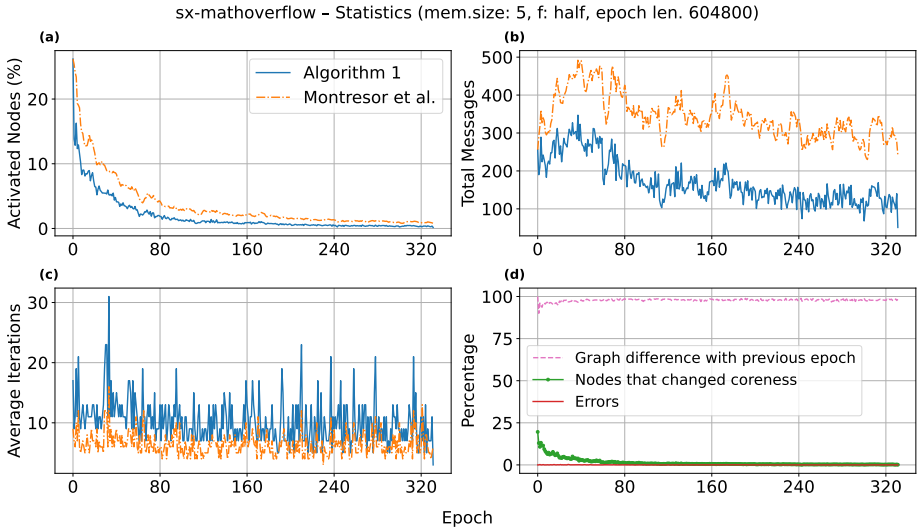

Fig. 3. Visualization of the results obtained on the *sx-mathoverflow* dataset, for the edge aggregation function *union-2*

Montresor et al. [16], together with two measures of how much the corresponding graph has changed with respect to the previous epoch. We first computed the Jaccard similarity between the edge list of any two consecutive epochs and subtracted it from 1 to obtain a measure of how much the graph has changed from the perspective of the edges. This number is trivially equal to 1 for the first epoch of every graph.

We also calculated the number of nodes that changed their coreness value with respect to the previous epoch: this number also includes those nodes who became isolated (i.e. have coreness equal to 0 or, equivalently, are not part of the graph anymore) on an epoch, hence the spikes in Fig. 2. Correctly, the number of activated nodes stays low even in the presence of such spikes, because the isolated nodes do not send any message, therefore, they cannot become active. On the other hand, we see that the trend of activated nodes in AS-733 (Fig. 2) follows the trend of how much the edges in the graph change, as high values of dissimilarity (pink line) correspond to higher values of activated nodes by Algorithm 1. This holds for our competitor too, but that algorithm is blind to changes in the graph[5], as it recomputes everything from scratch each time.

4.4 Takeaways

The experimental results suggest two main conclusions about our approach:

[5] For the AS-733 dataset, there is a significant change in the graph in epochs 54-59. This is intrinsic to the dataset itself and not caused by our implementation. The possible reasons behind this sudden change are discussed in [5].

1. Algorithm 1 significantly reduces the number of exchanged messages compared to the competitor solution. This translates into lower computational demand and enables the analysis of larger graphs than previously feasible.
2. Although our algorithm requires slightly more iterations to converge, it maintains a low level of node activation throughout. As a result, the majority of nodes remain idle during execution, minimizing overall resource consumption and increasing scalability despite the increased number of rounds.

5 Conclusions

We presented a new decentralized algorithm for the maintenance of core decomposition in large temporal graphs, an important task for community detection and analysis in nowadays networks. We conducted an extensive experimental phase, demonstrating significant performance improvements compared to the direct translation of an existing algorithm [16] into the temporal scenario. These improvements are measured in terms of the total messages exchanged during the algorithm's execution and the number of graph nodes that send at least one message. While this strategy sometimes leads to small errors in the computed coreness values, we showed that these errors are *(a)* always at ± 1 unit from the true value, and *(b)* extremely infrequent in our real-world dataset. Our findings pave the way for future work on this algorithm, to ensure its correctness in all cases while maintaining the same desirable performance.

References

1. S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis. Distributed k-core decomposition and maintenance in large dynamic graphs. In *Proc. of Int. Conference on Distributed and Event-Based Systems*, DEBS '16, page 161–168. ACM, 2016.
2. G. Audrito, D. Pianini, F. Damiani, and M. Viroli. Aggregate centrality measures for IoT-based coordination. *Science of Computer Programming*, 203:102584, 2021.
3. V. Batagelj and M. Zaversnik. An O(m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.
4. E. Carlini, P. Dazzi, A. Makris, M. Mordacchini, T. Theodoropoulos, and K. Tserpes. Efficient application image management in the compute continuum: A vertex cover approach based on the think-like-a-vertex paradigm. In *2024 IEEE 17th Int. Conf. on Cloud Computing (CLOUD)*, pages 399–403. IEEE, 2024.
5. A. Conte and D. Rucci. Are k-cores meaningful for temporal graph analysis? In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, SAC '24, page 1453–1460, New York, NY, USA, 2024. ACM.
6. S. Fortunato. Community detection in graphs. *Phys. Rep.*, 486(3):75–174, 2010.
7. E. Galimberti, M. Ciaperoni, A. Barrat, F. Bonchi, C. Cattuto, and F. Gullo. Span-core decomposition for temporal networks: Algorithms and applications. *ACM Trans. Knowl. Discov. Data*, 15(1), dec 2020.
8. C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In *2011 International Conference on Advances in Social Networks Analysis and Mining*, pages 87–93, 2011.

9. Y.-X. Kong, G.-Y. Shi, R.-J. Wu, and Y.-C. Zhang. k-core: Theories and applications. *Physics Reports*, 832:1–32, 2019. k-core: Theories and Applications.
10. J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
11. R.-H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai. Persistent community search in temporal networks. In *IEEE Int. Conf. on Data Engineering*, pages 797–808, 2018.
12. B. Liu and F. Zhang. Incremental algorithms of the core maintenance problem on edge-weighted graphs. *IEEE Access*, 8:63872–63884, 2020.
13. A. Lulli, E. Carlini, P. Dazzi, C. Lucchese, and L. Ricci. Fast connected components computation in large graphs by vertex pruning. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):760–773, 2017.
14. F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis. The core decomposition of networks: theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92, Jan 2020.
15. F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92, 2020.
16. A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Trans. Parallel Distrib. Syst.*, 24(2):288–300, Feb. 2013.
17. R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.
18. S. B. Seidman. Network structure and minimum degree. *Soc. Netw.*, 5(3):269–287, 1983.
19. K. Shin, T. Eliassi-Rad, and C. Faloutsos. Corescope: Graph mining using k-core analysis — patterns, anomalies and algorithms. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 469–478, 2016.
20. O. Ugurlu, N. Akram, Y. Aygul, V. K. Akram, and O. Dagdeviren. Distributed detecting of critical nodes for maximization of connected components in wireless multi-hop networks. *Ad Hoc Networks*, 169:103744, 2025.
21. M. Vincent, A. E. George, T. Christa, and N. Jayapandian. Systematic review on decentralised artificial intelligence and its applications. In *2023 International Conference on Innovative Data Communication Technologies and Application (ICIDCA)*, pages 241–246. IEEE, 2023.
22. T. Weng, X. Zhou, K. Li, P. Peng, and K. Li. Efficient distributed approaches to core maintenance on large dynamic graphs. *IEEE Trans. Parallel Distrib. Syst.*, 33(1):129–143, 2022.
23. H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu. Core decomposition in large temporal graphs. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 649–658. IEEE, 2015.
24. H. Xu, K. P. Seng, L. M. Ang, and J. Smith. Decentralized and distributed learning for AIoT: A comprehensive review, emerging challenges and opportunities. *IEEE Access*, 2024.
25. J. Yang, M. Zhong, Y. Zhu, T. Qian, M. Liu, and J. X. Yu. Scalable time-range k-core query on temporal graphs. *Proc. VLDB Endow.*, 16(5):1168–1180, jan 2023.
26. Y. Yigit, V. K. Akram, and O. Dagdeviren. Breadth-first search tree integrated vertex cover algorithms for link monitoring and routing in wireless sensor networks. *Computer Networks*, 194:108144, 2021.
27. B. Yin, P. Zhang, and T. Chen. Efficient state sharding in blockchain via density-based graph partitioning. *ACM Trans. Web*, 19(1), Dec. 2024.
28. D. Yu, N. Wang, Q. Luo, F. Li, J. Yu, X. Cheng, and Z. Cai. Fast core maintenance in dynamic graphs. *IEEE Trans. Comput. Soc. Syst.*, 9(3):710–723, 2022.