# A Parallel Algorithm for Tracking Dynamic Communities based on Apache Flink

Georgios Kechagias
School of Electrical and Computer Engineering
Technical University of Crete
Chania, Greece
gkechagias@isc.tuc.gr

Grigorios Tzortzis
Institute of Informatics and Telecommunications
NCSR "Demokritos"
Athens, Greece
gtzortzi@iit.demokritos.gr

Dimitrios Vogiatzis
The American College of Greece, Deree
Athens, Greece
Institute of Informatics and Telecommunications
NCSR "Demokritos"
Athens, Greece
dimitrv@iit.demokritos.gr

George Paliouras
Institute of Informatics and Telecommunications
NCSR "Demokritos"
Athens, Greece
paliourg@iit.demokritos.gr

## ABSTRACT

Real world social networks are highly dynamic environments consisting of numerous users and communities, rendering the tracking of their evolution a challenging problem. In this work, we propose a parallel algorithm for tracking dynamic communities between consecutive timeframes of the social network, where communities are represented as undirected graphs. Our method compares the communities based on the widely adopted *Jaccard similarity* measure and is implemented on top of Apache Flink, a novel framework for parallel and distributed data processing. We evaluate the benefits, in terms of execution time, that parallel processing brings to community tracking on datasets carrying different quantitative characteristics, derived from two popular social media platforms; Twitter and Mathematics Stack Exchange Q&A. Experiments show that our parallel method has the ability to calculate the similarity of communities within seconds, even for large social networks, consisting of more than 600 communities per timeframe.

## CCS CONCEPTS

• **Human-centered computing** → **Social network analysis**; • **Theory of computation** → **Graph algorithms analysis**; • **Computing methodologies** → **Parallel algorithms**;

## KEYWORDS

Social Network Analysis, Community Tracking, Parallel Processing, Apache Flink

## 1 INTRODUCTION

In real world social networks, users who share common interests or activities (e.g. hobbies, work) closely interact (e.g. via likes, retweets) and form user communities. The membership of communities tends to change gradually as a result of users changing interests. As social interactions in these networks evolve, it is of great importance to develop analysis tools that effectively capture this evolution. In social network analysis, social networks are often represented as graphs, the social actors, or users, constitute the vertices of these graphs, and the edges indicate the social relationships or interactions between the network actors. A social network graph usually consists of several communities which are commonly defined as densely connected subsets of users (i.e. subgraphs) who are loosely connected with others.

There exist two main directions in analyzing social networks represented as graphs. One is to handle them as static graphs which do not change as time passes. However, real life social networks, are dynamic and tend to evolve over time since their users dynamically join or leave communities, affecting the network's composition. Hence considering the dynamic nature of communities in their analysis, in order to reveal their structure and evolution, is of great importance [1, 3, 5, 10, 13]. Tracking the evolution of communities has several real life applications such as, in cable and mobile network management, to optimize the structure and capacity of their networks; in journalism, to check how a story develops and if it is gaining momentum; in law enforcement, to track criminal and terrorist network activity.

In order to model dynamic communities in a social network, it is essential to initially discretize the time dimension by segmenting a timestamped social network into snapshots, also called timeframes, that cover different, and possibly overlapping, time periods. Then by using community detection algorithms, such as the Louvain method [2], sets of highly intra-connected vertices (i.e. communities) are identified for each timeframe. Studying the evolution of a community, which is the main focus of the dynamic community tracking literature [1, 3, 10], consists of locating the instances (i.e. counterparts) of the community in the different timeframes. Often an event is also assigned between two instances of the same community to designate the type of evolution. Commonly used events are *community continuation*, *community shrinkage*, *community growth* and *community dissolution*.

Despite the recent bloom of methods tackling the problem of dynamic community tracking, one major issue of such approaches is the high number of comparisons required to calculate evolutionary events. In the typical case, community tracking algorithms have to compare every single community of a given timeframe, with

every community contained in the next one in order to uncover their counterparts and categorize their temporal evolution [3]. This procedure is executed, for all communities in every timeframe. Given that contemporary real world social networks, contain thousands or even millions of users and communities, it is clear that dynamic community tracking algorithms have to scale to that order of magnitude.

In this work, we propose a simple and efficient parallel community tracking method, which compares communities of consecutive timeframes based on the widely adopted *Jaccard similarity* measure. We implemented our method using Apache Flink[1], a framework for parallel and distributed data processing, which allows community comparisons to be executed across multiple CPUs. Subsequently, we evaluated our method using real world datasets derived from popular social media platforms such as Twitter[2] and Mathematics Stack Exchange Q&A[3]. Performing experiments using datasets containing different number of timeframes, communities and community sizes, enables the better assessment of the scalability of our method and of the behavior of Apache Flink under different loads. Finally, by conducting experiments using various Apache Flink parallelism configurations, we were able to fine tune the parallel framework for the given datasets and for the specific machine used for our experiments, thus leading to the fastest execution.

Our method has the ability to calculate the similarity of communities within seconds, scales quite well to social networks consisting of thousands of users and communities, and it is easily extensible towards supporting different similarity measures with minimum effort from the end user. Furthermore, as our method is implemented using the Apache Flink framework, it has the potential to scale to even bigger social networks, as long as more computational resources are available.

The rest of this paper is structured as follows. In Section 2 we briefly review related work. In Section 3 we introduce basic notation and formalize our problem. Then, in Section 4, we clarify some important characteristics of the functionality of Apache Flink. In Section 5, we present our parallel algorithm for tracking dynamic communities, whereas, in Section 6 we describe the characteristics of the datasets used in our experiments. The results and the evaluation of our method are discussed in Section 7. Finally, in Section 8 we conclude this work and discuss future directions.

## 2 RELATED WORK

**Dynamic Community Tracking**. Many methods have been proposed for tackling the problem of dynamic community tracking. In the general case, these methods are divided in two distinct categories. The one-step methods and the two-step methods. One-step methods are those which combine community detection and community tracking in a single-simultaneous step. Two-step methods, require at the first step, the application of a community detection algorithm. Then, at the second step, the execution of the community tracking algorithm using as input the previously detected communities follows. Two-step approaches do not tend to focus on the first step (community detection), but rather on tracking of the evolution of the already detected communities. Our approach falls under the two-step category.

Gauvin et al. [6], proposed an one-step method to simultaneously identify network communities together with their activity patterns over time based on non-negative tensor factorization techniques. They represented the interaction between users across time using a 3-dimensional tensor $T \in \mathbb{R}^{N \times N \times T}$, where $N$ denote the number of nodes (i.e. users) of the network and $S$ the number of network timeframes. This approach outputs two matrices that represent the community detection and the community tracking results. Sarantopoulos [12], proposed *Timerank*, an one-step method for detecting and tracking communities in dynamic social networks. This method represents the social network as a tensor, utilizes *Muturank* [14] in order to rank the timeframes, and as a final step, it applies spectral clustering to obtain the dynamic communities.

In the context of two-step methods, Asur et al. [1], proposed an event-based framework for characterizing dynamic communities which contain various evolutionary events such as, group *continue*, *dissolve* and *form*. They evaluated their framework using two real world datasets; one derived from DBLP (Digital Bibliography & Library Project)[4] and another from the clinical trials of a major pharmaceutical company. Brodka et al. [3], proposed Group Evolution Detection (GED), a method for dynamic community tracking based on a measure called *inclusion* which captures both the quantity (i.e. the number of members) and the quality (i.e. the importance) of members of a community. Goldberg et al. [7], proposed a two step method, which is based on an axiomatic foundation for the evolution of communities. By viewing each dynamic community as a chain, they proposed three axioms: (1) *Identity*, (2) *Monotonicity* and (3) *Extension* which formulate the chain strength based on the strength of the chain's weakest links. In this way, they correlated the lifespan of the communities to structural parameters of their early evolution. Greene et al. [8], addressed the problem of dynamic community tracking, by representing the social network as a collection of graphs, one for each timeframe. After applying a static community detection algorithm to extract the dynamic communities in each timeframe, they developed a community tracking strategy which finds the counterparts of the communities based on the *Jaccard similarity*. For the evaluation their method, they used synthetic datasets, as well as data coming from a real mobile network. Pavlopoulou et al. [11], tackled the problem of community evolution prediction. In their work employed community tracking, in order to assign evolutionary events to communities of a social networks and thus to derive a "ground truth" for training their prediction model.

**Distributed Processing using Apache Flink**. Apache Flink is a relatively new distributed data processing framework, which offers both streaming and batch data processing capabilities. Various works in the broad field of artificial intelligence are based on Apache Flink. Ciobanu and Lommatzsch [4], developed a news recommender system based on the DataStream API of Apache Flink, in order to facilitate the processing of news steams which at a next step were used to update their recommender system's models. The evaluation of their method performed in the context of a news recommendation benchmarking platform called *CLEF NewsREEL*.

[1] https://flink.apache.org/
[2] https://twitter.com/
[3] https://math.stackexchange.com/
[4] http://dblp.uni-trier.de

Kalavri et al. [9], used Apache Flink DataSet API in order to create methods for detecting web trackers and defending ordinary Internet users. Their methods were based on simple classifiers such as nearest neighbor and label propagation. They modeled user browsing as a graph using data they derived from user traces collected by a real world web proxy.

## 3 PROBLEM DEFINITION

Before describing our framework, it is important to introduce the basic notation used in this work. A static undirected graph is denoted by $G = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. A dynamic social network, is a collection of undirected graphs corresponding to the timeframes, that capture the time segmentation of the social network. This dynamic social network is defined as a set $TF = \{TF_0, TF_1, \ldots, TF_{T-1}\}$ of timeframes with $T$ being the number of timeframes available. Each timeframe $TF_t$, contains a set of communities (i.e. subgraphs of the timeframe's graph) $\{C_t^0, C_t^1, \ldots, C_t^{K_{t-1}}\}$ with $K_t$ denoting the number of communities in that timeframe. Each community $C_t^k$ contains a set of vertices $V_{t,k}$ corresponding to the users of the community and a set of edges $E_{t,k}$ representing the interactions among its users.

The main problem addressed in this work, is the speed up of community tracking by parallelizing the execution of the comparisons required to match communities between sequential timeframes, using Apache Flink as the underlying parallelization framework. Our method follows the two-step paradigm and each community $C_t^k$ of a given timeframe $TF_t$ must be compared to all communities $C_{t+1}^j$ of the following timeframe. To compare communities, a community similarity measure is needed. As similarity measure we choose the *Jaccard similarity*, as it is very easy to calculate and has already been exploited in tracking communities [7, 8]. Given a pair of communities at consecutive timeframes $C_t^k$, $C_{t+1}^j$, their *Jaccard similarity* is defined as:

$$Jaccard(C_t^k, C_{t+1}^j) = \frac{V_{t,k} \cap V_{t+1,j}}{V_{t,k} \cup V_{t+1,j}} \quad (1)$$

## 4 APACHE FLINK

Apache Flink is an open source framework for distributed data streaming applications. It offers data structures and functions that help users in parallelizing their algorithms which then Flink is responsible to execute. Thus the details of parallel programming are abstracted away. Flink consists of three distributed components: (1) the *job client*, (2) the *job manager* and (3) the *task manager*. All these components have to communicate in order to parallelize and execute the submitted program. The *job client* takes a Flink job (i.e. an algorithm) as input, creates its *job graph* which is a representation of the algorithm's workflow, and submits it to the *job manager*. The *job manager* is responsible to determine the resource allocation, task scheduling and state reporting of the program. When a *task manager* [5] becomes available and has successfully registered with the *job manager*, the latter starts to distribute the individual tasks of a job to the *task manager*. Indicative tasks of a job in Apache Flink are illustrated in Figure 1, where each rectangle corresponds to a different task (e.g. *Data Source*, *GroupReduce*, *Cross DataSink*).

It is obvious from the above description, that a Flink program does not start its execution immediately, like traditional sequential programs, but only after a task manager receives a task and spawns a thread to execute it.

The *Task manager* is a JVM (Java Virtual Machine) process which executes tasks using one or more threads. The *task manager* may have one or more available *task slots*, whose number determines the degree of Flink's *parallelism* of task execution. In more detail, tasks are possible to be splitted into several subtasks which are distributed to the available task slots for execution, and each such subtask, processes an appropriate subset of the task's input data. It is Flink's responsibility to decide the size of the input of each subtask. The number of parallel subtasks of a Flink task, is defined by the programmer by setting the number of task slots (i.e. the parallelism degree).

In Flink a task slot represents a fixed subset of computational resources managed by the task manager (e.g. a task manager with 6 slots, is going to dedicate $\frac{1}{6}$ of its managed RAM memory to each slot). In this fashion, Flink achieves that subtasks of the same task won't compete for memory, but instead have a certain amount of reserved memory. Furthermore it is important to mention that Flink by default utilizes all available CPU cores (even in the case of a single task slot) and that a task slot is not bound to a particular CPU core. Nevertheless, it is recommended by the Flink documentation to create as many task slots as the number of a system's CPU cores. The reason for this, is to guarantee that a physical CPU core is available for each task slot, reducing kernel context switching and, thus, increasing the system's overall performance.

As Flink by default spawns a thread for each task of a job, this means that these tasks are ready to start their execution as long as the input that corresponds to them is available. Hence Flink programs are inherently parallel, even in the scenario of *parallelism* 1, i.e. when single *task slot* is used. The only waiting time for some specific tasks occurs when their input depends on the output of other ancestral tasks which are still executing. An example which illustrates this idea is shown in Figure 1. Here, the *Filter* tasks start their execution at the same time, as they all depend on the same ancestral task *GroupReduce*. *Cross-Map* tasks, whose inputs are the results of the *Filter* tasks, are waiting for at least one of them to finish its processing in order to start their execution. This does not mean that they start to process actual data, but thread initializations and memory management operations which are essential for Flink's operation are performed.

## 5 PARALLEL DYNAMIC COMMUNITY TRACKING

To track communities across consecutive timeframes in the context of two-step methods, it is necessary to compare the communities, by estimating their similarities, in order to locate their counterparts on each timeframe. Assuming that the communities of each timeframe are available (extracted using a community detection algorithm), the typical steps required by a (serial) tracking algorithm to calculate the similarities of communities contained in sequential timeframes, are illustrated in Algorithm 1. Every community of a

[5] Flink allows the existence of multiple task managers, but to simplify the description of Flink we will refer to only one task manager.

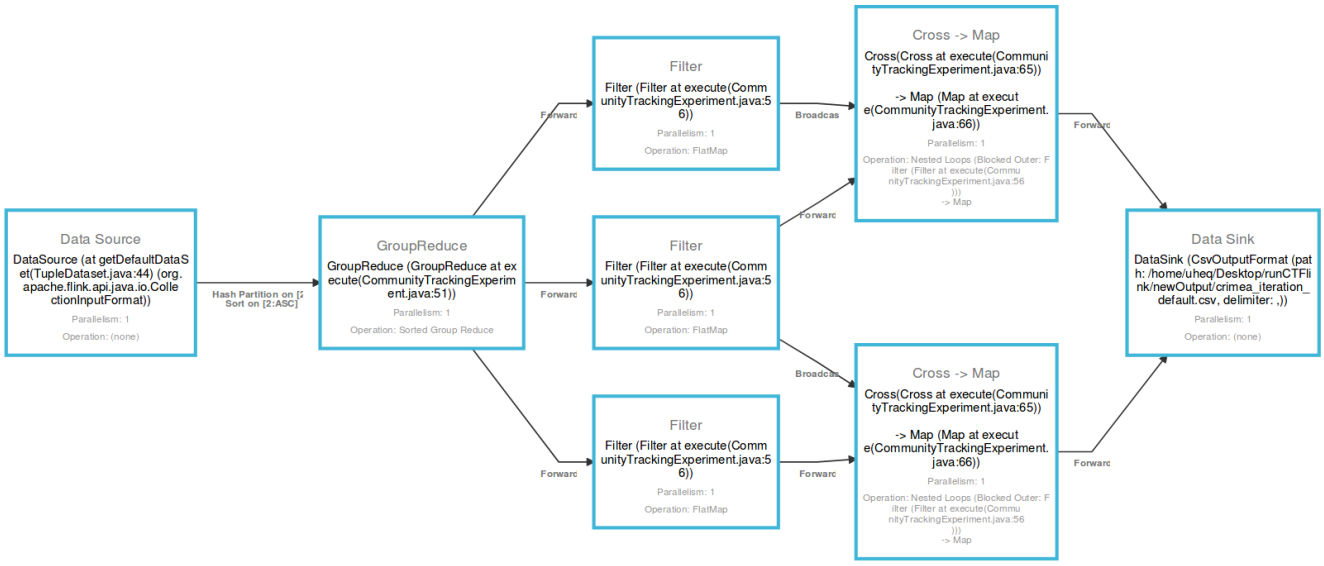



**Figure 1: The workflow of our method for an example network consisting of three timeframes as outputted by Apache Flink.**

**Algorithm 1** Serial Algorithm for Community Tracking

**Input:** $TF$ : the set of timeframes

```
JaccardSim = {}
for t = 0 to SizeOf(TF) − 2 do
    JaccardPartial = {}
    for C_t^k ∈ TF_t do
        for C_{t+1}^j ∈ TF_{t+1} do
            Result ← CalculateJaccard(V_{t,k}, V_{t+1,j})
            JaccardPartial ← JaccardPartial.append(Result)
        end for
    end for
    JaccardSim ← JaccardSim.append(JaccardPartial)
end for
```

**Output:** $JaccardSim$

given timeframe has to be compared with every other contained in the next one, and this procedure is repeated until all timeframes have been processed. The problem of calculating the similarities between two consecutive timeframes consisting of K and K' communities, respectively, has complexity $O(K \times K')$.

Due to the high number of comparisons required to perform community tracking, especially when using large datasets with many communities, it is imperative to parallelize this procedure. In order to speed up community tracking as describe in Algorithm 1, we use Apache Flink, a framework for parallel and distributed data processing. Therefore, the logic and the functions required to address the problem of dynamic community tracking have to be converted into the appropriate transformation functions and data structures that Flink is designed to handle. Note that in this work we solely focus on parallelizing the basic step involved in community tracking, i.e. the pairwise comparisons of communities. We do not study the steps of deciding which communities match and assigning evolutionary events between matched communities. The matching and the assignment of events can be performed as a post-processing step using as input the community similarities. Hence, different tracking approaches can benefit from our parallelization.

**Algorithm 2** `CalculateJaccard`

**Input:** $V_{t,k}, V_{t+1,j}$ : community vertex sets

```
VertexIntersec ← V_{t,k} ∩ V_{t+1,j}
VertexUnion ← V_{t,k} ∪ V_{t+1,j}
Jaccard ← |VertexIntersec| / |VertexUnion|
return Jaccard
```

An overview of our parallel method is described in Algorithm 3. After initializing Flink's execution environment (Step 1), we gather the vertices of each community into an array, and store these arrays into a new set ($TF_{vert}$) alongside with the their specific timeframe and community identifiers[6] (Step 2). By gathering the vertices of each community, there is a benefit of simpler data representation which helps their better manipulation inside Flink. This was done using the *GroupReduce* transformation of Apache Flink, whose functionality is similar to that of *GroupBy* statements used in SQL, combined with a reduce function. In our method, for all timeframes, we grouped the edges of each community using the community id,

[6]The timeframe identifiers are indexes from the set $\{0, 1, \ldots, T-1\}$. The community identifiers for timeframe $TF_t$ are indexes from the set $\{0, 1, \ldots, K_{t-1}\}$.

**Algorithm 3** Parallel Algorithm for Community Tracking

**Input:** $TF$ : the set of timeframes

1: Obtain Flink's execution environment
2: $TF_{vert} \leftarrow$Gather $V_{t,k}, \forall C_t^k \in TF_t, \forall t \in \{0, 1 \dots T-1\}$
3: $TF_{ids} = \{0, 1 \dots T-1\}$
4: $TF_{filt} = \{\}$
5: **for** $id \in TF_{ids}$ **do**
6: $\quad FilterResult \leftarrow Filter(TF_{vert}, id)$
7: $\quad TF_{filt} \leftarrow TF_{filt}.append(FilterResult)$
8: **end for**
9: $JaccardSim = \{\}$
10: **for** $i = 0$ to $SizeOf(TF_{filt}) - 2$ **do**
11: $\quad CrossResult \leftarrow Cross(TF_{filt,i}, TF_{filt,i+1}, \texttt{CalculateJaccard})$
12: $\quad JaccardSim \leftarrow JaccardSim.append(CrossResult)$
13: **end for**

**Output:** $JaccardSim$

and then we reduced the vertices of these communities to an array. Next (Steps 5-8), we filter each timeframe in the reduced dataset using its id. By filtering, an essential timeframe isolation is achieved since Flink's *DataSet*, an internal data structure which is used to store intermediate as well as final results, does not support indexes like those used in traditional programming languages. The result of this step, is a new set ($TF_{filt}$), in which, each element contains the communities of a particular timeframe which can be accessed using indexes. After that (Steps 10-13), we build all the pairwise combinations of the communities of two consecutive timeframes using Flink's *Cross* transformation. The Cross transformation automatically builds the community pairs and applies on each pair the user-defined `CalculateJaccard` function. The latter is illustrated in Algorithm 2 and simply computes the *Jaccard similarity* between two communities. Inside this function, the union and the intersection of the vertex arrays of each community pair are calculated, and the ratio of their cardinalities is returned as the *Jaccard* value which falls in the range $[0, 1]$. By using the *Cross* transformation we ultimately build the Cartesian product between the communities of two timeframes, which has the same complexity as a serial algorithm. The difference is that all these comparisons can now be distributed and executed efficiently over multiple CPU cores in a parallel fashion. It is Flink's responsibility to handle the details of distributing the comparisons to the available CPUs. Finally, when all calculations in the Cross transformation are completed, Flink collects and returns the results (Step 14).

A schematic representation of the workflow that Flink generates for our algorithm in the case of a social network with three timeframes is shown in Figure 1. It is obvious that the different *Filter* and *Cross* tasks are independent and can be executed in parallel by Flink. The main advantages of our method can be summarized on the following: it can exploit all available CPUs without any effort due to Flink, an alternative similarity measure can be easily incorporated by simply replacing the `CalculateJaccard` function and that the community matching and evolutionary events can be calculated at a post-processing step using the output of our algorithm.

## 6 DATASETS

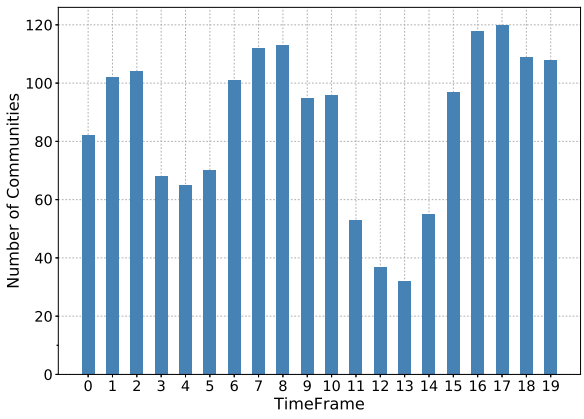
To experimentally evaluate our community tracking method, we used three different datasets containing communities, derived from real world social networks. More specifically, we used two datasets from Twitter, one discussing the Crimea crisis on the 18th of March 2014, containing 208,841 tweets, and another discussing the 2014 FIFA World Cup during May 2014, containing 1,112,875 tweets. Furthermore we used a dataset containing communities from the Mathematics Stack Exchange Q&A website, which includes 376,030 posts spanning between 2009 and 2013. Hereafter, each of the above three datasets is going to be referred as Crimea, WorldCup and MathExchange, respectively. The two Twitter datasets were split in 20 timeframes, while the MathExchange dataset in 10 timeframes. To segment datasets, tweets and posts were ordered based on their timestamp and assigned to timeframes so that each timeframe contains an equal number of tweets (posts). To detect communities on the Twitter datasets we employed the Louvain method [2] on the timeframed graphs. The vertices correspond to users who posted a tweet on a particular timeframe and an edge connects two users if either of the two mentioned the other in her tweets in that timeframe. In the MathExchange dataset each post is tagged with its subject areas (i.e. topics). To detect the communities in each timeframe, we take advantage of the topics and consider that users belong in the same community if they make posts about the same topic in a particular timeframe. Hence, each community is associated with a particular topic. Edges connect users who post an answer or comment to respond to the other's post. Note that communities that contain less than four members are considered as artifact communities and are ignored in our experiments.

The characteristics of the datasets used for our evaluation are summarized in Figure 2. The Crimea dataset illustrated in Figures 2a and 2b, contains timeframes consisting of a relatively small number of communities, containing at the most 120 communities and at the least 32 communities. Moreover, these communities, contain a small number of vertices (i.e. members), ranging on average from 15 vertices to almost 60. The WorldCup dataset contains significantly larger timeframes compared to Crimea, ranging between 175 and 327 communities as Figure 2c depicts. Furthermore, as Figure 2d shows, these communities contain considerably more vertices as the minimum mean number is 132, and the maximum is 250. Finally, the MathExchange dataset, (Figures 2e and 2f), consists of even bigger timeframes than WorldCup, as the largest contains 940 communities and the smallest 479. However, these communities have less members than WorldCup, ranging on average between 45 and 58 vertices.

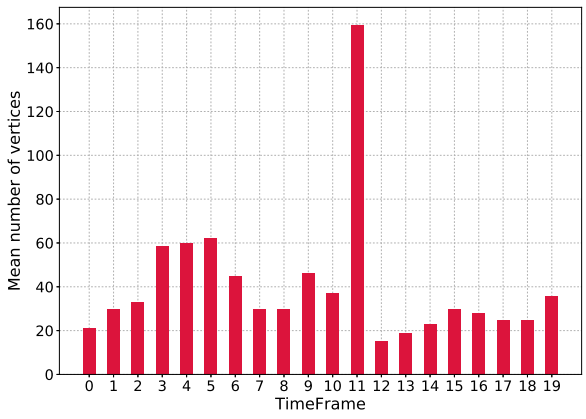
## 7 EXPERIMENTAL EVALUATION

The evaluation of our community tracking method was performed using a machine with 12 cores at 2.5GHz and 30GB of RAM memory and focuses on the execution time of the proposed parallel algorithm. Our parallel community tracking algorithm was implemented in Java using the DataSet API offered by the Apache Flink framework and is available on Github [7]. The tracking algorithm described in Section 5 was executed for the Crimea, WorldCup and MathExchange datasets and the similarities among communities

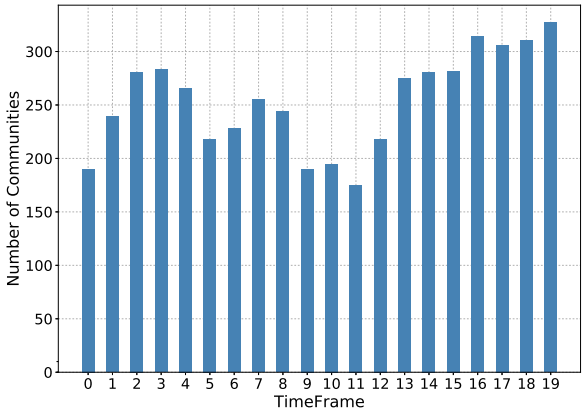
[7] https://github.com/gkech/A-Parallel-Algorithm-for-Tracking-Dynamic-Communities-Flink



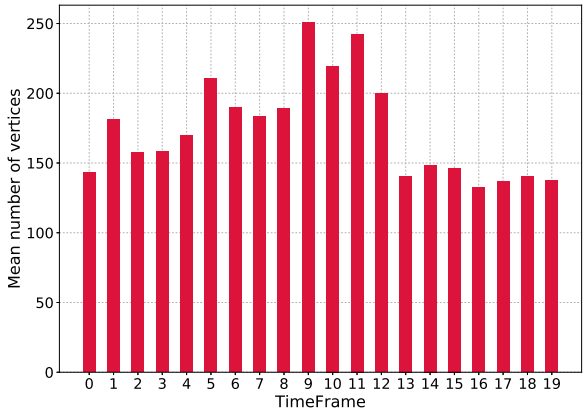

(a)



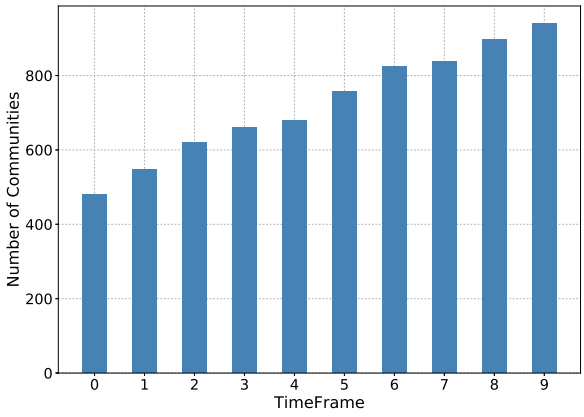

(b)



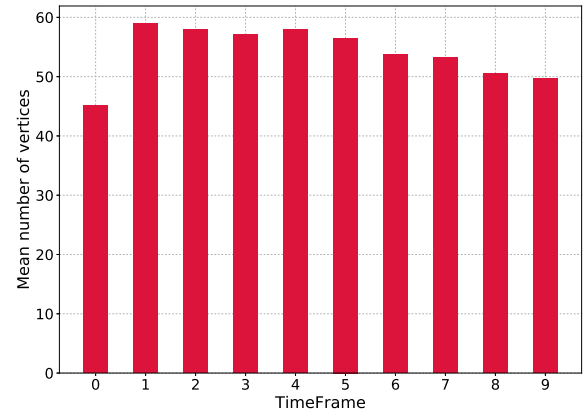

(c)




(d)

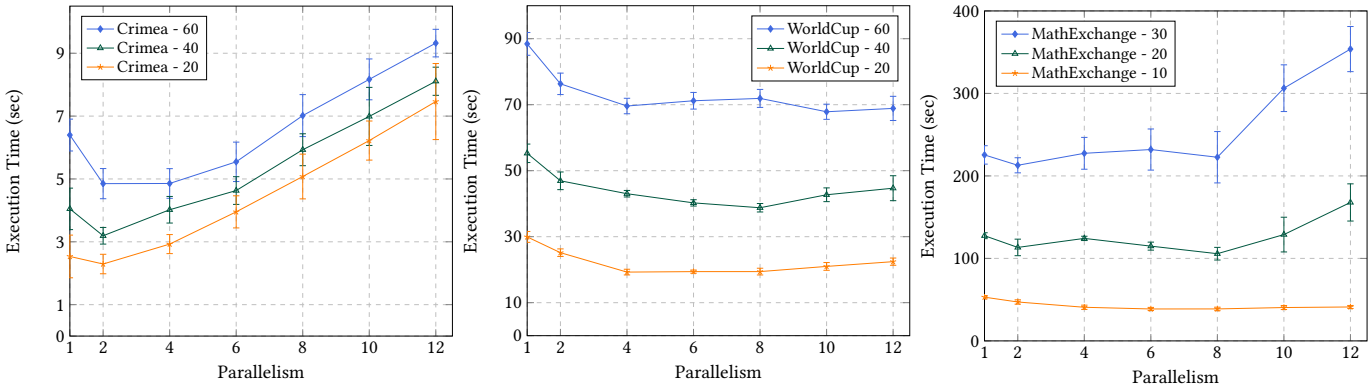



(e)




(f)

**Figure 2: The number of communities and the mean number of vertices per community in each timeframe, for each dataset. Figures 2a and 2b concern the Crimea dataset, Figures 2c and 2d the WorldCup dataset, while Figures 2e and 2f the MathExchange dataset.**




(a) Crimea dataset results (b) WorldCup dataset results (c) MathExchange dataset results

**Figure 3: Execution time of our community tracking approach for various Flink Parallelism levels on three social network datasets. The number next to the dataset name indicates the number of dataset timeframes.**

contained in consecutive timeframes were calculated. Performing experiments with real world social network datasets carrying different statistical characteristics Figure 2, enables us to test the scalability of the proposed algorithm. We performed experiments for different parallelism level configurations (i.e. number of task slots) of Apache Flink, ranging from parallelism 1 to parallelism 12 given that our machine had 12 CPUs, enabling the comprehensive evaluation of performance deviation among the different levels on different datasets and allowing us to locate the best configuration for each dataset. Remember (Section 4) that parallelism 1 (i.e. one task slot) does not imply serial execution and that Flink always exploits all available CPUs. Furthermore, to better test the scalability of our method in the context of a distributed framework like Apache Flink, we decided to conduct additional experiments by artificially enlarging each dataset. Specifically, we enlarged each dataset by repeating the existing timeframes two and three times, leading to two new datasets. For the Crimea and WorldCup datasets we got two larger datasets containing 40 and 60 timeframes respectively, while for MathExchange the two larger datasets contain 20 and 30 timeframes respectively. Notice that we repeated each experiment 25 times, and the mean execution time of these iterations is reported along with the standard deviation. The overall numerical results of our experiments are illustrated in Figure 3.

Using the Crimea dataset as input to our algorithm, as Figure 3a depicts, we can see that after an initial reduction of the execution time from parallelism 1 to 2, it is clear that as parallelism increases, the performance is reduced. The only exception to this pattern is the experiment with 60 Crimea timeframes, where from parallelism 2 to parallelism 4, the execution time remains constant. Results for higher parallelism than 4 follow the same pattern as the previous experiments. At first this result may be counterintuitive, as someone would expect to achieve better performance as the parallelism increases, but there is a simple explanation behind this behaviour. The Crimea dataset is a relatively small dataset, thus, by increasing parallelism (i.e. creating more subtasks and thus spawning more threads), the execution overhead also increases, becoming greater than the processing time needed for the actual data and leading to a higher execution time.

**Table 1: Best parallelism values from Figure 3 resulting in fastest execution.**

| Dataset | Parallelism | Exec. Time (sec) |
|---|---|---|
| Crimea 20 | 2 | 2.29 |
| Crimea 40 | 2 | 3.19 |
| Crimea 60 | 2 | 4.85 |
| WorldCup 20 | 4 | 19.27 |
| WorldCup 40 | 8 | 38.77 |
| WorldCup 60 | 10 | 67.86 |
| MathExchange 10 | 6 | 38.71 |
| MathExchange 20 | 8 | 105.71 |
| MathExchange 30 | 2 | 212.89 |

Experiments conducted using the WorldCup dataset, are presented in Figure 3b. Here, for 20 timeframes, as parallelism increases, the execution of the algorithm decreases until parallelism 4. After that, the execution time remains steady, showing a slight increase for higher parallelism levels. By increasing the number of timeframes to 40, we can observe that our method's execution time now decreases until parallelism level 8. After this point, it is slightly increasing due to the overhead created by the additional parallelism. Finally, by increasing the input to 60 timeframes the execution time, drops until parallelism 4 and then remains steady with some slight fluctuations. We observe that as the dataset becomes larger (i.e. more timeframes) a higher parallelism level is preferable. An interesting observation throughout the above experiments, is the consistency of the results as expressed by the small standard deviation.

The results of our method on the MathExchange dataset are depicted in Figure 3c. Experiments conducted for 10 timeframes, show that until parallelism 6, the execution time slightly decreases. After that, following the same behavior as the experiments reported before for WorldCup, the execution time remains relatively steady. However, for more timeframes, the general behavior is different. We observe a decrease in performance even for parallelism 4 and the standard deviation increases. Especially after parallelism 8, both

**Table 2: Execution time of GED algorithm implemented in Python.**

| Dataset | Exec. Time (sec) |
|---|---|
| Crimea 20 | 21.67 |
| WorldCup 20 | 810.45 |
| MathExchange 10 | 1670.69 |

for 20 and 30 timeframes experiments, a rapid drop in performance is evident, and the standard deviation further increases. This behavior reveals that the machine used to run our experiments has reached its limits. In fact, we observed that experiments with 20 and 30 timeframes of the MathExchange dataset, utilized fully the machine's RAM memory, which created a severe bottleneck in the execution of the experiments and introduced delays that are related to memory management issues and not to our method. These delays become more prominent for higher parallelism values, since more parallel subtasks are generated by Flink (due to more task slots being available) and, thus, more threads are spawned to handle them, creating additional memory overhead.

Until now, we discussed only the execution time variations observed for different parallelism levels of Apache Flink. Below we compare our parallel community tracking method in Apache Flink to a popular serial community tracking algorithm, in order to get a rough estimate of the speedup offered by our method. Table 2 illustrates the execution time needed to calculate evolutionary events of communities in consecutive timeframes using the GED method [3] implemented in Python[8], on the aforementioned datasets. We chose GED for this comparison as it follows a very similar approach to ours in order to track the communities, by calculating the similarity between all pairs of communities in two consecutive timeframes. Even for parallelism 1, our method achieves execution time which is 88.3% faster than the sequential GED on the Crimea dataset. In fact, for the two bigger datasets of WorldCup and MathExchange, this improvement increases to 96.3% and 96.8% respectively.

Summarizing the reported results, our experiments indicate that the parallel algorithm proposed in this work is much faster than serial algorithms for tracking dynamic communities. Moreover, the best parallelism level of Apache Flink, depends on the underlying dataset, as demonstrated in Table 1. It is clear that for larger datasets with more timeframes and communities per timeframe, higher parallelism is preferable. It is also clear from the experiments that a high parallelism level may have negative impact on performance if the dataset is not big enough (this is particularly evident on the Crimea dataset).

## 8 CONCLUSIONS

In this paper, we presented a parallel algorithm for tracking dynamic communities based on Apache Flink. Our proposed method has the ability to calculate the similarity of communities contained in consecutive timeframe within seconds, using a widely adopted similarity measure. The evaluation of our method was conducted using three real world social networks, which enabled the comprehensive evaluation of the scalability of our method with very encouraging results. Our method is highly modular, as the similarity function used to compare communities, could be easily substituted. Furthermore, as we implemented this method using Apache Flink, it has the ability to scale to larger datasets to those tested here, if more computational resources are available.

In the future we intend to extend our parallel algorithm, by incorporating the necessary steps to categorize the evolution of communities using the event labels proposed in the literature, such as community continuation, growth, shrinkage etc. Another possible future direction is the evaluation of more sophisticated similarity measures and a comparison among them in terms of event categorization accuracy using ground truth labelled datasets. Finally, we plan to devise a software suite for dynamic community tracking which is going to offer streaming capabilities in order to track evolutionary events in real time on top of the fast batch data processing proposed in this work.

[8] https://github.com/iit-Demokritos/community-Tracking-GED

## REFERENCES


[1] Sitaram Asur, Srinivasan Parthasarathy, and Duygu Ucar. 2007. An event-based framework for characterizing the evolutionary behavior of interaction graphs. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 913–921.

[2] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (2008), P10008.

[3] Piotr Bródka, Stanislaw Saganowski, and Przemyslaw Kazienko. 2013. GED: the method for group evolution discovery in social networks. *Social Netw. Analys. Mining* 3, 1 (2013), 1–14.

[4] Alexandru Ciobanu and Andreas Lommatzsch. 2016. Development of a News Recommender System based on Apache Flink. In *Working Notes of CLEF 2016 - Conference and Labs of the Evaluation forum*. 606–617.

[5] Georgios Diakidis, Despoina Karna, Dimitris Fasarakis-Hilliard, Dimitrios Vogiatzis, and George Paliouras. 2015. Predicting the Evolution of Communities in Social Networks. In *Proceedings of the 5th International Conference on Web Intelligence, Mining and Semantics, WIMS*. 1:1–1:6.

[6] Laetitia Gauvin, André Panisson, and Ciro Cattuto. 2013. Detecting the community structure and activity patterns of temporal networks: a non-negative tensor factorization approach. *CoRR* abs/1308.0723 (2013).

[7] Mark K. Goldberg, Malik Magdon-Ismail, Srinivas Nambirajan, and James Thompson. 2011. Tracking and Predicting Evolution of Social Communities. In *PASSAT/SocialCom 2011, Privacy, Security, Risk and Trust (PASSAT), 2011 IEEE Third International Conference on and 2011 IEEE Third International Conference on Social Computing (SocialCom)*. 780–783.

[8] Derek Greene, Dónal Doyle, and Padraig Cunningham. 2010. Tracking the Evolution of Communities in Dynamic Social Networks. In *International Conference on Advances in Social Networks Analysis and Mining, ASONAM*. 176–183.

[9] Vasiliki Kalavri, Jeremy Blackburn, Matteo Varvello, and Konstantina Papagiannaki. 2016. Like a Pack of Wolves: Community Structure of Web Trackers. In *Passive and Active Measurement - 17th International Conference, PAM*. 42–54.

[10] Gergely Palla, Albert-László Barabási, and Tamás Vicsek. 2007. Quantifying social group evolution. *Nature* 446, 7136 (2007), 664–667.

[11] Maria Evangelia G. Pavlopoulou, Grigorios Tzortzis, Dimitrios Vogiatzis, and George Paliouras. 2017. Predicting the evolution of communities in social networks using structural and temporal features. In *12th International Workshop on Semantic and Social Media Adaptation and Personalization, SMAP*. 40–45.

[12] Ilias Sarantopoulos. 2017. *Tracking the Evolution of Communities in Dynamic Social Networks*. Master's thesis. Athens University of Economics and Business.

[13] Etienne Gael Tajeuna, Mohamed Bouguessa, and Shengrui Wang. 2015. Tracking the evolution of community structures in time-evolving social networks. In *2015 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015*. 1–10.

[14] Zhiang Wu, Jie Cao, Guixiang Zhu, Wenpeng Yin, Alfredo Cuzzocrea, and Jin Shi. 2015. Detecting overlapping communities in poly-relational networks. *World Wide Web* 18, 5 (2015), 1373–1390.