Streaming Graph-Parallel Algorithms for Dynamic Community Detection using Spark GraphX

Yue Shi and Kai Hwang, Fellow IEEE Computer Society

University of Southern California, Los Angeles, CA, 900089
{yueshi, kaihwang} @usc.edu

Abstract— In this paper, we present a new streaming model for Graph-parallel community detection in dynamic social network using Spark GraphX tools on clouds. Two graph algorithms: SLP (streaming label propagation) and SGA (streaming genetic algorithm), are streamlined for Graph-parallel execution in the SparkX execution environment. We developed a new streaming pipeline model for GraphX-parallel execution.

Computational complexity are derived for the SLP and SGA algorithms. They compare very favorably over the conventional non-streaming label propagation (LP) and genetic algorithm (GA) graph algorithms by two to three orders of magnitude, when the social graph exceed 10 millions of edges. The improvement in processing performance scales well with the social graph size and with the cloud machine instances used in the graph-parallel execution pipeline.

Keywords— Social Networks, Graph Analytics, Community Detection, and Big Data Applications.

1. INTRODUCTION AND BACKGROUND

Community detection (CD) is a fundamentally important in social network graph analysis. This problem has been studied by many researchers in the past. Most previous CD algorithms are for identifying two disjoint communities. In reality, real-life social networks are often overlapped with some common members across multiple communities. For example, in the application of advertisement targeting in large in Facebook or Twitter, one person may belong to multiple interest communities. Thus there is growing attention on overlapping community detection where the social graphs are subject to change dynamically in time.

In Kuzmain, et al [7], a SPLA algorithm was proposed with message passing for parallel community detection. Some good speedup was observed with distributed parallel execution. However, synchronization among nodes need to be dealt with a message passing overhead increasing sharply against reduction in computation time as the graph scales. For this reason, Shi [13] have suggested a genetic algorithm (GA) with link clustering to optimize the link density. Yang and Leskovec [19] have suggested a nonnegative matrix factorization method to further improve the performance.

In all of the above methods, the input file comes in batch mode, which implies to process a static graph problem with a fixed dataset size, that can be processed one batch at a time. The Spark library has been developed to support the streaming processing of real-life datasets that can change dynamically. Also a GraphX module was developed in Spark to support graph-parallel execution. Our contribution is to combine the streaming mode with GraphX analytics for pipelined execution of graph-parallel algorithms.

We prove through computational complexity analysis that the proposed new SLP (streaming label propagation) and SGA (streaming genetic algorithm) can further enhance the performance of overlapping community detection, significantly. Our work aims to convert the existing overlapping community detection algorithms to parallel streaming versions. We utilize Spark Streaming and GraphX for overlapped community detection.

We choose to improve the GA for graph analytics by leveraging the graph property of modularity. Another Algorithm we choose to optimize is the SPLA[7], since it can apply the Pregel API to propagate the labels faster. In Section 2, we introduce the streaming GraphX execution pipeline. In Section 3, we show the parallel streaming version of our SLP and SGA algorithms for overlapped community detection on dynamic social networks. Time complexity and speedup results are reported in section 4. Finally, we elaborate on the benchmark testing of the new graph algorithms in future work.

2. Streaming GraphX Execution Environment

A. Spark Streaming and GraphX Execution

A social graph is represented by a weighted graph $G = (V, E)$, where $V$ is the vertex set and $E$ is the set of all edges. Consider $C$ communities of peers to be detected. The goal is to detect the community membership of each peer node. Each node may be member of multiple communities.

The graph-parallel streaming algorithms need to apply both Spark streaming and GraphX framework. Suppose there are $M$ machine instances provisioned for parallel execution. As shown in Fig.1, the input data leverages the Spark streaming features by which successive RDD (Resilient distributed Datasets) blocks [21] are streamed through each graph pipeline engine[5].

The input is a discretized stream of text data with all the edge source/destination and edge weights information known as the DStream. A DStream is a sequence of edge RDDs blocks with each RDD containing a small batch of data during each pipeline cycle time $t$. The edges in each edge RDD are
uniquely hashed into machine instance. Each machine instance handles almost balance workload of edges. Important parameters used in this paper are listed in Table 1.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Order of Magnitude</th>
<th>Name, Definition and Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>$O(10^2)$</td>
<td>Machine size or the number of machine instances used in the cloud</td>
</tr>
<tr>
<td>$v$</td>
<td>$(10^3)$</td>
<td>I/O rate or the number of edges loading per second.</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>$O (10 \sim 1000)$</td>
<td>Streaming rate or then number of RDD blocks processed in the graph.</td>
</tr>
<tr>
<td>$t$</td>
<td>$O(1)$ sec.</td>
<td>Graph pipeline cycle time for streaming successive RDD blocks</td>
</tr>
<tr>
<td>$C$</td>
<td>$O(10)$</td>
<td>No. of communities to be detected in the social graph</td>
</tr>
<tr>
<td>$E$</td>
<td>$O(10^6 \sim 10^7)$</td>
<td>Graph size or the number of edges in the entire social graph</td>
</tr>
<tr>
<td>$k$</td>
<td>Varies with LPA design</td>
<td>No. of iterations in running the LPA algorithm</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Varies with graph</td>
<td>Load index per machine or No. of RDD blocks per machine instance</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Varys with block design</td>
<td>Block capacity or the number of edges handled in each RDD block.</td>
</tr>
</tbody>
</table>

Each RDD block is first converted to an initial property graph in GraphX. The vertex ID identifies users. The edge block include all edge weight properties. Initially each node belongs to one home community. It affiliation with other communities can grow or decrease dynamically in time.

Figure 2 shows how to convert the input edges to initial property graph in parallel streaming mode. In the graph, there are 12 edges, 8 nodes and 2 machine instances. Suppose in each cycle time, there are 3 edges arriving. Each circle represents a node, and the number inside the circle is the node ID. The line between nodes carries the edge weight. For example, the edge weights can represent the closeness or interaction frequency between two nodes. These property graphs will be used in processing the graph algorithms in subsequent sections.

![Figure 1. Streaming GraphX-Parallel execution model](image)

![Figure 2 (a). An example property graph](image)
B. Parallel Streaming GraphX Execution Pipeline

Fig. 3 shows the pipeline for parallel streaming GraphX execution. The input is txt file, after initial processing, the txt file containing the edge and edge weight info is converted to the initial property graph in each RDD in a queue in streaming mode. The streaming is done by setting a streaming context using `ssc = new StreamingContext(conf, Seconds(t))`, `ssc` is the variable for streaming context, and `StreamingContext(conf, Seconds(t))` sets the streaming context with `conf` which sets the hardware configuration of the application and `Seconds(t)` is the time interval set for each RDD, `t` is the second set. Transform is through

And after computation for each RDD, the initial graph is converted to the community graph for each RDD, which will be discussed separately for different algorithms in the following section.
Since it is in parallel mode, the community graph in each RDD in each machine instance will exchange their community info through message sendMsg and gatherMsg function provided by GraphX Pregel API.

After exchange, the community graph will be saved and merged with the community graph computed from the next RDD.

3. The SGA and SLP Graph Algorithms

In this section, we present our new streaming genetic algorithm (SGA) and streaming label propagation (SLP) algorithms. We use flow to illustrate the streaming graph algorithms. The following sunroutine algorithms will be applied to process each RDD, plus a block merging algorithm for merging RDD blocks.

A. The streaming genetic algorithm (SGA) for processing each edge RDD block.

First, we define the fitness function to handle the population in the SGA process.

function GENETIC-ALGORITHM( Fitness-FN)

  population_initialization(n_nodes,n_population)

  for i = 0 to k_iteration do
    population_fitness()
    population_selectionandcrossover()
    population_mutation()
  endfor

Each population handles a chromosome with multiple genes corresponding to the number of peervertex. Each gene is represented by a vertex in the graph. The gene type is the community that the gene belongs to. Several gene operators are specified below:

Initialization(n_nodes,m_populations): There are n nodes. For n genes. Each node would select one of its neighbour or its own vertex ID as its initial gene type, identified by the community ID.

Population_crossover: Randomly select two populations, and exchange the gene type for the same gene to generate two new populations.

Population_mutation: Randomly select one population, and exchange the gene type for two genes from the same population to generate a new population.

For weighted graph, we use modularity to quantify the graph community structure. This can be applied to optimize the population_fitness. Consider using the GraphX subgraph API. The initial property graph would be divided to subgraphs with vertices having the same gene type. Knowing the in-degrees and out-degrees of each subgraph, the modularity could be calculated. And the initial property graph could be transformed to the following community graph.

For the vertexRDD, the ID is the detected community ID, and the property is a string concatenation of each community’s modularity, total weights inside the community and array of vertices belonging to the community. For the edgeRDD, the source and destination ID is the community ID, the property becomes concatenation of the total weights of all edges between the two communities, and an array of connecting edges with corresponding weight info between two communities.

<table>
<thead>
<tr>
<th>Vertex Table</th>
<th>Edge Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vertex Id</strong></td>
<td><strong>Property</strong></td>
</tr>
<tr>
<td>C1</td>
<td>Modularity, total degrees, Array[vertices]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 4. The community graph for Genetic Algorithm applied to each RDD block.

The above is for disjoint community detection in each RDD. For overlapping community detection, we need one more step. In the above EdgeRDD in Fig.4, for the array of the cutting edges between any two communities, the incident nodes might have multiple communities. These nodes will be added to each community to see if the modularity of that community increases.

Following is the pseudo code for the streaming algorithms. For either SGA or SLPA, the streaming algorithm mainly includes three parts, the first part is the algorithm of each RDD, the other is the intercommunication of each RDDs at the same time interval, and the third part is the serialized merging of consecutive RDDs.

For the SGA, the pseudo code for each RDD block is specified as follows:

For all RDD blocks in the Dstream:

// Convert information of edges to RDD of edges and vertex RDD
EdgesInfo => Vertex: RDD[(VertexID,VertexID)] and Edges: RDD[Edge[Src,Dst,Weight]];

Apply GA to the edges and vertex RDD
// To generate a Community subgraph
ComGraph (modularityInfo,community info);
// Use Pregel to send the following message
Use Pregel to sendMsg of ComGraph(community info)
// Exchange merging information
Output ComGraph(modularityInfo) to next RDD
Endforall

The above two algorithms are for each RDD during a time interval t in each machine instance. An additional merging algorithm is needed to join the community graph in
each RDD in one machine instance. Let \( \pi \) be the ratio of overlapped nodes to the total number of nodes in a detected community graph and \( \theta \) be the threshold to merge two communities subgraphs. The merging algorithm is given below:

**Serialized Merging Algorithm (SMA):**

For each two assigned community IDs:

If \( \pi > \theta \), two communities are assigned with the same ID, meaning merged.

Endfor

We could simply calculate the fraction of overlapped number of nodes by the GraphX join operator.

**B. Streaming Label Propagation (SLP) for each RDD.**

For each RDD the SLP works as follows:

**Step 1:** Each node is initialized with unique label, which is its vertex ID.

**Step 2:** In each iteration \( t \), for each node as a listener, its neighbors acting as speakers would pick and send a label from their label queues, with the probability positively correlated to the label occurrence frequency to the listener.

**Step 3:** Each listener would pick the most popular label receiving from their speakers and add to its label queue. Iteration \( t \) ends.

**Step 4:** Each node ends up with a label queue, and the label probability distribution is calculated accordingly. A node is determined to belong to a community if the probability is larger than a threshold.

During the execution, the vertexRDD changes as following. The vertexId is still the node ID, the property becomes an array of received communities labels and the corresponding frequencies as shown in Fig.5. And after all the steps, the community graph similar to Fig.4 is generated for each RDD.

![Vertex Table](image)

<table>
<thead>
<tr>
<th>Vertex Id</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \text{Array}[C_n F_n] )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Fig.5 The vertexRDD during LA execution**

For the SPLA, the pseudo code for each RDD is as following:

**forall RDD in the Dstream:**

//Convert information of edges to RDD of edges and vertex RDD

**EdgesInfo =>** Vertex: RDD[(VertexID,VertexID)] and Edges: RDD[Edge[Src,Dst,Weight]];

**Apply PLA with Pregel to sendMsg of vertex property among all vertices in RDD:**

// Generate community subgraph

ComGraph(community info)

**Use Pregel to sendMsg of ComGraph(communityinfo)**

// Exchange merge information in two communities;

**Output** ComGraph(modularityInfo) to next RDD block

**Endforall**

In what follows, we analyze the time complexity of the two non-streaming graph algorithms. Time complexity for original GA algorithm needs to consider two parts: The first part is the time to load the entire graph, which is approximated by \( O(E) \), and the execution time for GA was estimated as \( O(CE^2) \). The second term was found by Nicosia, et al [9]. Therefore the total time complexity for the GA without streaming is approximated as follows:

\[
\Omega(GA) = E + CE^2
\]  

Similarly, time complexity for original LPA includes the loading complexity \( O(|E|) \) and execution complexity: \( O(kV) \) [4], here we consider dense graph. Then \( O(V)=O(E) \), the above becomes \( O(KE) \). \( k \) is the number of iterations running the algorithms. The total time complexity for LPA is estimated by the following expression, approximately:

\[
\Omega(LPA) = E + kE
\]  

**4. Time Complexities of The SGA and SLP Algorithms**

Next we analyze the time complexity of the two streaming genetic algorithms: SGA and SLP.

**A. Time Complexity Analysis of SGA over GA**

Considering the pipeline in Fig3, the execution time involves three parts. Firstly, in each RDD, the community detection algorithm is applied. When GA is applied the execution. Each RDD in each machine instance has \( \beta \) edges, where \( \beta = Et/T \) is assumed. Thus the execution complexity in each RDD becomes \( C\beta^2 \).

Secondly, the communication cost among all machine instances are needed after each RDD computation for community detection. Also the message exchange among all machine instances during each time interval \( t \) is proportional to the number of edges in each RDD, which is \( \beta \).

Considering the above two parts, for each RDD the computation and message exchange time complexity is \( C\beta^2 + \beta \). And since there are \( \alpha \) RDDs in each machine instance, the total time complexity for the whole process is \( (C\beta^2 + \beta) \alpha \),
where $\alpha = T/Mt$ is assumed and $T$ is the total I/O overhead time in loading the entire graph.

The third part is the serialized merging of two continuous community graph. Since the merging occurs as soon as the community graph for each RDD is computed, the merging goes in parallel with the computation. The merging of last two RDDs takes $C$. Thus the total time complexity for SGA is $\Omega$ (SGA):

$$\Omega (\text{SGA}) = (Cp^2 + \beta)\alpha + C.$$  

$$= (C\left(\frac{Et}{T}^2 + \frac{Et}{TM} \right) + \frac{Et}{M} + C$$  

$$\cong C\left(\frac{Et}{M} + \frac{Et}{tM}\right) + C$$  

$$\cong C\left(\frac{Et}{M}\right) + C$$  

Thus the speedup of streaming parallel mode vs original genetic algorithm is:

$$S(\text{SGA}) = \frac{\Omega(\text{GA})}{\Omega(\text{SGA})} = \frac{E + CE^2}{\frac{Et^2}{M} + \frac{Et}{M} + C}$$  

$$\cong \frac{CE^2}{\frac{Et}{TM}} = M\lambda$$  

where $\lambda = \frac{T}{t}$, is the streaming rate defined in Table 1.

**B. Time Complexity Analysis of the SLP against LP Algorithms.**

Considering the pipeline in Fig 3, the execution time involves three parts. Firstly, in each RDD, the community detection algorithm is applied. When PLA is applied the execution. Each RDD in each machine instance has $\beta$ edges. Thus the execution complexity in each RDD becomes $k\beta$.

Secondly, the communication cost among all machine instances are needed after each RDD computation for community detection. Also the message exchange among all machine instances during each time interval $t$ is proportional to the number of edges in each RDD, which is $\beta$.

Considering the above two parts, for each RDD the compute and message exchange time complexity is $k\beta + \beta$. And since there are $\alpha$ RDDS in each machine instance, The total time complexity for the whole process is $(k\beta + \beta)\alpha$

The third part is the serialized merging of two continuous community graph. Since the merging occurs as soon as the community graph for each RDD is computed, the merging goes in parallel with the computation. The merging of last two RDDs takes $C$. Thus the total time complexity for SPLA is $\Omega$ (SLPA):

$$\Omega (\text{SLPA}) = (k\beta + \beta)\alpha + C.$$  

$$= \left(\frac{kEt}{T} + \frac{Et}{T}\right) \frac{T}{tM} + C$$  

$$= (k + 1)\frac{Et}{M} + C$$  

Thus the speedup of streaming parallel mode vs original genetic algorithm is:

$$S(\text{SLPA}) = \frac{\Omega(\text{LPA})}{\Omega(\text{SLPA})}$$  

$$= \frac{E + CE}{(k+1)\frac{Et}{M} + C}$$  

In the above equation, $E$, $M$ and $C$ is variable. In the above equation if we ignore the items with relatively low order of magnitude in both denominator and numerator, $S_1$ approximates as $M$.

**5. Plotted Results and Analysis**

In the complexity equations, $C$, $E$, $t$, and $M$ are variable, the I/O rate $v$ is a constant determined by HDFS (Hadoop Distributed File System) setting and the disk access time. The typical value of this I/O rate equals 10M edges/sec. In the following plots, we assume the typical values of $C$, $t$, and $M$ as in Table 1.

For example, $C$ is assume in the tens, the $t$ is set as 1 sec as specified in the Spark Streaming Programming Guide. Finally, we use hundreds of machine instances for the parameter $M$. The unit time used to express the total processing time in Figure 6 is assumed to be 1 µsec, based on common server speed used in commercial clouds.

**A. Plotted Performance Results and Discussions**

In Figure 6, we plot the processing time with increasing number of edges. Fig 6(a) shows the processing time of GA and SGA which is as following: $\Omega(\text{GA}) = E + CE^2$

$$\Omega (\text{SGA}) = C\left(\frac{Et}{M}\right) + \frac{Et}{M} + C$$

Figure 6(b) shows the processing time of LPA and SLPA, $\Omega (\text{LPA}) = (k + 1)\frac{Et}{M} + C$, $\Omega (\text{SLPA}) = E + kE$, the variable here are the $k$ and $M$, here we fix the value of $k$ as 100, and $M$ as 100.
In Fig 6(a), we see that when the graph size or the number of edges are less than 100,000, the streaming mode takes even longer processing time. However for large graphs with millions of more edges, we observe the increasing reduction of total processing time by 100’s or 1000’s times. When the number of edges exceeds 100 millions, the SGA has 1,000 times performance gain.

In Fig 6(b), we observe that SLP algorithm always has a 2 orders magnitude of reduction in processing time compared with the conventional non-streaming LPA. This gain comes from the parallel execution mode in using hundreds of machine instances.

In Fig 7, we plot the speedup of SGA and SLPA. S(SGA)= \lambda M \text{, } \lambda is the streaming factor, if we fix M as 4. S(SLPA)= \lambda M. The speedup for these two are both linear, however, the slope is different. For the SGA, apart from the parallel mode gain, it also takes advantage of the streaming pipeline.

We could observe that when the original algorithm is linear in time, the streaming mode will not work better. It will only benefit from the parallel mode. The streaming mode works better with larger data size when the original algorithm is polynomial or higher complexity.
The streaming mode works better with larger graph size. All the claimed advantages are based on theoretical complexity analysis. They need to be validated with real-life graph benchmark experiments in the future. This will be done in our continued work. A good graph benchmark suite, known as LFR [8] will be used in our future experiments.

6. CONCLUSIONS

We have proposed a new pipelined graph processing model for streaming execution of the genetic algorithm (GA) and the label propagation algorithm (LPA) for overlapped community detection in Spark GraphX environment. Through computational complexity analysis, we prove that $O(10^2)$ to $O(10^3)$ improvement in graph processing time in detection and merging of overlapped social communities. The analytical results are yet to be validated by graph benchmark experiments in future continued effort.

REFERENCES


[5] Gonzalez, Joseph; Xin, Reynold; Dave, Ankur; Crankshaw, Daniel; Franklin, Michael; Stoica, Ion (Oct 2014). "GraphX: Graph Processing in a Distributed Dataflow Framework", operating systems design and implementation


Yue Shi is currently a PhD student of Electrical Engineering at the University of Southern California. She received the B.S from Huazhong Univ. of Science and Technology in China and M.S. degree from USC. She is interested in big data analytics and integrating clouds, mobile systems, social networks and IoT (Internet of Things) for mashup applications. She can be reach by Email: Mesh Cloud: yueshi@usc.edu.

Kai Hwang is a Professor of Electrical Engineering and Computer Science, University of Southern California (USC). He received the Ph.D. from the University of California, Berkeley in 1972. Prior to joining USC in 1986, he has taught at Purdue University for 11 years. He has served as the founding Editor-in-Chief of the Journal of Parallel and Distributed Computing from 1983 to 2011.

Dr. Hwang has published 8 books and 250 scientific papers. According to Google Scholars, his work was cited over 16,000 times with an h-index of 54. His most cited book on Computer Architecture and Parallel Processing was cited more than 2,400 times and his PowerTrust (IEEE-TPDS, April 2007) paper was cited over 680 times. An IEEE Life Fellow, Hwang received Lifetime Achievement Award from IEEE Cloudcom-2012 for his pioneering contributions in the field of computer architecture, parallel, distributed and cloud computing, and cyber security.