Simulation of Streaming Graph Partitioning

Yangguang Liao, Siqi Wu, Liwei Wu
UC Davis
November 2015

1 Introduction

Since today's data set scale is huge, large-scale graph-structured computation is critical for most modern computation problem, which has led to the development of graph-parallel abstractions. There are huge amount of graph based data, for example the large web search engine Google crawls over about ten trillion links of world wide webpage. User relation in Facebook is also a typical graph structured data set with billions of friends link. Other social media network also provide valuable graph structured data, In July 2009, Twitter had over 41.7 million users with over 1.47 billion social relations. Examples of large graph datasets are not limited to the Internet and social networks, biological networks, like protein interaction networks, are of a similar size. Astronomy and planetology science has same size scale or larger astrophysical data. People characterize the essential issue of graph-based parallelism by clustering and graph partition to maximize the computation performance and penalty for cluster communication.

Typical method for dealing with large scale graph data is using reasonable partition algorithm to split the data across a large cluster of commodity machines and use parallel, distributed algorithms for the computation. This approach introduces a host of systems engineering problems of which we focus only on the problem of data layout. For graph data, this is called balanced graph partitioning. The goal is to minimize the number of cross partition edges, while keeping the number of nodes (or edges) in every partition approximately even.

There are various reasons for us to apply a good graph partition algorithm. First, graphs that we encounter and care about in practice are not random. The graph definitely reveal some nature of our real world. The edges display a great deal of locality, whether due to the vertices being geographically close in social networks, or related by topic or domain on the web. This locality gives us hope that good partitions, or at least partitions that are significantly better than random cuts, exist in real graphs. Second, inter-machine communication, even on the same local network, is substantially more expensive than inter-processor communication. Network latency is measured in microseconds while inter-process communication is measured in nanoseconds. This disparity substantially slows down processing when the network must be used. For large graphs, the data to be moved may change the communication structure, causing network links to become saturated.

With the scaling up of the system, big graph has to be cut into several pieces and then loaded into several clusters. Existing graph partitioning algorithm usually have very large cost, may be as large as future computation cost. However, since the graph need to be loaded into clusters anyway, we simulate the streaming graph partitioning, i.e. partitioning the graph at the same when the graph is loading into clusters.

2 Related work

There are a lot of related research in this topic. The most famous one, which is regarded as NP-Hard, is to cut a graph into k balanced parts, while minimize the number of edges being cut. Stanton and Kliot discuss several partition methods, and streaming orders of the graph, and their model is proved better than the partitioning method used in Spark.

3 Methodology

The algorithm is intuitive, vertices are coming in a stream, and several different methods were provided, like Balanced, Randomized Greedy, greedy EvoCut, to determine which machine should we put the new node. The streaming order like BFS, DFS or random is also considered here. In this way, we could partition the graph into several pieces when loading it into clusters. We simulate the combinations of different datasets, streaming orders, heuristics and number of clusters on our own offline system to evaluate their performance.
In this section, we show the methodology,including all 9 heuristics and 3 streaming orders separately.

Heuristics

We use P^t to refer the partition status at time t, $P^t(i)$ denotes the ith partition. Current node in the streaming at time t is denoted by v, $\Gamma(v)$ means the set of neighbors of node v. C is capacity of each partition, and —S— denotes the size of a set S. Each heuristics will give an index ind of the selecting partition.

1. Balanced

Balance algorithm will assign node v to the partition which has smallest size now, if several partitions all have smallest size, randomly choose one:

$$ind = arg \min_{i \in [k]} \{|P^t(i)|\}$$

2. Chunking

Divide the stream into chunks of size C and fill the partitions completely in order:

$$ind = \lceil t/C \rceil$$

3. Hashing

Use a hash function to calculate the index of the partition, we use:

$$H(v) = (v \bmod k) + 1$$

4. (Weighted) Deterministic Greedy

Assign v to the partition where it has the most neighbors. Here we also use a penalty function to separate nodes:

$$ind = arg\max_{i\in[k]}\left\{\left|P^t(i)\bigcap\Gamma(v)\right| w(t,i)\right\}$$

Below are three different penalty functions: unweighted, linear weighted and exponentially weighted.
$w(t,i) = 1$
$w(t,i) = 1 - \frac{|P^t(i)|}{C}$
$w(t,i) = 1 - \exp\{|P^t(i)| - C\}$

5. (Weighted) Randomized Greedy

Instead of choosing partition where has most neighbors of the current node, we use a distribution to define the possibly of each partition. Use same penalty function as before and Z is normalizing constant:

$$Pr(i) = \left|P^t(i)\bigcap\Gamma(v)\right| w(t,i)/Z$$

6. (Weighted) Triangles

Here we use below function to find the partition where neighbors of node v have most edges:

$$ind = arg\max_{i\in[k]}\left\{\frac{\left|E(P^t(i)\bigcap\Gamma(v), P^t(i)\bigcap\Gamma(v)\right|}{\binom{|P^t(i)\bigcap\Gamma(v)|}{2}} w(t,i)\right\}$$

7. Balance Big

We first set a limit to differentiate high and low degree of a node v, then we apply **Balanced** to high-degree node, and **Deterministic Greedy** to low-degree node.

Streaming Order

In some degree, streaming order will also effect the performance of heuristics a lot, we discuss three different ordering in this section: Random, BFS, DFS.
Random - This is a standard ordering in streaming literature and assumes that the vertices arrive in an order given by a random permutation of the vertices.
BFS - This ordering is generated by selecting a starting node from each connected component of the graph uniformly at random and is the result of a breadth-first search that starts at the given node. If there are multiple connected components, the component ordering is

done at random.
DFS - This ordering is identical to the BFS ordering except that depth-first search is used.

Implementation Details

In this section, we discuss details of implementation, includes language choosing, pre-processing, data structures we use, how to implement BFS and DFS.

1. Language and library choosing

We use Python to simulate the whole partition process, generally the reason is this program is not very big and complicated. Also, Python has connivent IO interface with files, and more importantly, it is also quit easy to write a script to run simulation. As for library choosing, since we only need a random pick function, so we only import library *random.*

2. Dataset Pre-Process

The datasets only provide the information of connections, i.e. the information of edges. We have to do pre-processing to convert that information to adjacent list or adjacent matrix, because the streaming consists of each nodes along with its neighbors. Since the graph is usually sparse, considering space complexity, we choose adjacent list to store the whole graph (our datasets are small enough, so a whole graph could be stored easily in a single computer).

3. Data structures

Data structures we use are shown in the following table

item data structure Adjacent list hash table BFS/DFS/Rand Order list Each Partition set Other list

We decide to use a hash map to store the adjacent list, because of O(1) accessing time, the key is index of the node while the value is all its neighbors stored in a list. Then we read the origin file line by line, i.e. an edge each time, and build the adjacent list according to the node number.

For streaming order, after traversal all the node in a given order, we store that order as a list, the value is index of each node. Because the adjacent list is stored as a hash map, we could easily find a node *v's* neighbors.

As for each partition, we choose set to store $P^t(i)$, the reason is mainly that we always need to examine a given node is in a partition or not, like *Deterministic Greedy*, *Randomized Greedy*, *Triangles.*

4. BFS and DFS traversal

BFS and DFS traversal are two basic traversal methods of a graph. For BFS, first choose a random start point, and enqueue all its neighbors, dequeue the first one, iterate this two steps until the queue is empty. For DFS just use a stack instead of a queue. What we need to consider is that the graph may not be a connected graph, so after the queue or stack is empty, if the number of currently visited node is not equal to the total number, we should randomly pick a second start point to continue the traversal. As shown in the last section, the streaming order is stored in a list.

implementation why python or library usage datastructure dfs bfs algorithm

Different streaming strategy and ordering could fit different dataset type. Our experiment is trying to find the most reasonable heuristic for specific graph type. To find out relationship between computation performance and scaling size, data features and graph properties.

Experiment dataset description simulation pre processing format transformation streaming simulation

Dataset type
The datasets are chosen from the paper and online data library SNAP. The data size would be small? since we are expected to using offline system to simulate the heuristics implementation. For each dataset, we will record its scaling size, data feature and graph properties.
Evaluation method
We will simulate each heuristic with 3 different ordering strategies on all the datasets 5 times, which could reveal an average performance for each combination.
Quality evaluation
We use following approach to evaluate heuristics quality:
1. Upper bound: RANDOM HASHING and lower bound METIS.
2. Time cost improvement.
3. Fraction of edges cut.
4. Real system implementation.
5. Page ranking implementation in spark.
6. Page ranking in offline system with small size data.

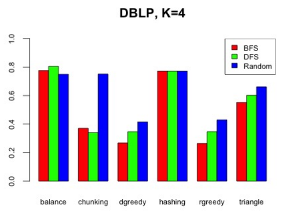
Conclusion Recall that in the experiment, we examined all the combination of datasets, heuristics and streaming orders. Each of the six heuristic was run on the same ordering of the data and we repeat the experiment for different partitions: $k = 4, k = 8, k = 12$. The performance of the heuristics is measured by the fraction of edges cut, a number between 0 and 1. To visualize the results, we summarize them in the following tables corresponding to different data sets. The first data set is the DBLP collaboration network data set. In the table below, the row names represent the six heuristics described before and the columns are first divided in three categories by the streaming orders and then further divided into three smaller categories by the partition numbers.

DBLP	BFS			DFS			Random		
	k=4	k=8	k=12	k=4	k=8	k=12	k=4	k=8	k=12
balance	0.78	0.89	0.93	0.81	0.92	0.95	0.75	0.87	0.92
chunking	0.37	0.51	0.56	0.34	0.41	0.43	0.75	0.88	0.92
dgreedy	0.27	0.38	0.41	0.35	0.43	0.46	0.42	0.50	0.53
hashing	0.77	0.89	0.93	0.77	0.89	0.93	0.77	0.89	0.93
rgreedy	0.26	0.35	0.38	0.35	0.43	0.46	0.43	0.50	0.53
triangle	0.55	0.67	0.72	0.60	0.73	0.78	0.66	0.78	0.83

Similarly, results for the other two data sets (Enron email network data set and Facebook friendship data set) are summarized as below.

Email	BFS			DFS			Random		
	k=4	k=8	k=12	k=4	k=8	k=12	k=4	k=8	k=12
balance	0.76	0.88	0.92	0.76	0.88	0.91	0.75	0.88	0.92
chunking	0.35	0.50	0.59	0.47	0.61	0.66	0.75	0.88	0.92
dgreedy	0.28	0.44	0.51	0.44	0.60	0.66	0.55	0.66	0.69
hashing	0.77	0.89	0.92	0.75	0.87	0.91	0.77	0.89	0.92
rgreedy	0.29	0.45	0.53	0.44	0.60	0.66	0.59	0.70	0.74
triangle	0.68	0.80	0.83	0.68	0.80	0.84	0.71	0.84	0.88

Facebook	BFS			DFS			Random		
	k=4	k=8	k=12	k=4	k=8	k=12	k=4	k=8	k=12
balance	0.75	0.88	0.92	0.76	0.88	0.92	0.75	0.88	0.92
chunking	0.41	0.49	0.61	0.29	0.56	0.62	0.75	0.88	0.92
dgreedy	0.38	0.31	0.46	0.30	0.56	0.63	0.33	0.55	0.48
hashing	0.75	0.88	0.92	0.75	0.88	0.92	0.75	0.88	0.92
rgreedy	0.35	0.50	0.59	0.29	0.56	0.62	0.44	0.63	0.59
triangle	0.74	0.86	0.88	0.74	0.86	0.88	0.75	0.86	0.88



DBLP, K=4
BFS
DFS
Random
0.0
0.2
0.4
0.6
0.8
1.0
balance
chunking
dgreedy
hashing
rgreedy
triangle

In the graph Rplot11 above, we can compare the fraction of edges cut for combinations of six heuristics and three streaming orders for the DBLP collaboration network data with the partition size $k = 4$. The smaller fraction of edges cut means a better performace of the algorithm.One can observe that using BFS leads to a substantial reduction of the fraction of edges cut for the dgreedy, rgreedy, and triangle heuristics. Even for the balance, chunking and hashing heuristics, using BFS performs almost as well as DFS or Random, i.e. the fraction of edges cut are not significantly larger than DFS or random. When we increase $k = 4$ to $k = 8$ and $k = 12$, we find that the observation we had for $k = 4$ still holds true. The question arises: does this phenomenon still hold even for different data sets? So we repeat the experiment for the other two data sets and the observation that BFS performs better than DFS and random across heuristics still holds true for these data sets. For the email data set, we find that using BFS leads to a substantial reduction of the fraction of edges cut for the dgreedy, rgreedy, and chunking heuristics. For the rest of the heuristics, the fraction of edges cut are more or less the same for BFS, DFS and random. For the facebook data set, we find that using BFS leads to a substantial reduction of the fraction of edges cut for the dgreedy, and chunking heuristics. For the rest of the heuristics, the fraction of edges cut are more or less the same for BFS, DFS and random. Therefore it suggests that we should use BFS all the time for whatever heuristics.

After determing that BFS is the most proper ordering to use, we still need to decide what heuristics works the best for a given data set. For the DBLP collaboration data set, we can see that dgreedy and rgreedy gives the best result for $k = 4$ and for $k = 8, 12$ the rgreedy performs slightly better than dgreedy. For the Email data set, it is also true that dgreedy and rgreedy gives the best result for $k = 4$, and for $k = 8, 12$ the dgreedy performs slightly better than rgreedy. For the Facebook data set, the dgreedy and rgreedy gives more or less same results for $k = 4$ but dgreedy performs much better than rgreedy for $k = 8, 12$. The difference are probably due to the large sample size of the Facebook data set. Obviously, we can only choose one out of dgreedy and rgreedy since all the other heuristics are not even comparable to these two in terms of performace. The dgreedy works better than rgreedy when the sample size is large, which is usually the case in practice. Therefore, overally speaking dgreedy seems to be a more reliable heuristic to use than rgreedy.

Another observation we have is that when we increase k value and keep everything else the same, the fraction of edges cut will increase no matter what heuristics and streaming orders are. But the relative advantage of BFS ordering and dgreedy/rgreedy is preserved. One can see clearly this from the line segment plot.

In conclusion, we can safely conclude that the BFS and dgreedy are the most proper streaming order and heuristic to use in practice.

result table analysis graph which heuristics applied better in specific data type, why different order of streaming for different data set ..... verify original paper our assumption

Future work We would like to find the best performance heuristic for our specific graph type and algorithm, or we could find the difference of each heuristic when applying on various data type and algorithm.