Future Generation Computer Systems 71 (2017) 32–42

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Partitioning dynamic graph asynchronously with distributed FENNEL

Zhan Shi [a], Junhao Li [a,*], Pengfei Guo [b,1], Shuangshuang Li [a], Dan Feng [a,*], Yi Su [a]

[a] Huazhong University of Science and Technology, Wuhan, Hubei 430074, China
[b] qingting.fm, Inc., Shanghai, China

HIGHLIGHTS

- Streaming graph partitioning is hard to scale because of its sequential natural.
- An asynchronous streaming graph partitioning model is proposed to improve throughput.
- Network utilization can be maximized by proposed tree-shaped map-reduce network.

ARTICLE INFO

Article history:
Received 5 July 2016
Received in revised form
31 October 2016
Accepted 7 January 2017
Available online 14 January 2017

Keywords:
Graph partitioning
Streaming
FENNEL
Asynchronous
Tree-shaped map-reduce network

ABSTRACT

Graph partitioning is important in distributed graph processing. Classical method such as METIS works well on relatively small graphs, but hard to scale for huge, dynamic graphs. Streaming graph partitioning algorithms overcome this issue by processing those graphs as streams. Among these algorithms, FENNEL achieves better edge cut ratio, even close to METIS, but consumes less memory and is significantly faster. On the other hand, graph partitioning may also benefit from distributed graph processing. However, to deploy FENNEL on a cluster, it is important to avoid quality loss and keep efficiency high. The direct implementation of this idea yields a synchronous model and a star-shaped network, which limits both scalability and efficiency. Targeting these two problems, we propose an asynchronous model, combined with a dedicated tree-shaped map-reduce network which is prevail in systems such as Apache Hadoop and Spark, to form AsyncFENNEL (asynchronous FENNEL). We theoretically prove that, the impact on partition quality brought by asynchronous model can be kept as minimal. We test AsyncFENNEL with various synthetic and real-world graphs, the comparison between synchronous and asynchronous models shows that, for streamed natural graphs, AsyncFENNEL can improve performance significantly (above 300% with 8 workers/partitions) with negligible loss on edge cut ratio. However, more worker nodes will introduce a heavier network traffic and reduce efficiency. The proposed tree-shaped map-reduce network can mitigate that impact and increase the performance in that case.

1. Introduction

At present, graph processing is applied in many fields, for example, in social networks, graph processing can either be used for security analysis [1] or finding trending topics [2], and in traditional fields such as SSSP (Single Source Shortest Paths) and paper citations, social network analysis [3], data mining [4], protein interactions [5]. The ever growing complexity and scale of various graphs are now posing a big challenge to graph processing. Currently, the indexed Web contains at least 4 billion interlinked pages [6]. On Facebook, there are over 1.65 billion monthly active users which is a 15% increase year over year [7]. Besides, every 60 s, among those friend links, 510 comments are posted, 293,000 statuses are updated, and 136,000 photos are uploaded [8]. Such an unprecedented data deluge brings us not only new opportunities and benefits, but also challenges in computing infrastructure.

Most graph computing tasks, such as Community Detection [9], Connected Components [10], Triangle Counting [11], PageRank [12], Shortest Path [13] and Graph Diameter [14], process graph data iteratively, which makes those tasks formidable to any stand-alone machine when the graph is very large. A traditional method for dealing with this problem is to divide the large graph into several smaller subgraphs, then processing it using a distributed system. These subgraphs must be balanced, so it can take advantage of parallel computing to accelerate processing.

* Corresponding authors.
E-mail addresses: zshi@hust.edu.cn (Z. Shi), allenlee@hust.edu.cn (J. Li), 1010382609@qq.com (P. Guo), doublelee@hust.edu.cn (S. Li), dfeng@hust.edu.cn (D. Feng), suyi@hust.edu.cn (Y. Su).

[1] Work done while at Huazhong University of Science and Technology.

http://dx.doi.org/10.1016/j.future.2017.01.014
0167-739X/

Although a good partition is important for processing graph efficiently, it is also hard to attain. Classical definition of balanced partition problem is to partition a graph in a way that all partitions have roughly the same vertex set size, and minimizes the edges whose two endpoints are in different partitions (cut edges). This problem was proved to be NP hard [15,16]. For years, there are many approximative algorithms been proposed and we will briefly introduce those algorithms next.

Modern distributed graph processing platforms such as MapReduce [17], Pregel [18], PEGASUS [10] and GraphLab [19] by default use hash partition to randomly partition the graph. This strategy is easy to implement, and can make the vertices of the subgraphs well-balanced, but the edge cut ratio goes up to $1 - 1/k$ (k is the number of partitions). In these systems, graph processing has to exchange messages between different partitions along the inter-partition edges. Those messages will travel through the network, which could be very costly if edge cut ratio is high.

Another major challenge for large-scale graph partitioning is how to handle dynamic graph data. The 60 s statistics of Facebook mentioned previously reveals a classical scenario of modern applications, the underlying graph is changing constantly and rapidly, and many real-world graphs share the same feature. Therefore, the processing on these graphs should be fast enough to catch up the change. Stream processing provides a viable solution to this problem. Combining with the idea of stream processing, graph partitioning becomes streaming graph partitioning, every arrived vertex needs to be immediately determined which partition it belongs to.

Given that the goal of large-scale graph partitioning is to generate better partition faster, with limited resources, we aim to use our distributed system to accelerate the graph partition. It is because a single machine has limited resources such as CPUs and memory, so it is necessary for us to use a distributed system.

The remainder of this paper is organized as follows. Section 2 provides a short study on graph partitioning and recent advances on streaming graph partitioning. In Section 3, we present our method and our analysis of partition quality and performance. The experiments results are presented in Section 4. Section 5 contains general conclusions and directions for future work.

2. Related work

For years, many researchers have proposed various graph partitioning methods. Spectral method [20,21] converts the graph into a matrix, then use eigenvectors to partition it, however this requires massive computation. Geometric method [22–24] partitions the graph based on geometric characteristics, but suffers a high edge cut ratio. Kernighan–Lin (KL) algorithm [25] starts from a vertex and add its neighbour level by level to the partition until the added vertices reach the half of the whole vertices, and its improved method FM (Fiduccia–Mattheyses) [26] provides an efficient solution to the problem of separating a network of vertices into 2 separate partitions in an effort to minimize the number of nets which contain vertices in each partition. Based on classical algorithms, modern libraries such as METIS [27] adopts a multilevel approach. The main idea is to iteratively coarsen the initial graph by merging vertices, then uncoarsen the graph iteratively with local improvement algorithms such as the KL and FM applied at each level. A multilevel KL-based algorithm [28] is presented as a fast partitioner which allows realtime deployment calculations. Above algorithms are designed for static graphs.

At present, the most common dynamic graph partitioning algorithms are hash algorithm, deterministic greedy algorithm, minimum non-neighbour algorithm. Compared to static graph partitioning algorithms, these graph partitioning algorithms use less computation and do not need the whole information of a graph to determine the partition that every incoming vertex belongs, so the partition would be faster, but the edge cut ratio is higher than the static graph partitioning algorithms. Recently, streaming graph partitioning algorithms [29–32] are proposed to handle massive graphs as data streams. Balanced edge partition [33] has emerged as a new approach to partition an input graph data for the purpose of scaling out parallel computations, which is of interest for several modern data analytic computation platforms, including platforms for iterative computations, machine learning problems, and graph databases. Furthermore, JA-BE-JA [34] is proposed to run partitioning in a distributed graph-processing system, and achieves high parallelism. PAGE [35] is a partition aware engine for parallel graph computation that equips a new message processor and a dynamic concurrency control model. Leopard [36], cooperates with FENNEL, achieves vertex reassignment and replication, can partition dynamic graphs heuristically.

FENNEL [37] was proposed to partition large-scale streaming graph with less computational complexity, which is $O(\log(k)/k)$, where k is the number of partitions or hosts. FENNEL is significantly faster than METIS, and its edge cut ratio is close to METIS. Although modern graph processing systems usually adopt parallel architecture such as map-reduce to handle big graphs [38], to use FENNEL in the same way is not easy. As a stream partitioner, FENNEL processes incoming vertices as one stream, simply running multiple processes or threads is not enough for improving parallelism. Furthermore, while doing greedy vertex assignment, every partition will calculates the cost if the vertex is assigned to this partition, the central machine node will compare the costs of every partition altogether. This structure of the system will be a star-shaped network, with a central node for making decisions, as for network, aggregated data transfer network flow will become a limitation.

3. Distributed partitioning

As we have mentioned in Section 2, among streaming graph partitioning algorithms, FENNEL has a better edge cut ratio, even catches up METIS in many cases. But to deploy in a distributed system, for scaling performance, we need to handle the problems from processing and transferring.

3.1. Processing model of FENNEL

In a distributed system, by assigning one partition to one worker node, we get a direct implementation of FENNEL processing model. For every newly arrived vertex v, a proxy node will broadcast the vertex's data, including its neighbour list to all K worker nodes. The workers will cache that data firstly, and use a greedy vertex assignment algorithm to calculate the gradient $\delta g(v, S_i) = |N(v) \cap S| - \alpha((|S| + 1)^\gamma - |S|^\gamma)$, which gives the outcome if vertex v is allocated to this worker (partition), then return the value to proxy. After proxy have gathered all the returned values $\delta g(v, S)$, it will choose $\delta_{\max} g(v, S_i)$, and broadcast the decided optimal partition i back to K workers. Then, for every worker, it will check whether it holds the optimal partition i or not. If yes, the worker takes corresponding vertex data from cache and puts in local storage. Otherwise, the worker removes corresponding vertex data from cache and puts a key–value pair $\langle v, i \rangle$ into local table for future reference.

But there are two major problems in the processing of above FENNEL model:

1. Low network efficiency caused by synchronous processing. Obviously, for every worker, the process is comprised of three phases: receiving vertex, calculating gradient and sending gradient back, which are completely synchronous. Then, the

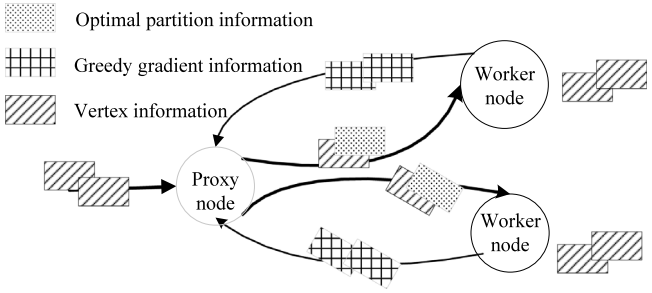


Fig. 1. Asynchronous vertex data processing model.

processing time for an incoming vertex is

$$\begin{aligned} T &= T_s + T_p + T_g \\ &= 1 * RTT + T_p + T_t \\ &= 1 * RTT + T_p + (T_p + T_g)/B. \end{aligned} \tag{1}$$

T_s is time for sending a vertex' adjacency list, T_p is for gradient calculation, T_g is for gathering gradient values, T_t is for data transmission, B is network bandwidth, RTT is round-trip time. In a generic 1 Gbps Ethernet, RTT is around 0.15 ms. The Twitter social network graph dataset from Stanford SNAP(Stanford Network Analysis Project) [39] has an average edge-vertex ratio of around 30, a few vertices have 100–200 edges connected, which is pretty high among those real-world graphs published on SNAP. We set the size of a vertex adjacency list to the maximum possible value in Twitter dataset: 200, which is 800 bytes (4 bytes per vertex ID), gradient value type is Double (8 bytes), network bandwidth is 1 Gbps, so ideal data transmission time is roughly 0.0065 ms, which is much smaller than RTT. Besides, sparser graphs will have a lower edge-vertex ratio, making worker calculation time even shorter. So, for most real-world graphs, most vertex adjacency lists are relatively small, while processing one vertex a time, the network will usually be idle, total process time per vertex is about $T = 1 * RTT$, which is dominated by RTT. This process follows algorithm logic strictly, but it is inefficient.

2. Limited scalability in star-shaped network topology. Before gradient calculation, the proxy sends a vertex and its neighbour list to K worker nodes, so K copies are spread over the network. When the network is not saturated, it does not matter. However, as K increases, all these K copies combined could easily generate a network traffic that reach or exceed the maximum bandwidth of proxy node in the centre, affecting performance. The bandwidth of a single connection between the proxy node and a worker node is:

$$\begin{aligned} Bandwidth &= \frac{datasize\ per\ \frac{RTT}{2}}{RTT\ per\ second} \\ &= \frac{800\ \text{bytes}}{(1\ \text{s}/0.05\ \text{ms})} \\ &= 5.333\ \text{MB}. \end{aligned} \tag{2}$$

For example, in an ideal 1 Gbps Ethernet, if edge-vertex ratio of input graph is $\eta = 200$, the original model of FENNEL will reach the maximum bandwidth when there are more than 23 worker nodes. Considering other factors such as protocol overhead, the maximum allowed workers would be fewer.

3.2. Asynchronous model

In this subsection, we will introduce our asynchronous model, improve its performance, and analyse the potential cost on partition quality. Our asynchronous data processing model for FENNEL is shown in Fig. 1. In this model, proxy will send vertices with their adjacency lists to all workers continuously, without waiting for the return of their gradients. Considering a concurrency of N, on the worker side, the gradients of no more than N vertices can be calculated and sent back to proxy simultaneously.

We name the original processing model as synchronous model and the our improved model as asynchronous model. In Fig. 2, ① stands for the stream of vertex adjacency lists sent from proxy, ② represents greedy gradients returned by workers. ③ is the decided partition for each vertex, and it is sent from proxy to workers as a data stream too. Every worker has a cache, which holds vertices that just completed greedy calculation and result transfer, waiting for corresponding partition decision. Among those data been transferred, apparently the vertex adjacency list ① is the biggest, greedy gradient and vertex partition decision are comparatively much smaller. Therefore, vertex adjacency list transmission is the bottleneck of the entire pipeline. So if vertex adjacency lists fill up the network, the system will meet its highest throughput, and been fully utilized. As shown in Fig. 2, assuming that 50 vertex adjacency lists will saturate the network, the required concurrency (the total of vertices that entered the system) will be the sum of the number of packets (one for each vertex) in ① and ②.

Therefore, to maximize network efficiency, the concurrency is determined by the total of vertex data packets that fill up the network and greedy gradient data packets of vertices that entered previously, which is:

$$N_{\max} = RTT * \frac{B}{S}. \tag{3}$$

B is network bandwidth, S is the size of vertex data packet. Actual vertex data packet size is usually not the same, so this value is ideal. In a star-shaped network, K workers consume K copies of vertex data, so for each worker, the actual bandwidth utilization is B/k, and the asynchronous should be

$$N_{star\text{-}\max} = RTT * \frac{B}{k}/S. \tag{4}$$

For example, if $\eta = 50$, which means the average number of edges connected to each vertex is 50, in 1 Gbps Ethernet with

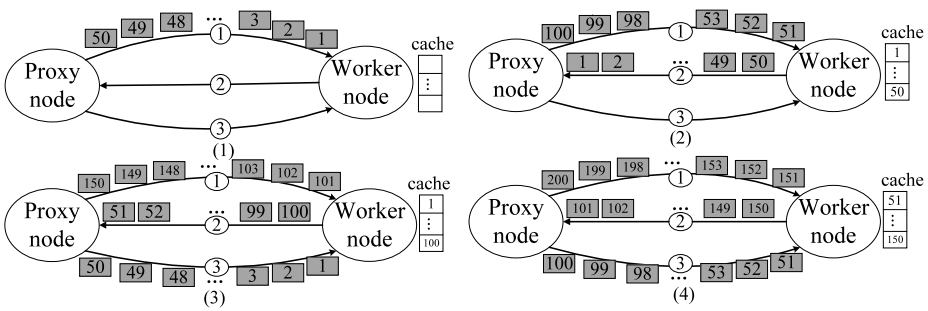


Fig. 2. Asynchronous model of fully-loaded network (50 packets).

$RTT = 0.15$ ms and one worker, data transmission delay is $T_h = 0.075$ ms, the concurrency is

$$N_{\max} = (T_h + T_h) * \frac{B}{S} = \frac{0.15\text{ ms} * 1\text{ Gbps}}{200\text{ bytes}} = 93.75 \tag{5}$$

and for K workers, the concurrency is $N(k)_{\max} = 93.75/k$, since this reaches the network bandwidth limit, higher concurrency will not benefit the efficiency. On the other hand, if the concurrency N is set within $[1, N(k)_{\max}]$, it will increase the system efficiency N times theoretically.

The basic model of FENNEL can be described as a concurrency of 1 or a little bigger, and the increased concurrency means vertex v_{i+1} can enter the system and been broadcasted to all workers before the processing of vertex v_i completed. Although this improves partitioning performance, partition quality, on the other hand, needs to be investigated. Next we will analyse the impact on $\delta g(v_{i+1}, S)$ when this system processes i and $i+1$ simultaneously. We take the classical graph500 dataset generated by Kronecker Generator (scale factor 20, which means 2^{20} vertices) as an example, this graph has about 1 million vertices and about 44.6 million edges, the edge-vertex ratio is slightly smaller than 50. While running greedy vertex assignment algorithm mentioned in Section 3.1, if the number of workers is k, then

$$\delta g(v_{i+1}, S) = |N(v_{i+1}) \cap S| - \eta\sqrt{\frac{k}{n}}((|S|+1)^{\frac{3}{2}} - |S|^{\frac{3}{2}}). \tag{6}$$

For a worker, S denotes the vertex set, and its size is $|S|$, above calculation should be done before assigning vertex v_i to any partition.

1. If we use synchronous model, then for vertex v_i, the gradient value is $\delta g_s(v_{i+1}, S') = \delta g(v_{i+1}, S')$ In which S' is the vertex set of local partition after vertex v_i is assigned.
2. If the concurrency is set to 2, then for vertex v_i, its gradient value is $\delta g_p(v_{i+1}, S) = \delta g(v_{i+1}, S)$ The difference with synchronous model is $\Delta = \delta g_p(v_{i+1}, S) - \delta g_s(v_{i+1}, S')$, assuming that asynchronously processing errors ratio is ρ, defined as

$$\rho = \frac{\Delta}{\delta g_s(v_{i+1}, S')}. \tag{7}$$

- If vertex v_i is allocated to partition S, its average probability is $1/k$, difference value is
$\Delta_1 = |(N(v_{i+1}) \cap \{v_i\})|$
$$+ \eta\sqrt{\frac{k}{n}}((|S|+2)^{\frac{3}{2}} + |S|^{\frac{3}{2}} - 2(|S|+1)^{\frac{3}{2}}). \tag{8}$$
In which the first part $|N(v_{i+1}) \cap \{v_i\}|$ indicates if vertex v_i and vertex v_{i+1} are neighbours, then the value is 1, otherwise, the value is 0. The second part $\eta\sqrt{k/n}((|S|+2)^{3/2} + |S|^{3/2} - 2(|S|+1)^{3/2})$ begins with $(2^{3/2}-2)*50*\sqrt{k/(2^{20})} \approx 0.828*\sqrt{k}/20$ when $|S| = 0$. With $|S|$ increasing, it is always greater than 0 and keeps decreasing. When $|S| \to +\infty$, it converges to 0. Particularly when $|S| = 100$, the value of second part is $0.0746 * \sqrt{k}/20$. In this case, asynchronous processing error ratio is
$$\rho = \frac{\Delta_1}{\delta g_s(v_{i+1}, S')}. \tag{9}$$
When vertices arrive randomly, the probability of vertex v_i and vertex v_{i+1} are neighbours is η/n, for the graph500 dataset used here, $\eta = 50$, $n = 1\,000\,000$, the error ratio is
$$\rho_n < \frac{1.1}{|N(v_{i+1}) \cap S'| - \frac{1}{10}((|S'|+1)^{\frac{3}{2}} - |S'|^{\frac{3}{2}})}. \tag{10}$$
The expectation of error ratio is
$$E\rho_{sn} = \frac{1}{k} * \frac{\eta}{n} * \rho = \frac{\rho}{80\,000} < \frac{0.00001375}{|N(v_{i+1}) \cap S'| - \frac{1}{10}((|S'|+1)^{\frac{3}{2}} - |S'|^{\frac{3}{2}})}. \tag{11}$$
If vertex v_i and vertex v_{i+1} are not neighbours, the probability is $1 - \eta/n$ (vertices arrive randomly), for the same graph500 dataset, the expectation of error ratio is
$$E\rho_{snn} = \frac{1}{k} * \left(1 - \frac{\eta}{n}\right) * \rho < \frac{1}{4}\rho = \frac{0.025((|S|+2)^{\frac{3}{2}} + |S|^{\frac{3}{2}} - 2(|S|+1)^{\frac{3}{2}})}{|N(v_{i+1}) \cap S'| - \frac{1}{10}((|S'|+1)^{\frac{3}{2}} - |S'|^{\frac{3}{2}})}. \tag{12}$$
When $|S| \geq 100$ and
$$E\rho_{nn} \leq \frac{0.001865}{|N(v_{i+1}) \cap S'| - \frac{1}{10}((|S'|+1)^{\frac{3}{2}} - |S'|^{\frac{3}{2}})}.$$
- If the vertex v_i is not allocated to local partition S, its average probability is $(k-1)/k$, then the difference is $\Delta_2 = 0$, and the asynchronously processing error ratio is
$$\rho_0 = \frac{\Delta_2}{\delta g_s(v_{i+1}, S')} = 0. \tag{13}$$

Based on above analysis, we can see that, when $|S| \geq 100$, the expectation of asynchronously processing error ratio is

$$E\rho = E\rho_{sn} + E\rho_{snn} + E\rho_0 < \frac{0.00187875}{|N(v_{i+1}) \cap S'| - \frac{1}{10}((|S'|+1)^{\frac{3}{2}} - |S'|^{\frac{3}{2}})}. \tag{14}$$

In which $S' = S \cap \{v_i\}$, $|S'| = |S| + 1$.

If vertices arrive randomly, then $E(|N(v_{i+1}) \cap S'|) = \eta(|S|+1)/n$, when balance factor $v = 1.1$ and $k = 4$, $|S|_{\max} = n * v/k = 275\,000$. While $|S| \in [100, 275\,000]$, from Eq. (14), we obtain $E(|E\rho|) < 0.000512\%$.

We can conclude that, when the vertices arrive randomly at the asynchronous of 2, the partitioning quality will almost be unaffected.

3.3. Tree-shaped map-reduce network

Based on previous analysis, we can conclude that, in our improved model, concurrency is mainly limited by network bandwidth. Suppose the total of replicated vertex data been transferred in the network is R, in previous Sections 3.1 and 3.2, the processing model of FENNEL works as a star-shaped network, in which $R = K$, $B_p \propto N * R$, where K is partition amount, and N is concurrency. Obviously, the bandwidth consumption B_p can be reduced by decreasing N and R, but decreasing N will lower the concurrency. This problem exists in star-shaped network, a tree-shaped map-reduce network (Fig. 3) is then proposed here to fix it.

In this model, the tree is an H-ary tree, in which each node has no more than H children, and proxy is the root. Proxy receives new vertices with their adjacency lists and sends to its child worker nodes. Workers will then run FENNEL greedy calculation on those vertices, and at the same time, forward those vertices to their respective child worker nodes. In this way, vertex adjacency lists will eventually be spread over the tree. For every worker, when calculation is done, the results will be sent back to proxy directly, since those packets are much smaller. According to gathered gradient values of every vertex, proxy uses the maximum one to decide the partition for the vertex, then passes the decided partition id down along the tree. In this tree-shaped map-reduce network, the maximum number of copies that the proxy and all

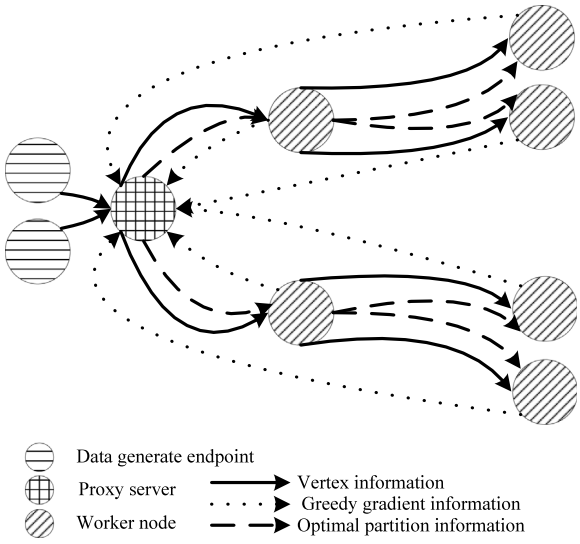


Fig. 3. Tree-shaped map-reduce network.

the forwarding workers send is up to H instead of K. Since H can be much smaller than K, this will raise the bar on maximum allowed concurrency. But this model will also increase the RTT, which is determined by the height of tree, thus may lower the efficiency.

When $H = 2$ (binary tree), suppose there are K workers, in an 1 Gbps ethernet, $RTT = 0.15$ ms and $\eta = 50$, if the network is star-shaped, transmission delay is $T_h = 0.075$ ms, and the data amount $D = (T_h + T_h) * B = 18\,550$ bytes if bandwidth B is saturated, and single data packet size is $S = 200$, based on $D = S * N * K$, we have $N = 93.75/K$. Increase the K will decrease the concurrency, when $K = 93$, the concurrency will be reduced to 1 or a little bigger, degraded to synchronous model.

We name the vertex data flow that travelled along all the branches to every leaf as "download network". It is obvious that if the download network to lower level worker nodes is fully utilized, then the download network of upper level nodes will also be fully utilized, then the whole system will has the same efficiency with an ideal star-shaped network (unlimited bandwidth) of the same size. So we only need to make the download network of lowest level (leaf) worker fully utilized. In tree-shaped map-reduce network, the maximum transmission length is $L_{\max} = \lfloor \log_2(K+1) \rfloor$, max time delay from proxy node to work nodes is $T_{\max} = L_{\max} * T_h$, so we need to increase the concurrency N to fill up the lower level worker nodes, and the max concurrency is

$$\begin{aligned} N_{\max} &= (T_{\max} + T_h) * \frac{1}{2} B/S \\ &= 23.4375 * (\lfloor \log_2(K+1) \rfloor). \end{aligned} \tag{15}$$

If we evaluate the system as a whole, the maximum concurrency from proxy to leaf workers is $23.4375 * 2$, $23.4375 * 3$, $\ldots$, $23.4375 * (\lfloor \log_2(K+1) \rfloor)$, and the concurrency of every node must be the same. In order to maximize the overall bandwidth utilization and concurrency, we need to increase the concurrency of upper nodes, make it aligned with the bottom nodes, that is, the concurrency of the whole system should be $23.4375 * (\lfloor \log_2(K+1) \rfloor)$ to achieve the highest efficiency. In this way, the upper nodes do not need to wait for the completion of their child nodes, thus the system efficiency will not be affected by the number of worker nodes. We name the resulting system AsyncFENNEL (Asynchronous FENNEL). Moreover, in AsyncFENNEL, we need to improve overall system concurrency, and may impact on the accuracy of FENNEL, which will be covered in tests.

Table 1
Experimental environment.

OS	RedHat 5.3 Linux 2.6.18
MEMORY	12 GB
DISK	296 GB
NETWORK	Full-duplex Gigabit Ethernet
CPU	Xeon (r) CPU X5560 @ 2.80 GHz

Table 2
Datasets used in experiments.

Dataset G	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	AvgDeg	DegDist
graph500	1M	48M	48	Power-law
er20-8	1M	8M	8	Poisson
er20-16	1M	16M	16	Poisson
er20-24	1M	24M	24	Poisson
patentcite	3.8M	16.5M	4.4	Power-law
hudong-int.	2M	15M	7.5	Power-law
SOC-pokec	1.6M	30M	18.8	Power-law

4. Evaluation

We use both synthetic graphs and real-world social network datasets to evaluate AsyncFENNEL, on partition quality, concurrency, network efficiency and different topologies.

There are four synthetic datasets, and three of them has the same type with different parameters. graph500 is a dataset from graph500 benchmark (2^{20} vertices, R-MAT parameter: $A = 0.57, B = 0.19, C = 0.19, D = 0.05$, edge-vertex ratio: 48). er20-8 is a synthetic Erdos–Renyi graph, and has 2^{20} vertices with a edge-vertex ratio of 8, and er20-16 has 2^{20} vertices with a edge-vertex ratio of 16, er20-24 has 2^{20} vertices with a edge-vertex ratio of 24.

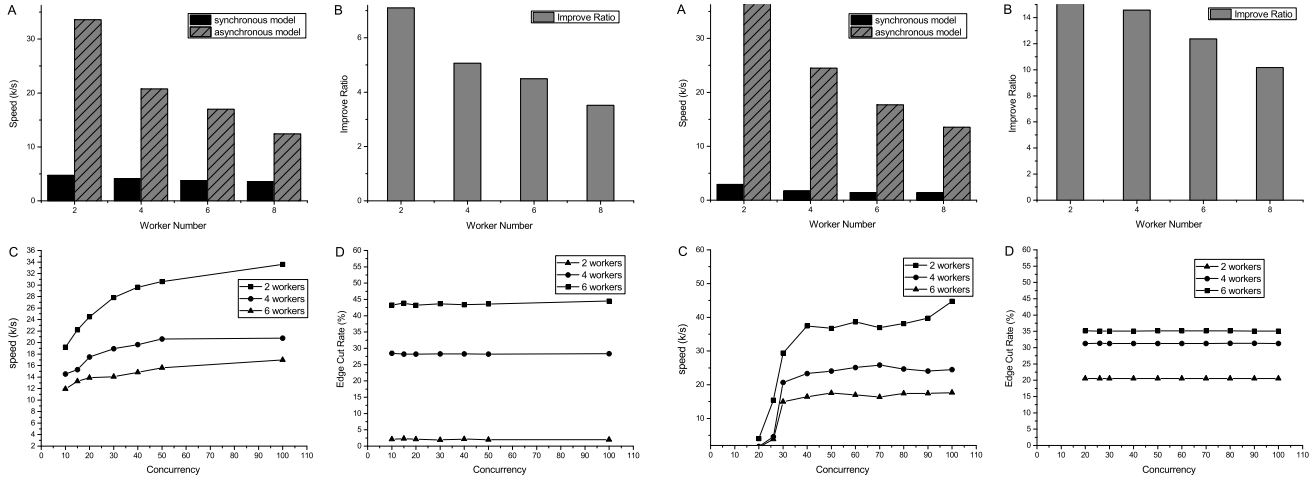
There are also three real-world dataset. SOC-pokec is a dataset from SNAP and KONECT [40], this is the friendship network from the Slovak social network Pokec, nodes are users of Pokec and directed edges represent friendships, it has 1,632,803 vertices and 30,622,564 edges and a average degree of 37.509. patentcite [41] is also a dataset from KONECT, it is the patent citation graph, it has 3,774,768 vertices (patents) and 16,518,947 edges (citations), vertices-edges ratio is 8.7523. hudong-internallink [42] is the internal link graph from hudong encyclopedia with 1,984,484 vertices (articles) and 14,869,484 edges (hyperlinks), its vertices-edges ratio is 14.986. Besides, we also use BFS-sorted graph dataset to evaluate the impact on partition quality caused by the locality of input graph stream.

4.1. Asynchronous model test

In this part, we have 2 test cases about the asynchronous model and original synchronous model. One for performance comparison, the other for showing the impact on partition quality caused by increased concurrency. 4 synthetic datasets (graph500, er20-8, er20-16 and er20-24) and 3 real-world datasets (patentcite, hudong-internallink and SOC-pokec) with different properties are used during tests (see Table 2), all real-world datasets are hosted by Stanford Network Analysis Project (SNAP) and KONECT. Our experimental environment is listed in Table 1.

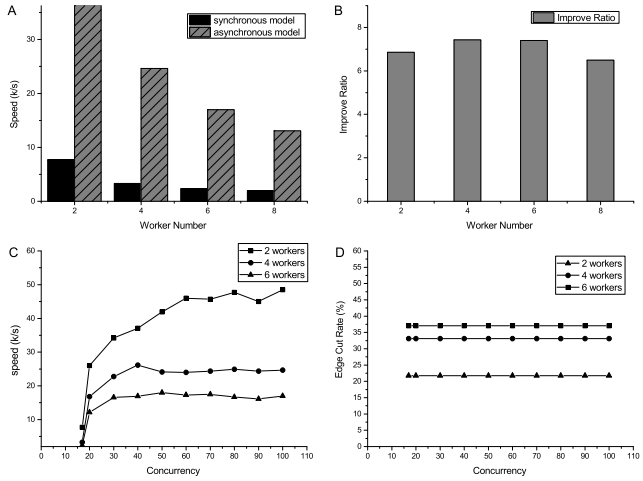
Fig. 4 is the asynchronous model tests on synthetic and real-world datasets, Fig. 4(a)–(d) are tests on synthetic datasets and Fig. 4(e)–(g) are tests on real-world datasets.

Figure A of each sub figure shows the partitioning speed while running {2, 4, 6, 8} worker nodes, the concurrency is set to 100 in asynchronous model, and the concurrency is set about 20 in synchronous model, partitioning speed is represented as processed vertex number per second, with a unit of k/s (Kilo-Vertices/Second), we can see the partitioning speed in

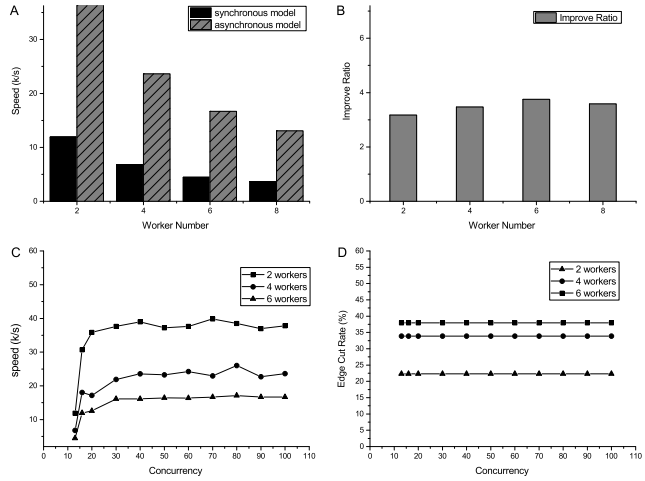


(a) graph500.

(b) er20-8.



(c) er20-16.



(d) er20-24.

Fig. 4. Asynchronous tests on three synthetic and real-world datasets.

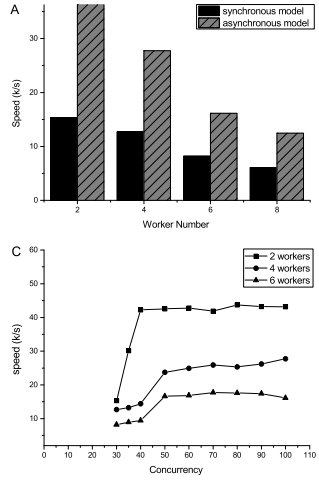
asynchronous model is significantly faster than the original synchronous model. Figure B of each sub figure shows the corresponding speed up ratio, usually the performance gain from asynchronous model gradually decreases as the number of the worker nodes increases, which implies the network traffic is growing. Figure C of each sub figure shows how partitioning speed change with increasing concurrency: {20, . . . , 30, . . . , 50, . . . , 100}, there are 3 different cases which stand for different worker total {2, 4, 6} respectively. We can see that, when concurrency is relatively low, the speed becomes higher with the concurrency increasing, and while the concurrency reaches a certain degree, the speed increase diminishes. According to the analysis in Section 3.2, asynchronous model benefits FENNEL by exploiting network utilization. In this model, worker total decides vertex copies been sent over the network. When network is not fully utilized, worker total or number of vertex copies will not be an issue, but either worker total or concurrency increases above a certain level, the network will then be saturated, and thus cannot carry more vertex data in one *RTT*, lowering the network efficiency. As a result, the partitioning speed can hardly increase when concurrency reaches 100, regardless of the worker node total. In the case of 2 worker nodes, the improvement is the most significant, which also means more worker nodes actually bring more network congestion in this system configuration, rather than improving performance.

Figure D of each sub figure shows the impact on partition edge cut ratio by concurrency {1, 2, 3, . . . , 50, 100} and worker (partition) total {2, 4, 6}.

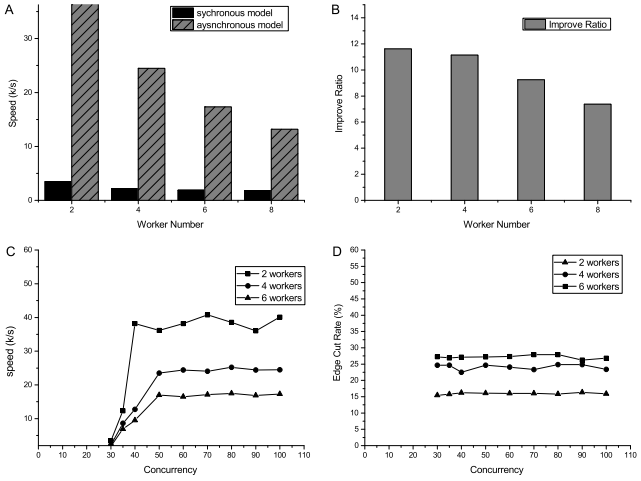
When concurrency is set to 1, it stands for the original synchronous model. Section 3.2 theoretically proves that, while the vertices in graph stream arrive randomly, if the concurrency is increased from 1 to 2, thus working asynchronously, the edge cut ratio will be almost unaffected. The test result shows that, when there are {2, 4, 6} workers (partitions), the edge cut ratio is almost unaffected by concurrency.

It is worthy to note that synthetic graphs like graph500 and real-world graphs like SOC-pokec have different properties, which leads to differences between test results. graph500 dataset is generated by classical Kronecker generator, which is based on random number sequence with permutation distribution parameters [43]. Since the sequence is purely random, it is very unlikely for consecutive vertices to be neighbours, and vertex degrees are random too. The distribution of adjacent vertices is shown in Fig. 5, which is uniformly distributed. Besides, the edge-vertex ratio is $\eta = 48$ in graph500 dataset, not too small for vertex data transferring over generic network, so this actually helps network utilization.

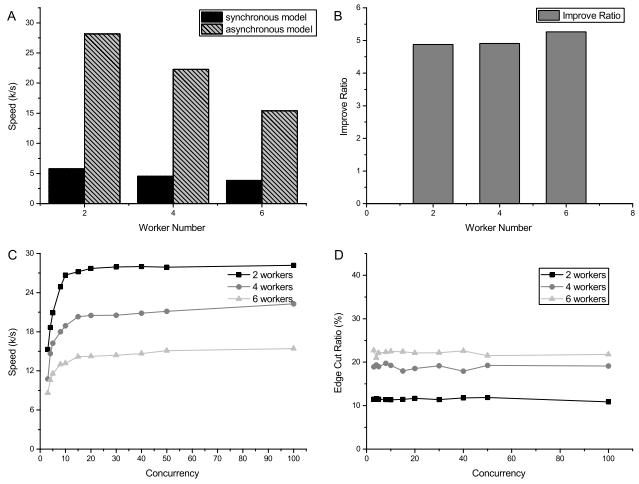
SOC-pokec dataset is gathered from a real-world social network, an important feature for social networks is called preferential attachment, that the user joined earlier usually owns more



(e) patentcite.



(f) hudong-internallink.



(g) SOC-pokec.

Fig. 4. (*continued*)

neighbours than users joined later. Fig. 6 shows the average adjacent vertices of SOC-pokec data, the size of sliding window is 100. As the window sliding from left to right, simulating the stream, the average number of neighbours decreases continuously. So, unlike the graph500 dataset, vertices with higher degree will gather around the beginning of the graph stream, and vertices with lower degree will be left behind, creating an unbalanced data packet flow. Apparently those smaller vertex data packets will decrease network utilization, so when there are more workers, the speed up ratio keeps steady.

Because the two datasets have different characteristics, so we can see the improvements in two tests, including speed up and edge cut ratio are different. In theory, the SOC-pokec should have a higher speed up than graph500 because of its low value η, but because the SOC-pokec data distribution is not balanced, which makes the actual network utilization gradually declined and eventually be slower than graph500. In addition, due to the unbalanced feature of SOC-pokec data, which makes its improved speed ratio in Fig. 4(g)-B not decreases like shown in Fig. 4(a)-B.

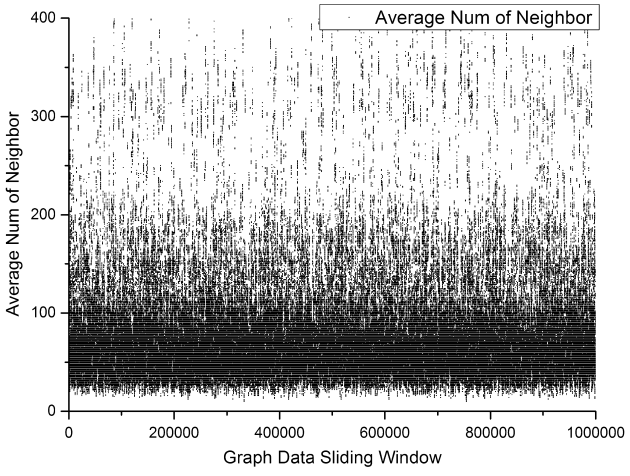


Fig. 5. Average number of neighbours in graph500 dataset (sliding window size: 100).

4.2. Tree-shaped map-reduce network test

According to Section 3.3, the proposed tree-shaped map-reduce network is able to support more workers (partitions) at the same concurrency setting and bandwidth limit, maximizing network utilization, but at the cost of increased round-trip time. According to previous tests in Section 4.1, the partitioning speed becomes the highest when the number of workers is 2, more workers (4, 6) will actually lower the performance. In this part, we will evaluate the tree-shaped map-reduce network in different configurations,

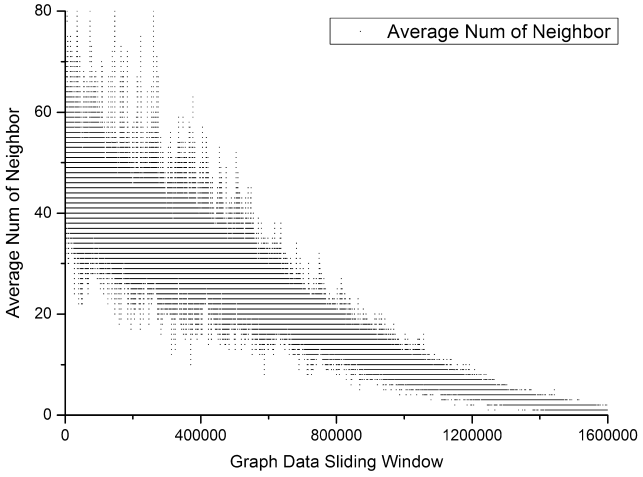


Fig. 6. Average number of neighbours in SOC-pokec dataset (sliding window size: 100).

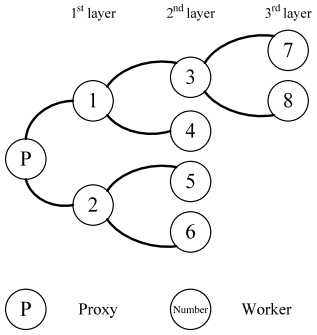


Fig. 7. Binary-tree map-reduce network with 8 workers in 3 layers.

and investigate how to handle the performance problem caused by more workers, with a correct-configured tree. This will be demonstrated by testing a binary tree. Since previous results show that, for 2 workers, a concurrency of 100 will push the speed to upper limit, so in this tree, the concurrency N of the 1st layer will be set to 100, a layer deeper means a higher RTT, N will be increased by a step value to fill the child/leaf workers, the step value is set to {0, 50, 100, 200} respectively. {2, 3, 4, 5, 6, 7, 8} workers are tested, to form a tree with 1–3 layers, we use graph500 datasets and it is shown in Fig. 7.

We can see from Fig. 8, as for original star-shaped model with $N = 100$, the partitioning time simply increase linearly with more workers. In this case, a concurrency of 100 actually saturates the network connection of proxy, additional workers will then suffer from increased latency. When $N = 50$, $step = 0$, begin with 2 workers and an under-utilized network, the partitioning time of tree network is slightly higher than previous case, and still increases with worker total, but not as fast as original model, the tree network surpasses original model when there are 4 workers. Then for $N = 100$, $step = 0$, the tree network will begin with 2 workers and a fully utilized network, while the worker total increasing, the partitioning time still rises, but slowly, from 4 to 8 workers, the change is pretty small. When $N = 100$, $step = \{50, 100, 200\}$, from 2 to 4 workers, partitioning time increases as the previous case ($N = 100$, $step = 0$). Beyond that, when worker total is $\{4, \ldots, 8\}$, partitioning time basically remain stable, and lower than all above cases.

We use another three synthetic graph and two real-world graph to test ours tree-shaped map-reduce network, from Fig. 9, the five lines indicate that with the increasing of the workers, that is, the

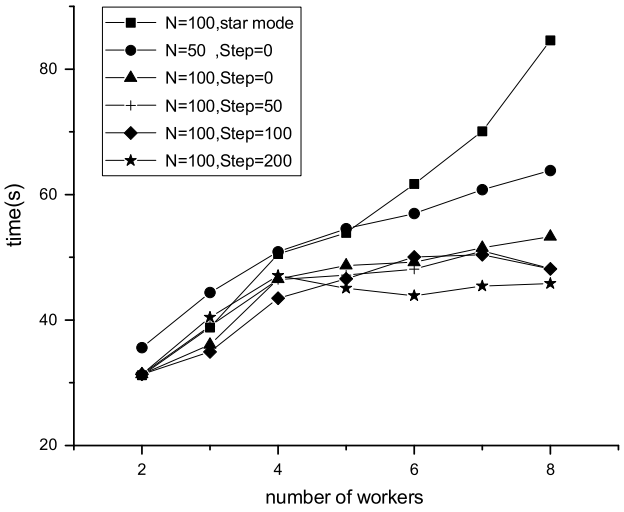


Fig. 8. graph500 partitioning time using different networks (initial concurrency: {50, 100}, step: {0, 50, 100, 200}, topology: {star, tree}).

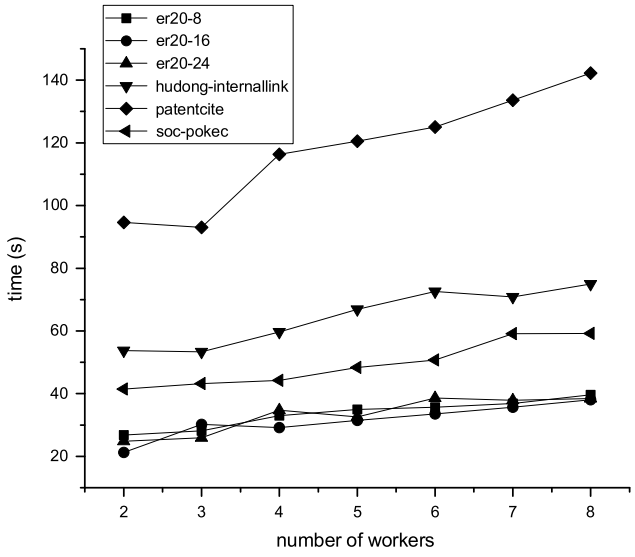


Fig. 9. Datasets partitioning time with concurrency of 100 and a step 0.

tree network is more deep, the graph partitioning speed is slower, this show the same result with Fig. 8.

From the results, tree network can improve overall network utilization, thus improving partitioning performance, especially when more workers or partitions required, as analysed in Section 3.3. The improvement is achieved by adjusting a few parameters, including initial (1st layer) concurrency N and $step$, according to network characteristics such as available bandwidth and latency etc., however, those factors could be complex. In general, leaf workers have a higher RTT, while filling the leaf workers up with more asynchronous vertex data, the efficiency can thus be improved. The same to all different layers in this tree network, with a well-chosen $step$. the well-chosen $step$ is the concurrency difference between two layer of the tree network.

4.3. Graph sorted in BFS order

In Section 3.2, partition quality (edge cut ratio) is proved to be nearly unaffected by a concurrency of 2, on the premise of random vertex arrival. In Section 4.1, the test results reveal that, even higher concurrency is allowed, for not affecting partition quality, under the same premise. However, in some cases, the graph

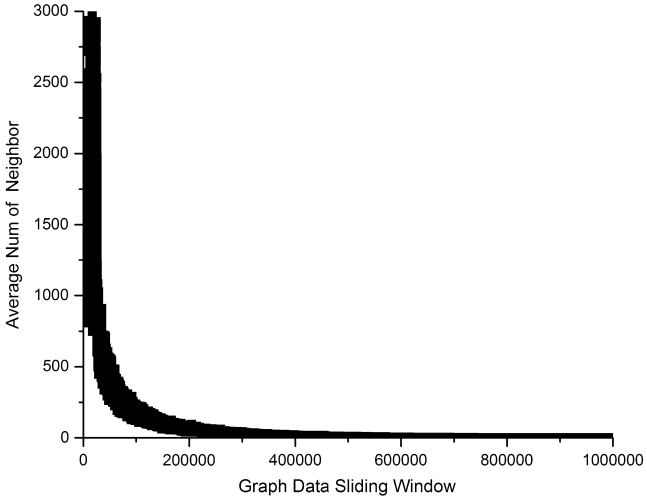


Fig. 10. Average number of neighbours in BFS-sorted graph500 dataset (sliding window size: 100).

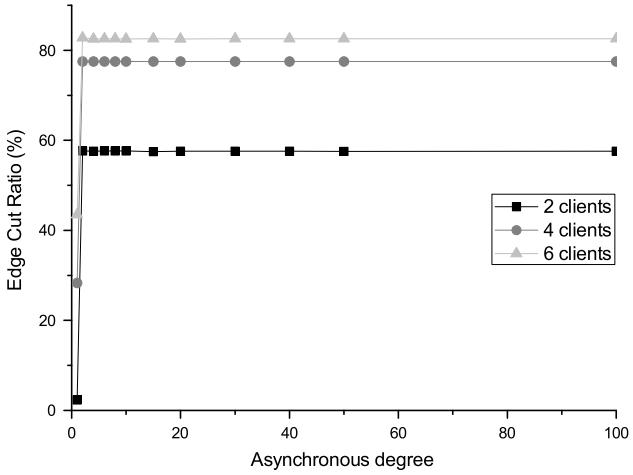


Fig. 11. Impact on edge cut ratio while partitioning graph500 dataset that sorted in BFS sequence.

stream may not arrive randomly. For example, some graphs are pre-processed for a better storage layout, because sometimes an increased locality can optimize the access to external storage, but it may also bring negative effects to partition quality. In other words, the probability of consecutively arrived vertices are neighbours, could be the fundamental cause of affecting partition quality, the higher the probability goes the higher the edge cut ratio becomes. Here, we will sort graph500 dataset in BFS sequence, to build up locality in this dataset, and then use the {2, 4, 6} workers with {1, 2, 3, . . . , 50, 100} concurrency to repeat the tests on AsyncFENNEL, try to find out the outcome.

Fig. 10 shows the average number of neighbours in BFS-sorted graph500 dataset, the size of sliding window is 100. Back to Fig. 5, the same statistic of original graph500 dataset is uniformly distributed. But in BFS-sorted graph500 dataset, the average number of neighbours is strongly polarized, vertices at the beginning tend to have much more neighbours, and vertices by the end have much fewer neighbours. The dataset shows a strong locality after been sorted in BFS sequence. While inputting the sorted graph as a data stream, it will begin with strongly related vertices that most of them are neighbours, followed by a long tail comprised of many low-degree vertices. Based on previous analysis, the former part of the stream will impact partition quality, and most edge cut will be contributed by this part, it will also bring a heavy traffic to network, since the vertex degree is pretty high. For similar reasons, the latter part of the stream will lower network efficiency, and pose little change to edge cut.

Fig. 11 shows the impact on edge cut ratio while partitioning BFS-sorted graph500 dataset. The edge cut ratio is significantly higher than that in Fig. 4(g)-D, which is tested in the same environment with the only exception that the input graph is not sorted in BFS sequence. For 2 workers, edge cut ratio increases from 2% to 57%. For 4 workers, edge cut ratio increases from 28% to 77%. For 6 workers, edge cut ratio increases from 43% to 82%. In addition, all 3 test cases (2, 4, 6 workers) show a sharp increase when concurrency goes above 1. After that, when concurrency $N \geq 2$, the edge cut ratios become steady, and much higher than those in original graph500 tests. Apparently, for graph sorted in BFS order, the vertex sequence of input graph stream becomes a serious problem to asynchronous model. The fundamental cause appears to be the strong locality, which brings a high probability of consecutive neighbouring vertices, the unbalanced network traffic is another negative effect.

5. Conclusions and future work

For partitioning and processing massive graphs, it is nature to run in a distributed system. For handling dynamic graphs, or simply partitioning with higher performance, streaming graph partitioning can be applied. But the combining of the two is not easy. In this work, we take a closer look at the processing model of FENNEL, a state-of-art streaming graph partitioning system. We first propose the asynchronous model, prove the viability of doing so, based on the performance gain and negligible quality loss. We then focus on the derived problem of network I/O congestion, propose a tree-shaped map-reduce network to mitigate the impact of using more workers and maximize the throughput. We implement above two improvements as AsyncFENNEL, use both synthetic and real-world graphs to evaluate, verify both asynchronous model and tree network under various circumstances.

Massive applications and datasets demand new processing architectures, a tree shaped map-reduce network reveals some potential on improving performance, but it is definitely not the only way to serve our purpose, we plan to push our experiment further to more real-world datasets and system topologies. In addition, for graph data with strong locality, such as graphs sorted in BFS order, we will also look into this case to improve its partition quality.

Acknowledgements

This work is supported by the National High Technology Research and Development Program (863 Program) No. 2015AA015301, No. 2013AA013203, No. 2015AA016701, NSFC No. 61173043, No. 61303046, No. 61472153, State Key Laboratory of Computer Architecture, No. CARCH201505 and Wuhan Applied Basic Research Project (No. 2015010101010004). This work is also supported by Key Laboratory of Information Storage System, Ministry of Education, China No. 81.

References

[1] W. Jiang, G. Wang, J. Wu, Generating trusted graphs for trust evaluation in online social networks, Future Gener. Comput. Syst. 31 (2012) 48–58. http://dx.doi.org/10.1016/j.future.2012.06.010.

[2] T. Hachaj, M.R. Ogiela, Clustering of trending topics in microblogging posts: A graph-based approach, Future Gener. Comput. Syst. (2016). http://dx.doi.org/10.1016/j.future.2016.04.009.

[3] L. Quick, P. Wilkinson, D. Hardcastle, Using pregel-like large scale graph processing frameworks for social network analysis, in: Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining, (ASONAM 2012), IEEE Computer Society, 2012, pp. 457–463.

[4] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed graphlab: a framework for machine learning and data mining in the cloud, Proc. VLDB Endow. 5 (8) (2012) 716–727.
[5] F.C. Bernstein, T.F. Koetzle, G.J. Williams, E.F. Meyer, M.D. Brice, J.R. Rodgers, O. Kennard, T. Shimanouchi, M. Tasumi, The protein data bank, Eur. J. Biochem. 80 (2) (1977) 319–324.
[6] The size of the world wide web (the Internet), http://www.worldwidewebsize.com/ (accessed 09.02.16).
[7] Facebook reports first quarter 2016 results and announces proposal for new class of stock, https://investor.fb.com/investor-news/press-release-details/2016/Facebook-Reports-First-Quarter-2016-Results-and-Announces-Proposal-for-New-Class-of-Stock/default.aspx (accessed 27.04.16).
[8] The top 20 valuable facebook statistics updated may 2016, https://zephoria.com/top-15-valuable-facebook-statistics/ (accessed 31.05.16).
[9] S. Fortunato, Community detection in graphs, Phys. Rep. 486 (3) (2009) 75–174. http://dx.doi.org/10.1016/j.physrep.2009.11.002.
[10] U. Kang, C.E. Tsourakakis, C. Faloutsos, PEGASUS: A peta-scale graph mining system implementation and observations, in: 2009 Ninth IEEE International Conference on Data Mining, pp. 229–238. http://dx.doi.org/10.1109/ICDM.2009.14.
[11] M.N. Kolountzakis, G.L. Miller, R. Peng, C.E. Tsourakakis, Efficient triangle counting in large graphs via degree-based vertex partitioning, Internet. Math. 8 (1) (2012) 161–185. http://dx.doi.org/10.1080/15427951.2012.625260.
[12] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank citation ranking: Bringing order to the web.
[13] B.V. Cherkassky, A.V. Goldberg, T. Radzik, Shortest paths algorithms: Theory and experimental evaluation, Math. Program. 73 (2) (1996) 129–174. http://dx.doi.org/10.1007/BF02592101.
[14] U. Kang, C.E. Tsourakakis, A.P. Appel, C. Faloutsos, J. Leskovec, HADI: Mining radii of large graphs, ACM Trans. Knowl. Discov. Data 5 (2) (2011) 8:1–8:24. http://dx.doi.org/10.1145/1921632.1921634.
[15] M.R. Garey, D.S. Johnson, L. Stockmeyer, Some simplified NP-complete problems, in: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, STOC '74, ACM, 1974, pp. 47–63. http://dx.doi.org/10.1145/800119.803884.
[16] K. Andreev, H. Rcke, Balanced graph partitioning, in: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'04, ACM, New York, NY, USA, 2004, pp. 120–124. http://dx.doi.org/10.1145/1007912.1007931.
[17] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. http://dx.doi.org/10.1145/1327452.1327492.
[18] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 135–146. http://dx.doi.org/10.1145/1807167.1807184.
[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. Hellerstein, GraphLab: A new framework for parallel machine learning, in: The 26th Conference on Uncertainty in Artificial Intelligence (UAI), Catalina Island, USA, 2010.
[20] A. Pothen, H. Simon, K. Liou, Partitioning sparse matrices with eigenvectors of graphs, SIAM. J. Matrix Anal. & Appl. 11 (3) (1990) 430–452. http://dx.doi.org/10.1137/0611030.
[21] A. Pothen, H.D. Simon, L. Wang, S.T. Barnard, Towards a fast implementation of spectral nested dissection, in: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, Supercomputing'92, IEEE Computer Society Press, 1992, pp. 42–51.
[22] S.T. Barnard, H.D. Simon, Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, Concurrency: Pract. Exper. 6 (2) (1994) 101–117. http://dx.doi.org/10.1002/cpe.4330060203.
[23] M.J. Berger, S.H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, IEEE Trans. Comput. C-36 (5) (1987) 570–580. http://dx.doi.org/10.1109/TC.1987.1676942.
[24] B. Nour-Omid, A. Raefsky, G. Lyzenga, Solving finite element equations on concurrent computers.
[25] An efficient heuristic procedure for partitioning graphs - kernighan - 2013 - bell system technical journal - wiley online library.
[26] C.M. Fiduccia, R.M. Mattheyses, A linear-time heuristic for improving network partitions, in: 19th Conference on Design Automation, 1982, pp. 175–181. http://dx.doi.org/10.1109/DAC.1982.1585498.
[27] G. Karypis, V. Kumar, METIS unstructured graph partitioning and sparse matrix ordering system, version 2.0.
[28] T. Verbelen, T. Stevens, F. De Turck, B. Dhoedt, Graph partitioning algorithms for optimizing software deployment in mobile cloud computing, Future Gener. Comput. Syst. 29 (2) (2012) 451–459. http://dx.doi.org/10.1016/j.future.2012.07.003.
[29] C. Gkantsidis, B. Radunovic, M. Vojnovic, FENNEL: Streaming Graph Partitioning for Massive Scale Graphs, Technical Report MSR-TR-2012-213, Microsoft Research, Redmond, WA, 2012, p. 98052.
[30] I. Stanton, Streaming balanced graph partitioning algorithms for random graphs, in: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2014, pp. 1287–1301.
[31] C. Battaglino, P. Pienta, R. Vuduc, GraSP: distributed streaming graph partitioning, in: HPGM: High Performance Graph Mining - 1st High Performance Graph Mining workshop, Barcelona Supercomputing Center, Sydney, 2015, http://dx.doi.org/10.5821/hpgm15.3.
[32] D. Margo, M. Seltzer, A scalable distributed graph partitioner, Proc. VLDB Endow. 8 (12) (2015) 1478–1489. http://dx.doi.org/10.14778/2824032.2824046.
[33] F. Bourse, M. Lelarge, M. Vojnovic, Balanced graph edge partition, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2014, pp. 1456–1465.
[34] F. Rahimian, A.H. Payberah, S. Girdzijauskas, M. Jelasity, S. Haridi, A distributed algorithm for large-scale graph partitioning, ACM Trans. Auton. Adapt. Syst. 10 (2) (2015) 1–24. http://dx.doi.org/10.1145/2714568.
[35] Y. Shao, B. Cui, L. Ma, Page: a partition aware engine for parallel graph computation, IEEE Trans. Knowl. Data Eng. 27 (2) (2015) 518–530.
[36] J. Huang, D.J. Abadi, Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs, Proc. VLDB Endow. 9 (7) (2016) 540–551. http://dx.doi.org/10.14778/2904483.2904486.
[37] C. Tsourakakis, C. Gkantsidis, B. Radunovic, M. Vojnovic, FENNEL: Streaming graph partitioning for massive scale graphs, in: Proceedings of the 7th ACM International Conference on Web Search and Data Mining, WSDM '14, ACM, 2014, pp. 333–342. http://dx.doi.org/10.1145/2556195.2556213.
[38] T. Kajdanowicz, P. Kazienko, W. Indyk, Parallel processing of large graphs, Future Gener. Comput. Syst. 32 (2013) 324–337. http://dx.doi.org/10.1016/j.future.2013.08.007.
[39] stanford snap, http://snap.stanford.edu/ (accessed 01.10.16).
[40] Konect datasets, http://konect.uni-koblenz.de/ (accessed 01.10.16).
[41] Us patents network dataset – KONECT (Oct. 2016). URL http://konect.uni-koblenz.de/networks/patentcite.
[42] Hudong internal links network dataset – KONECT (Oct. 2016). URL http://konect.uni-koblenz.de/networks/ zhishi-hudong-internallink.
[43] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, Z. Ghahramani, Kronecker graphs: An approach to modeling networks, J. Mach. Learn. Res. 11 (2010) 985–1042.

Zhan Shi received his B.S. degree and Master degree in Computer Science, and Ph.D. degree in Computer Engineering from Huazhong University of Science and Technology (HUST), China. He is working at the Huazhong University of Science and Technology (HUST) in China, and is an Associate Researcher in Wuhan National Laboratory for Optoelectronics. His research interests include graph processing, distributed storage system and cloud storage.

Junhao Li, born in 1991, M.S. candidate. His research interest is cloud storage and graph computing.

Pengfei Guo, born in 1989, he received M.S. degree in 2015. His research interest is graph computing.

Shuangshuang Li, born in 1992, M.S. candidate. Her research interest is graph computing.

Dan Feng received the B.E., M.E., and Ph.D. degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and the dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEETC, IEEETPDS, ACM-TOS, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She has served as the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012, 2015. She is a member of IEEE and a member of ACM.

Yi Su, received the B.S. degree in Computer Science from the Huazhong University of Science and Technology (HUST), China, in 2012. He is currently a Ph.D. candidate of the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. His research interests include big data processing systems, cloud storage systems.