2017 IEEE 37th International Conference on Distributed Computing Systems

Parallel Algorithm for Core Maintenance in Dynamic Graphs

Na Wang, Dongxiao Yu*, Hai Jin, Chen Qian, Xia Xie, Qiang-Sheng Hua
Services Computing Technology and System Lab
Big Data Tecgnology and System Lab
Clusters and Grid Computing Lab
School of Computer Science and Technology.
Huazhong University of Science and Technology, Wuhan, 430074, China
Email: {Ice_lemon,dxyu,hjin,M201572720,shelicy,qshua}@hust.edu.cn

***Abstract*—This paper initiates the studies of parallel algorithm for core maintenance in dynamic graphs. The core number is a fundamental index reflecting the cohesiveness of a graph, which is widely used in large-scale graph analytics. We investigate the parallelism in the core update process when multiple edges and vertices are inserted. Specifically, we discover a structure called *superior edge set*, the insertion of edges in which can be processed in parallel. Based on the structure of superior edge set, an efficient parallel algorithm is then devised. To the best of our knowledge, the proposed algorithm is the first parallel one for the fundamental core maintenance problem. Finally, extensive experiments are conducted on different types of real-world and synthetic datasets, and the results illustrate the efficiency, stability and scalability of the proposed algorithm. The algorithm shows a significant speedup in the processing time compared with previous results that sequentially handle edge and vertex insertions.**

***Keywords*—parallel algorithm; core maintenance; dynamic graph**

I. Introduction

Graph analytics has drawn much attention from research and industry communities, due to the wide applications of graph data in different domains. One of the major issues in graph analytics is identifying cohesive subgraphs. K-core is recognized as one of the most efficient and helpful index to depict the cohesiveness of a graph. Given a graph G, the k-core is the largest subgraph in G, such that the minimum degree of the subgraph is at least k. The core number of a vertex v is defined as the largest k such that there exists a k-core containing v. Besides the analysis of cohesive subgroup, k-core are widely used in a large number of applications to analyze the structure and function of a network.

In static graphs, the computation of the core number of each vertex is known as the k-core decomposition problem which has been well studied. The algorithm presented in [4] can compute the core number of each vertex in $O(m)$ time, where m is the number of edges in the graph. However, in many real-world applications, graphs are subject to continuous changes like insertions of vertices and edges. In such dynamic graphs, many applications require to maintain the core number for every vertex in-time, given the network changes over time. But it would be expensive to recompute the core numbers of vertices after every change of the graph, though the computation time is linear, as the size of the graph can be very large. Furthermore, the graph change may only affect the core numbers of a small part of vertices. Hence, the *core maintenance* problem [10] is recommended, which is to identify the vertices whose core numbers will be definitely changed after the graph changes.

Previous works focus on maintaining the core numbers of vertices in the scenario that a single edge is inserted into the graph. For multiple edge/vertex insertions, the inserted edges are processed sequentially. The sequential processing approach, on the one hand, incurs extra overheads when multiple edges/vertices are inserted, and on the one hand, it does not fully make use of the computation power provided by multicore and distributed systems. Therefore, it is necessary to investigate the parallelism in the edge/vertex processing procedure. But to the best of our knowledge, there are no parallel algorithms proposed for the core maintenance problem.

As we can see, the insertions of vertices can be regarded as multiple edge insertions. Hence, we consider only the scenario of edge insertions, the core maintenance problem under which is called the *incremental* core maintenance problem.

We present an efficient algorithm for incremental core maintenance. Specifically, we propose a structure called *superior edge set* the insertion of which can change the core number of every vertex by at most 1. Hence, the core numbers of vertices when inserting a superior edge set can be maintained using a parallel procedure: first identifying the vertices whose core numbers will change due to the insertion of every edge in parallel, and then updating the core number of these vertices by 1. A parallel algorithm can then be obtained by iteratively handling the insertions of split superior edge sets using the above parallel procedure.

In summary, our contributions are summarized as follows.

- We propose a structure called *superior edge set*, and show that if the edges of a superior edge set is inserted into a graph, the core number of each vertex can increase by at most 1. We also give sufficient conditions for identifying the vertices whose core numbers will increase.
- We then present a parallel algorithm for incremental core maintenance. Comparing with the sequential algorithm,

*The Corresponding Author is Dongxiao Yu (dxyu@hust.edu.cn).

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.288

our algorithm reduces the number of iterations for processing s inserted edges from s to the maximum number of edges inserted to each vertex. In large-scale graphs, the acceleration is significant, since each vertex can connect to only a few inserted edges. For example, as shown in the experiments, even if inserting 2×10^4 edges to the LiveJournal graph (refer to Table I in Section VI), the number of iterations is just 3 in our parallel algorithm, in contrast with 2×10^4 ones in the sequential processing algorithm.

- We also conduct extensive experiments over both real-world and synthetic graphs, to evaluate the efficiency, stability and scalability of our algorithm. The results show that comparing with the sequential processing algorithm, ours significantly speeds up core maintenance, especially in cases of large-scale graphs and large amounts of edge insertions.

The rest of this paper is organized as follows. In Section II, we briefly review closely related works. In Section III, the problem definitions are given. Theoretical results supporting the algorithm design are presented in Section IV. The parallel algorithm is proposed in Section V. In Section VI, the experiment results are illustrated and analyzed. The whole paper is concluded in Section VII.

II. Related Work

In static graphs, the core decomposition problem has been extensively studied [4], [6], [9], [11]. In contrast, researches on core maintenance in dynamic graphs are fewer. All previous works focus on the case of single edge update, and sequentially handle multiple edge updates. Efficient algorithms were proposed in [10], [12]. In [14], an algorithm was proposed to improve the I/O efficiency. Furthermore, [1] and [2] solved the core maintenance problem in the distributed environment.

III. Problem Definitions

We consider an undirected, unweighted simple graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Let $n = |V|$ and $m = |E|$. For a vertex $u \in V$, the set of its neighbors in G is denoted as $N(u)$ and the number of u's neighbors in G is called the degree of u, denoted as $d_G(u)$. So $d_G(u) = |N(u)|$. The maximum and minimum degree of vertices in G is denoted as $\Delta(G)$ and $\delta(G)$ respectively. We next give formal definitions for the *core number* of a vertex and other related concepts.

*Definition 1 (**k-Core**):* Given a graph $G = (V, E)$ and a non-negative integer k, the k-core is a connected subgraph H of G, where each vertex has at least k neighbors in H, i.e., $\delta(H) \geq k$.

*Definition 2 (**Core Number**):* Given a graph $G = (V, E)$, the core number of a vertex $u \in G$, denoted by $core_G(u)$, is the largest k, such that there exists a k-core containing u.

For simplicity, we use $core(u)$ to denote $core_G(u)$ when the context is clear.

*Definition 3 (**Max-k-Core**):* The max-k-core associated with a vertex u, denoted by H_u, is the k-core with $k = core(u)$.

In this work, we aim at maintaining the core numbers of vertices when a set of edges are inserted to the original graph.

IV. Theoretical Basis

In this section, we give some theoretical Lemmas that constitute the theoretical basis of our algorithm.

At first, we introduce some definitions. Given a graph $G = (V, E)$, an edge $e = < u, v >$ is called a *superior edge* for u if $core(v) \geq core(u)$. Notice that in the definition, we do not require $e \in E$, i.e., e may be an edge that is about to insert to graph G. Furthermore, we define the core number of an edge as the smaller one of its endpoints, i.e., $core(e) = \min\{core(u), core(v)\}$.

*Definition 4 (**k-Superior Edge Set**):* An edge set $E_k = \{e_1, e_2, ..., e_p\}$ is called an *k-superior edge set*, if for each edge $e_i =< u_i, v_i >$, $1 \leq i \leq p$, it satisfies:

(i) e_i is a superior edge with core number k.

(ii) if e_i and $e_j (1 \leq i, j \leq p, i \neq j)$ have a common endpoint u', $core_G(u') > k$.

In other words, in a k-superior edge set E_k, each edge is a superior edge of a vertex with core number k, and each vertex connects to at most one superior edge for it.

The union of several k-superior edge sets with distinct k values is called a *superior edge set*. It can be known that in a superior edge set, each vertex can still connect to at most one superior edge for it.

In the following, we will first show that when inserting a superior edge set the core number of every vertex can increase by at most 1 (Lemma 1 and Lemma 2), and then give a sufficient condition for identifying vertices whose core numbers change (Lemma 5, Lemma 6). Due to the page limit, the detailed proofs of these Lemmas are omitted. Please refer to [13] for the proofs.

A. Superior Edge Set Insertion

We first show a result on the core number increase of every vertex when inserting a k-superior edge set.

Lemma 1: Given a graph $G = (V, E)$, if a k-superior edge set $E_k = \{e_1, e_2, e_3, ..., e_p\}$ is inserted into G, where $k \geq 0$, for each node v, it holds that:
(i) if $core(v) = k$, $core(v)$ can increase by at most 1;
(ii) if $core(v) \neq k$, $core(v)$ will not change.

From the above Lemma 1, we have known that for a graph $G = (V, E)$, after a k-superior edge set $E_k = \{e_1, e_2, e_3, ..., e_p\}$ is inserted into G, only vertices with core numbers k may increase, and the increase is at most 1. This implies that it will be enough to only visit vertices whose core numbers are k and check if their core numbers will increase when a k-superior edge set is inserted. In fact, we can get even better results, which are given in the following Lemma 2.

Lemma 2: Given a graph G=(V, E) and a superior edge set $\mathcal{E}_q = E_{k_1} \cup E_{k_2} \cup, ..., \cup E_{k_q}$, where E_{k_i} for $1 \leq i \leq q$ is a k_i-superior edge set and $k_i < k_j$ if $i < j$, it holds that after inserting $\mathcal{E}_q$ into G, the core number of each vertex u can increase by at most 1.

Lemma 2 shows that when inserting a superior edge set into a graph, the core numbers of vertices can change by at most 1. This implies that the core updates of inserting edges in a superior edge set can be processed in parallel, as it only needs to find the union of affected vertices by the edge insertions. In the subsequent section, we give more accurate conditions for a vertex to change its core number when inserting a superior edge set.

B. Core Number Change

Here we consider the scenario defined as follows: given a graph $G = (V, E)$ and an edge set $E_s = \{e_1, e_2, ..., e_s\}$, w.l.o.g., assume that for each $e_i = < u_i, v_i >$, $core_G(v_i) \geq core_G(u_i) = k_i$, where $s > 0$, $1 \leq i \leq s$, and $k_i \geq 1$. Denote by $G' = (V, E')$ the obtained graph after inserting E_s into G.

*Definition 5 (**Superior Degree**):* For a vertex $u \in V$, v is a *superior neighbor* of u if v is a neighbor of u in G' and $core_G(v) \geq core_G(u)$. The number of u's superior neighbors is called the *superior degree* of u, denoted as $SD_{G'}(u)$.

It can be known that only superior neighbors of a vertex may affect the change of its core number.

*Definition 6 (**Constraint Superior Degree**):* The *constraint superior degree* $CSD_{G'}(u)$ of a vertex u is the number of u's neighbors w in G' that satisfies $core_G(w) > core_G(u)$ or $core_G(w) = core_G(u) \wedge SD_{G'}(w) > core_G(u)$.

When the context is clear, we use $SD(u)$ and $CSD(u)$ to represent the SD and CSD values of vertex u in the current new graph.

*Definition 7 (**K-Path-Tree**):* For the new graph $G' = (V, E')$, for a vertex u with a core number $core_G(u)$, the *K-Path-Tree* of u is a DFS tree rooted at u and each vertex w in the tree satisfies $core_G(w) = core_G(u)$. For simplicity we use $KPT^u_{G'}$ to represent K-Path-Tree of u in G'.

When a superior edge of u is inserted, as shown in Lemma 1, only vertices in $KPT^u_{G'}$ may change their core numbers. And a more accurate condition was given in [12] for identifying the set of vertices that may change core numbers, as shown below.

Lemma 3 ([12]): Given a graph $G = (V, E)$, if an edge $< u, v >$ is inserted and $core(u) \leq core(v)$, then only vertices w in the $KPT^u_{G'}$ of u and $CSD_{G'}(w) > core_G(u)$ may have their core numbers increased, and the increase is at most 1.

However, the above Lemma 3 is just suitable for the one edge insertion scenario. We next generalize it to the scenario of inserting multiple edges, as shown in Lemma 5 below. Before giving the result, we need to generalize the concept of K-Path-Tree to exPath-Tree.

*Definition 8 (ex**Path-Tree**):* For the new graph $G' = (V, E')$ and the edge set E_s, the union of K-Path-Tree for every u_i is called the *exPath-Tree* of E_s in G', denoted as $exPT_{G'}(E_s)$.

By Lemma 3, we can get that when inserting a k-superior edge set E_k, only vertices w in the $exPT_{G'}(E_k)$ satisfying $CSD_{G'}(w) > k$ may have their core numbers increase, and Lemma 1 ensures that these vertices can change their core numbers by at most 1. This result is summarized in the following Lemma.

Lemma 4: Given a graph $G = (V, E)$, if a k-superior edge set E_k is inserted, then in the obtained graph G', only vertices w in the $exPT_{G'}(E_k)$ satisfying $CSD_{G'}(w) > k$ may have their core numbers increased, and the increase is at most 1.

The above Lemma 4 implies that after an edge in a k-superior edge set E_k is inserted, the vertices whose core numbers change during the insertion will not change any more when inserting other edges in E_k. Based on the above result and Lemma 2, we can get the set of vertices whose core number change when inserting a superior edge set.

Lemma 5: Given a graph $G = (V, E)$, if a superior edge set $\mathcal{E}_q = E_{k_1} \cup \ldots \cup E_{k_q}$ is inserted, and G becomes G', then only vertices w in every *exPath-Tree*s of every E_{k_i} for $1 \leq i \leq q$ satisfying $CSD_{G'}(w) > k_i$ may have their core numbers increased, and the core number change is at most 1.

Additionally, by the definition of CSD, we have the following result.

Lemma 6: For a vertex u, if $CSD_{G'}(u) \leq core_G(u)$, v will not increase its core number.

Lemma 6 can help us remove the vertices that cannot increase core numbers.

Lemma 5 and Lemma 6 give accurate conditions to determine the set of vertices that will change the core numbers, after inserting a superior edge set. In the subsequent Section V, we will show how to utilize these theoretical results to design parallel algorithm for incremental core maintenance.

V. Incremental Core Maintenance

The parallel incremental core maintenance algorithm is given in Algorithm 1. We consider the core number update of vertices after inserting a set of edges E' to graph $G = (V, E)$. Let V' denote the set of vertices connecting to edges in E'. The set of core numbers of vertices in V' is denoted as $\mathcal{C}$.

The algorithm is executed in iterations. In each iteration, a parallel algorithm is conducted to find a superior edge set from the inserted edges. Then for each k-superior edge set E_k, a parallel algorithm is executed to identify the set of vertices whose core numbers change, and increase the core numbers of these vertices by 1.

In Algorithm 2, for each $k \in \mathcal{C}$, a child process is assigned to find the vertices whose core number changes are caused by the insertion of the computed k-superior edge set. Algorithm 2 first computes SD values for each vertex in $exPT$ of E_k, and then for each edge $e_i =< u_i, v_i >$, finds the set of vertices whose core numbers change due to the insertion of e_i. For e_i, a *positive* Depth-First-Search (DFS) is conducted on vertices in $KPT^r_{G_s}$ from the root vertex r, which is one of u_i or v_i that has a core number k,[1] to explore the set of vertices whose core numbers potentially change. In the algorithm, the cd value of each vertex v is used to evaluate the potential of a vertex to increase its core number. The initial value of $cd(v)$ is set as $CSD(v)$. For a vertex v, if $cd[v] \leq k$, its core number cannot increase. If a vertex v with $cd[v] \leq k$

[1] If both u_i and v_i have a core number equal to k, then r can be either u_i or v_i.

Algorithm 1: SuperiorEdgeInsert($G, E', V', core()$)

Input
The graph, $G = (V, E)$;
The inserted edge set, E';
The set of vertices V' connected to edges in E';
The core number $core(v)$ of each vertex in V;
while E' *is not empty* **do**
1 **for** *each vertex* u *in* V' **do**
2 **if** u *connects to a superior edge in* E' *and* $core(u) \notin \mathcal{C}$ **then** add core(u) to $\mathcal{C}$;
3 **for** *each core number* k *in* $\mathcal{C}$ *in parallel* **do**
4 **for** *each vertex* u *in* V' *with core number* k **do**
5 find a superior edge $< u, v >$ of u from E';
6 add $< u, v >$ to E_k;
7 insert $\cup_{k \in \mathcal{C}} E_k$ into G and denote the new graph as $G_s = (V, E_s)$;
8 delete $\cup_{k \in \mathcal{C}} E_k$ from E';
9 **for** *each core number* k *in* $\mathcal{C}$ *in parallel* **do** $V_k \leftarrow K$-SuperiorInsert($G_s, E_k, core()$);
10 **for** *each vertex* v *in* $\cup_{k \in \mathcal{C}} V_k$ **do** $core(v) \leftarrow core(v) + 1$;

is traversed in the positive DFS procedure, a *negative* DFS procedure initiated from v will be started, to remove v and update the cd values of other vertices with core number k. After all vertices in $KPT^r_{G_s}$ are traversed, the vertices that are visited but not removed increase the core numbers by 1.

Performance Analysis. We next analyze the correctness and efficiency of the proposed incremental algorithm. At first, some notations are defined, which will be used in measuring the time complexity of the algorithm.

For graph $G = (V, E)$, the inserted edge set E' and a subset S of E', let $G_S = (V, E \cup S)$ and $K(G_S)$ be the set of core numbers of vertices in G_S.

For G_S, let $L_S = \max_{u \in V}\{CSD(u) - core_{G_S}(u), 0\}$. As shown later, L_S is the max times a vertex u can be visited by negative DFS procedures in the algorithm execution.

For $k \in K(G_S)$, let $V_S(k)$ be the set of vertices with core number k, and $N(V_S(k))$ be the neighbors of vertices in $V_S(k)$. Let $n_S = \max\{|V_S(k)| : k \in K(G_S)\}$.

Denoted by $E[V_S(k)]$ the set of edges in G_S that are connected to vertices in $V_S(k) \cup N(V_S(k))$. Then we define m_S as follows, which represents the max number of edges travelled when computing SD in the case of inserting edges to G_S.

$$m_S = \max_{k \in K(G_S)} \{|E[V_S(k)]|\}.$$

Furthermore, we define the *maximum inserted degree* as the maximum number of edges inserted to each vertex in V, denoted as Δ_I. Then we have the following result which states

Algorithm 2: K-SuperiorInsert(G_s, E_k, core())

Input
The graph, $G_s = (V, E_s)$;
The k-superior edge set, E_k;
The core number $core(v)$ of each vertex in V;
Initially, $S \leftarrow$ empty stack;
for each vertex $v \in V$,
$visited[v] \leftarrow false, removed[v] \leftarrow false, cd[v] \leftarrow 0$;
1 compute $SD(v)$ for each vertex v in $exPT$ of E_k;
2 **for** *each* $e_i =< u_i, v_i > \in E_k$ **do**
3 **if** $core(u_i) \geq core(v_i)$ **then** $r \leftarrow v_i$; **else** $r \leftarrow u_i$;
4 **if** $visited[r] = false$ *and* $removed[r]$ = *false* **then**
5 **if** $CSD[r] = 0$ **then** compute $CSD[r]$;
if $cd[r] >= 0$ **then** $cd[r] \leftarrow CSD[r]$;
else $cd[r] \leftarrow cd[r] + CSD[r]$;
$S.push(r)$;
$visited[r] \leftarrow true$;
6 **while** *S is not empty* **do**
$v \leftarrow S.pop()$;
7 **if** $cd[v] > k$ **then**
8 **for** *each* $< v, w > \in E_s$ **do**
9 **if** $core(w) = k$ *and* $SD(w) > k$ *and* $visited[w]$ = *false* **then**
$S.push(w)$;
$visited[w] \leftarrow true$;
if $CSD[w] = 0$ **then** compute $CSD[w]$
$cd[w] \leftarrow cd[w] + CSD[w]$;
10 **else**
if $removed[v] = false$ **then** InsertRemove($G_m, core(), cd[]$, $removed[], k, v$);
11 **for** *each vertex* v *in* G_s **do**
if $removed[v] = false$ *and* $visited[v] = true$ **then** $V_k \leftarrow V_k \cup \{v\}$;
12 **return** V_k;

the correctness and efficiency of our algorithm. The proof of Theorem 7 can be found in [13].

Theorem 7: Algorithm 1 can update the core numbers of vertices after inserting an edge set E' in $O(\Delta_I * \max_{S \subseteq E'}\{m_S + L_S * n_S\})$ time.

VI. Experiment Studies

In this section, we conduct empirical studies to evaluate the performances of our proposed algorithm. The experiments use three synthetic datasets and seven real-world graphs, as shown in Table I.

There are two main variations in our experiments, the original graph and the inserted edge set. We first evaluate

Algorithm 3: InsertRemove($G_s, core(), cd[], removed[], k, r$)

Input
The current new graph, $G_s = (V, E_s)$;
The $core(v), cd[v], removed[v]$ of each vertex;
The core number k and root r;

```
1  S ← empty stack;
2  S.push(r);
3  removed[r] ← true;
4  while S is not empty do
       v ← S.pop();
       for each < v, w >∈ E_s do
           if core(w) = k then
5              cd[w] ← cd[w] − 1;
               if cd[w] = k and removed[w] = false then
6                  S.push(w);
7                  removed[w] ← true;
```

the efficiency of our algorithm on real-world graphs, by changing the size and core number distribution of inserted edges. Then we evaluate the scalability of our algorithm using synthetic graphs, by keeping the inserted edge set stable and changing the sizes of synthetic graphs. At last, we compare our algorithm with the sequential edge processing approach based on the state-of-the-art core maintenance algorithm for single edge insertion, TRAVERSAL given in [12], to evaluate the acceleration ratio of our parallel algorithm. The comparison experiments are conducted on four typical real-world datasets.

All experiments are conducted on a Linux machine with Intel Xeon CPU E5-2670@2.60GHz and 64 GB main memory, implemented in C++ and compiled by g++ compiler.

Datasets. We use seven real-world graphs and random graphs generated by three models. The seven real-world graphs can be downloaded from SNAP [8], including social network graphs (LiveJournal, Youtube, soc-Slashdot), collaboration network graphs (DBLP, ca-astroph), communication network graphs (WikiTalk) and Web graphs (web-BerkStan). The synthetic graphs are generated by the SNAP system using the following three models: the Erdös-Rényi (ER) graph model [7], which generates a random graph; the Barabasi-Albert (BA) preferential attachment model [3], in which each node creates k preferentially attached edges; and the R-MAT (RM) graph model [5], which can generate large-scale realistic graphs similar to social networks. For all generated graphs, the average degree is fixed to 8, such that when the number of vertices in the generated graphs is the same, the number of edges is the same as well.

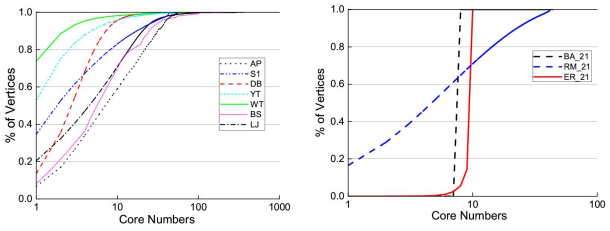
We use the *average processing time per edge* as the efficiency measurement of the algorithm, such that the efficiency of the algorithm can be compared in different cases.

A. Performance Evaluation

We evaluate the impacts of three factors on the algorithm performance: the size of inserted edges, the core number distribution of edges inserted, and the original graph size. The first factor affects the iterations needed to process the inserted edges, and the last two factors affect the processing time in each iteration. The first two evaluations are conducted on real-world graphs, and the third one is on synthetic graphs.

TABLE I
REAL-WORLD GRAPH DATASETS

Datasets	n=$\|V\|$	m=$\|E\|$	max degree	max core
AP(ca-Astroph)	18.7K	198.1K	504	56
S1(soc-Slashdot)	82.1K	500.5K	2548	54
DB(DBLP)	0.31M	1.01M	343	113
YT(YouTube)	1.13M	1.59M	28754	35
WT(wiki-Talk)	2.4M	9.3M	100029	131
BS(web-BerkStan)	0.68M	13.3M	84230	201
LJ(LiveJournal)	4.0M	34.7M	20334	360

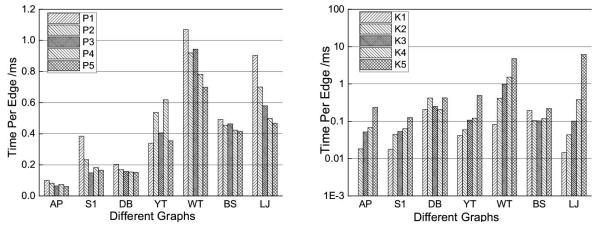


(a) Real-world Graphs (b) Generated Graphs

Fig. 1. Core Distribution

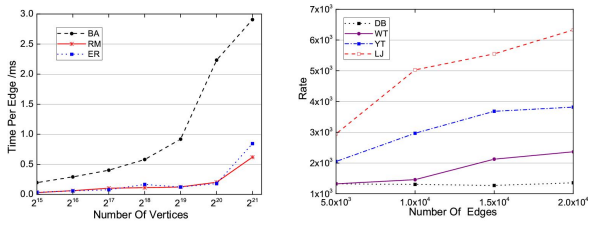
We first evaluate the impact of the number of inserted edges on the performances of our algorithm. The results for the incremental maintenance algorithm are illustrated in Fig. 2(a). In the experiments, we randomly insert P_i% edges with respect to the original graph, where $P_i = 3 * i$ for $i = 1, 2, 3, 4, 5$. It can be seen that the processing time per edge is less than $1.2ms$ in all cases, and except for WT and LJ, the processing time is much smaller than $1.2ms$. The figure also shows that the processing time decreases as the number of inserted edges increases, which demonstrates that our algorithm is suitable for handling large amount of edge insertions. In this case, more edges can be selected into the superior edge set in each iteration, and hence our algorithm achieves better parallelism.

We then evaluate the impact of the core number distribution of inserted edges on the algorithm performance. The results are illustrated in Fig. 2(b). In particular, by the core distributions showed in Fig. 1(a), we choose five typical core numbers $\{K1, K2, K3, K4, K5\}$ in an increasing order for each of the seven graphs. For each core number, 20% edges of that core number are selected randomly as the inserted edge set. From Fig. 2(b), it can be seen that larger core number induces a larger average processing time. This is because when inserting edges to vertices with larger core numbers, the degree of these vertices generated by these inserted edges is larger. In our algorithm, only one superior edge can be handled for each vertex in each iteration. Hence, it takes more iterations to process the inserted edges. But on the other hand, it can be



(a) Change Edge Number (b) Change Core Number

Fig. 2. Impact of Inserted Edge Number and Core Number



(a) Change Original Graph Size (b) Comparison

Fig. 3. Impact of Original Graph Size and Comparison with the TRAVERSAL Algorithm

also seen that the processing time per edge does not vary significantly.

We finally evaluate the scalability of our algorithm in synthetic graphs, by letting the number of vertices scale from 2^{15} to 2^{21} and keeping the average degree fixed as 8. The results are shown in Fig. 3(a). For each graph, we randomly select 10000 edges as the inserted edge set. Fig. 3(a) shows that though the graph size increases exponentially, the average processing time increases linearly. It demonstrates that our algorithm can work well in graphs with extremely large size.

B. Performance comparison

In this section, we evaluate the acceleration ratio of our parallel algorithm, comparing with the algorithm sequentially handling edge insertions. We compare with the state-of-the-art sequential algorithm, **TRAVERSAL** given in [12]. The comparison is conducted on four typical real-world graphs, DB, WT, YT and LJ as given in Table I. For each graph, we randomly select 5K-20K edges as the update set. The evaluation results are illustrated in Fig. 3(b). In the figure, the x-axis and y-axis represent the number of inserted edges and the acceleration ratio, respectively.

Fig. 3(b) shows that in almost all cases, our algorithm achieves an acceleration ratio as large as 10^3 times in the four graphs. The acceleration ratio increases as the number of edges inserted increases, which illustrates that our algorithm has better parallelism in scenarios of large amounts of graph changes. Furthermore, it is also shown that our algorithm achieves larger acceleration ratios as the graph size increases.

All evaluation results show that our algorithm exhibits good parallelism in core maintenance of dynamic graphs, comparing with the sequential algorithm. The experiments illustrate that our algorithm is suitable for handling large amounts of edge insertions in large-scale graphs, which is desirable in realistic implementations.

VII. Conclusion

In this paper, we present the first known parallel algorithm for core maintenance when multiple edges are inserted. Our algorithm has significant accelerations comparing with sequential processing algorithm that handles inserted edges sequentially, and reduces the number of iterations for handling s inserted edges from s to the maximum number of edges inserted to a vertex. Experiments on real-world and synthetic graphs illustrate that our algorithm implements well in reality, especially in scenarios of large-scale graphs and large amounts of edge insertions.

VIII. Acknowledgements

This work is supported by the National Natural Science Foundation of China Grants 61602195, 61572216, 61433019 and U1435217, Grant from the International Science and Technology Cooperation Program of China (No. 2015DFE12860), National Key Research and Development Program of China under grant No.2016QY02D0202, Fundamental Research Funds for the Central Universities, HUST: 0118210142 and 0180210115.

References

[1] H. Aksu, M. Canim, Y.-C. Chang, I, Korpeoglu, and O. Ulusoy. Distributed K-core View Materialization and Maintenance for Large Dynamic Graphs. *IEEE Transactions on Knowledge & Data Engineering*, 26(10):2439-2452, 2014.

[2] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis. Distributed K-core Decomposition and Maintenance in Large Dynamic Graphs. In *DEBS*, 2016.

[3] A.L. Barabasi, R. Albert. Emergence of Scaling in Random Networks. In *Science*, 286(5439):509-512, 1999.

[4] V. Batagelj and M. Zaversnik. An O(m) Algorithm for Cores Decomposition of Networks. In *CoRR, vol. cs.DS/0310049*, 2003.

[5] D. Chakrabarti, Y. Zhan, C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *ICDM*, 2004.

[6] J. Cheng, Y. Ke, S. Chu, M. T. özsu, Efficient Core Decomposition in Massive Networks. In *ICDE*, 2011.

[7] P. Erdös, A. Renyi. On the Evolution of Random Graphs. Publication of the Mathematical Institute of the Hungarian Academy Ofences, 38(1):17-61, 1960.

[8] L. Jure, K. Andrej. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, 2014.

[9] W. Khaouid, M. Barsky, V. Srinivasan, et al. K-core Decomposition of Large Networks on a Single PC. *In Proceedings of the VLDB Endowment*, 9(1):13-23, 2016.

[10] R.H. Li, J.X. Yu, R. Mao. Efficient core maintenance in large Dynamic Graphs. *IEEE Transactions on Knowledge & Data Engineering*, 26(10):2453-2465, 2014.

[11] A. Montresor, F.D. Pellegrini, D. Miorandi. Distributed K-core Decomposition. *IEEE Transactions on Parallel & Distributed Systems*, 24(2):288-300, 2011.

[12] A.E. Sariyuce, B. Gedik, G. Jacques-Silva, et al. Incremental K-core Decomposition: Algorithms and Evaluation. *The VLDB Journal*, 25(3):425-447, 2016.

[13] N. Wang, D. Yu, H. Jin, C. Qian, and X. Xie, Q-S. Hua. Parallel Algorithms for Core Maintenance in Dynamic Graphs. Technical Report. https://arxiv.org/pdf/1612.09368.pdf.

[14] D. Wen, L. Qin, Y. Zhang, X. Lin, and J.X. Yu. I/O Efficient Core Graph Decomposition at Web Scale. In *ICDE*, 2016.