Mathematical Biosciences and Engineering

http://www.aimspress.com/journal/MBE

MBE, 19(3): 2700–2719.
DOI: 10.3934/mbe.2022123
Received: 23 September 2021
Revised: 28 November 2021
Accepted: 27 December 2021
Published: 10 January 2022

Research article

The parallel computing of node centrality based on GPU

Siyuan Yin, Yanmei Hu* and Yuchun Ren

College of Computer and Cyber Security, Chengdu University of Technology, Chengdu, China

* **Correspondence:** Email: huyanmei@cdut.edu.cn.

Abstract: Many systems in real world can be represented as network, and network analysis can help us understand these systems. Node centrality is an important problem and has attracted a lot of attention in the field of network analysis. As the rapid development of information technology, the scale of network data is rapidly increasing. However, node centrality computation in large-scale networks is time consuming. Parallel computing is an alternative to speed up the computation of node centrality. GPU, which has been a core component of modern computer, can make a large number of core tasks work in parallel and has the ability of big data processing, and has been widely used to accelerate computing. Therefore, according to the parallel characteristic of GPU, we design the parallel algorithms to compute three widely used node centralities, i.e., closeness centrality, betweenness centrality and PageRank centrality. Firstly, we classify the three node centralities into two groups according to their definitions; secondly, we design the parallel algorithms by mapping the centrality computation of different nodes into different blocks or threads in GPU; thirdly, we analyze the correlations between different centralities in several networks, benefited from the designed parallel algorithms. Experimental results show that the parallel algorithms designed in this paper can speed up the computation of node centrality in large-scale networks, and the closeness centrality and the betweenness centrality are weakly correlated, although both of them are based on the shortest path.

Keywords: node centrality; parallel computing; CUDA

1. Introduction

Many systems in real world can be represented as network, e.g., the proteins and their interactions can be represented as protein-protein interaction network [1], the sensors and their communications can be represented as wireless sensor network [2], and the social relationships can be represented as social network. The analysis of network is necessary for us to understand the world. Node centrality is an important research problem in social network analysis, which aims to assess the importance of nodes and thus identify the important nodes in the network. A node usually has different importance in different applications. Thus, the centrality of nodes can be assessed from different perspectives. Anyway, it is general that an important node determines the information propagation in the network or plays a leader role in its group. Based on this consideration, various centrality metrics such as degree centrality, closeness centrality, betweenness centrality and PageRank centrality [3] were proposed, and have been widely used in the literature. However, with the rapid increase of network data, it is highly time-consuming for many centrality metrics.

On the other hand, parallel computing, as an efficient approach to tasks with high time complexity, has been widely concerned and applied in many fields, e.g., deep learning [4] and the parallel implementation of the evolutionary algorithms [5−7], especially with the development of GPU (Graphic Processing Unit) since it has superiority in a lot of parallel tasks. Therefore, it is a good alternative to evaluate and analyze the node centrality in large-scale networks by parallel computing using GPU. There have been some works about the parallel computing of node centrality [8−12], but they were not demonstrated on large-scale networks and the relationships of different centralities are not analyzed. Besides, there is no work presenting the parallel process in details, which makes the parallel algorithms confusing. Based on this, we design two parallel algorithms on GPU for three classical centralities, which are betweenness centrality, closeness centrality and PageRank centrality. Specifically, based on the definitions of centrality metrics and the structure of GPU we first design the parallel algorithms and present them in details. Then, we test the designed parallel algorithms on several networks including large-scale networks. Finally, based on the centrality values obtained, we analyze the correlations between different centralities in different networks. The experimental results show that the parallel algorithms designed for node centrality in this paper can speed up the calculation of node centrality in large-scale networks. Moreover, we find that the betweenness centrality strongly correlates to the PageRank centrality while weakly correlates to the closeness centrality, although both of the betweenness centrality and closeness centrality are based on the shortest path.

The organization of the remained part is as follows. Section 2 presents the related work and Section 3 presents the centrality metrics we focus on in this paper. Section 4 presents the proposed parallel algorithms for node centrality. Experimental results and analysis are presented in Section 5. Finally, Section 6 concludes our work.

2. Related works

We first briefly describe the classical works on node centrality, and then introduce the most related works about parallel computing.

2.1. Node centrality

In the current research of node centrality, most researches focus on global centrality and local centrality. The local centrality includes degree centrality, local clustering coefficient [13] and conductance [14] and density of the ego network [15]. Among these four centrality, although degree centrality has been widely studied in the literature [16], it is still limited for the identification of important nodes and is usually used to supplement other measures. Local clustering coefficient is the ratio of the actual triangle count at a node to the number of possible triangles at that node based on how many neighbors it has. In addition, the local centrality of a node can be measured from its ego network since the ego network represents its local structure. Density of an ego network is the percentage of ties that are actually present in the ego network to the possible ties, and conductance is the ratio of ties that go out the ego network to the total ties related to the nodes in the ego network. However, local centrality doesn't consider the whole network structure, so this paper mainly researches the global centrality.

Global centrality takes into account the whole network structure in the calculation of centrality and mainly includes eigenvector centrality [17], betweenness centrality [18], closeness centrality [19] and PageRank centrality [20]. The eigenvector centrality means that the importance of a node depends not only on the number of its neighbors, but also on the importance of its neighbors. And both closeness centrality and betweenness centrality are measured by the shortest path between nodes. The difference is that closeness centrality focuses on the sum of the distances from the node to other nodes, while the betweenness centrality pays more attention to the proportion of the shortest paths through a certain node. The later one can be used to find nodes that have a bridge effect in the network, which plays an important role in network research. But betweenness centrality is time-consuming if it is calculated directly from its definition. Therefore, scholars have conducted a lot of researches to find the algorithms that can reduce the time complexity of this centrality. Fortunately, there is an algorithm for fast computation of the betweenness centrality was found, which is based on breadth-first search [21]. Based on this algorithm, a lot of works about the calculation of the betweeneess centrality and its application in other fields have been witnessed [22−26]. PageRank centrality originates in the field of information retrieval and is used to rank the related web pages in the search engine [27]. As the WWW can be seen as a huge network, the technique of PageRank is naturally extended to measure node importance in a network, which results in PageRank centrality. In addition, there are also some other metrics that can evaluate the centrality of nodes. For example, a metric based on the idea that the closer the node is to the inner layer, the more important it is [28]; the neighbors of a node and its degree were combined to measure the centrality in [29]. Inspired by gravity, some researchers argue that nodes have "attraction" and consider this "attraction" as a metric to evaluate the centrality of nodes. These researches mentioned above have introduced new insights to study the problem of node centricity, but the most widely used ones are still the closeness centrality, the betweenness centrality and the PageRank centrality. However, these centralities as well as other global centralities have not been analyzed in large-scale networks which are common in real applications, because in this case it costs a lot of time to calculate them by serial methods.

2.2. Parallel implementation for node centrality

Early parallel algorithms were stuck in parallel computing using a single processor, and various computational models for single processor parallel computing were summarized in [8]. To speed up the computation of betweenness centrality, parallel algorithms with multicore processors were proposed and implemented on the Cary MTA-2 shared-memory multiprocessor by reducing the atomic operations during parallelism [9].

With the development of GUP, betweenness centrality was computed in parallel on GPU for the first time in [10], and then parallel computing based on GPU has been attracting more and more attention, especially for large-scale data. In [11], several parallel implementations of betweenness based on GPU were compared and then a parallel algorithm based on multi-GPUs was proposed; the parallel implementation jointing multi-GPUs achieved a speedup ratio of more than 10 times in large scale networks.

In addition, edge parallel computing was also studied based on GPU. In [12], the authors implemented the edge parallel computing on GPU by considering better load balancing and less overhead. However, some of these parallel algorithms are limited by the synchronization access of data from multi-GPUs and high cost since GPU devices are usually very expensive; and they are not demonstrated on large-scale networks which is very necessary. Therefore, in this paper, we implement two parallel computing algorithms to solve the problem of node centrality in parallel and test them on several large-scale networks. Moreover, benefited from this, we analyze the correlations of different centralities in different networks, which can provide guidable suggestions in the practical application of these centralities.

3. Centrality metrics

In this section, we briefly describe the three centrality metrics we focus on in this paper, namely, closeness centrality, betweenness centrality and PageRank centrality.

3.1. Betweenness centrality

Betweenness centrality of a given node v_i is defined as follows:

$$C_{bc}(v_i) = \sum_{s \neq i \neq t} \frac{\sigma_{st}^{i}}{\sigma_{st}}, \quad (1)$$

where σ_{st} is the number of the shortest paths between nodes v_s and v_t, and σ_{st}^{i} is the number of the shortest paths that pass node v_i. It can be seen that the betweenness centrality is based on the shortest path between nodes. For each node it needs to calculate the shortest path between each pair of the other nodes in the network, so its time complexity reaches $O(n^3)$, which is quite high in real applications. Therefore, an algorithm for fast calculation of betweenness centrality was proposed in [21] to reduce the time complexity.

3.2. Closeness centrality

The closeness centrality of a given node v_i is defined based on the shortest paths of this node to other nodes, which is shown in Eq (2). It represents the ability of nodes to reach all the other nodes in the network, and achieves a balance between locality and globality. However, its time complexity, which is approximate to $O(n^2)$, is relatively high, especially for large-scale networks.

$$C_c(v_i) = \frac{n-1}{\sum_{j \neq i} dist_{ij}}, \quad (2)$$

where $dist_{ij}$ is the shortest path between nodes v_i and v_j. The shorter the shortest paths from node v_i to other nodes are, the higher the centrality of node v_i is.

3.3. PageRank centrality

PageRank centrality is based on the webpage ranking from information retrieve. Initially, each node is given an equal score, which is usually $1/n$; then each node passes its score to neighbors. This process is iterated a specified number of times or until the rank of nodes stops changing. The definition of PageRank centrality for a given node v_i is as follows:

$$C_{pr}(v_i) = \frac{1-\alpha}{n} + \alpha \sum_{v_j \in N(v_i)} \frac{C_{pr}(v_j)}{d_j}, \tag{3}$$

where α is the scaling factor which is used to avoid that the nodes with no neighbors absorb all the scores; $N(v_i)$ is the neighbors of node v_i; d_j is the degree of node v_j. In practical applications, the number of iterations is usually not large to obtain the node ranking. But for large-scale networks, its calculation is still time-consuming.

4. The parallel algorithms for node centrality

In this section, we firstly introduce the CUDA C programming model that implements parallel programming based on GPU, and then present the parallel algorithms for the three node centralities described in the previous section.

4.1. The CUDA C programming model

The CUDA programming model provides an abstract computer architecture that serves as a bridge between an application and the available hardware. It connects the main memory to the GPU memory, and allows data to be exchanged between the two memories. A typical CUDA program has the following steps: 1) copying data from main memory to GPU memory; 2) calling core function to operate on the data stored in GPU memory; and 3) transferring data from GPU memory to main memory. The core function is application-driven and is programmed by the designer of parallel algorithm according to the application.

The CUDA programming model is composed of a host and a device (here is the GPU), each having its own memory. There are mainly global memory and shared memory in GPU's memory structure. The global memory is used to store global variables and clock count variables, and the shared memory is used to store the variables that all the threads are allowed to access. Figure 1 shows a memory structure of GPU. The function cudaMemcpy executes the data transmission between the host and the GPU; and the cudaMalloc function is used to allocate memory to store the data copied from the host.

In order to facilitate the parallel execution of the core functions in all threads, CUDA requires a standardized organization of the threads so that the developer can quickly obtain the threads needed. See Figure 1 for an illustration of the thread hierarchy, which is composed of thread blocks and threads. All threads share the global memory, and the threads in each block share a piece of shared memory.

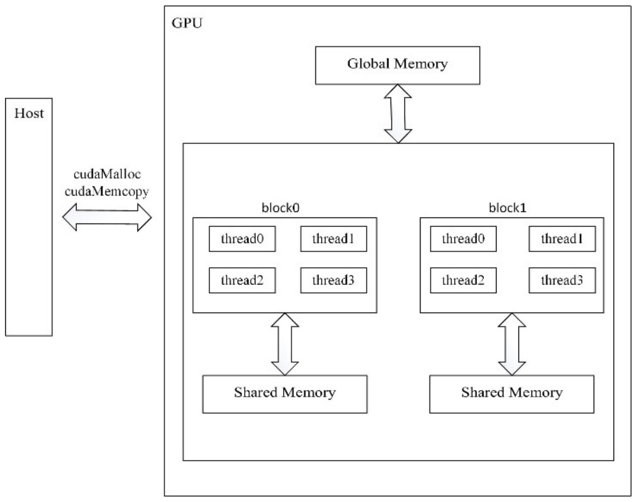


Figure 1. The memory structure and thread organization of GPU.

4.2. Parallel algorithms for node centrality

We classify the three node centralities described in the Section 3 into two groups, according to their definitions. The first group contains the betweenness centrality and the closeness centrality, and the second group contains the PageRank centrality. Next, we will describe how to calculate these three centralities in an undirected and unweighted network in parallel using GPU.

4.2.1. The parallel calculation of betweenness centrality and closeness centrality

From their definitions, it can be inferred that both of the betweenness centrality and the closeness centrality can be obtained by breadth-first search on the network, since the shortest path between a pair of nodes in an undirected and unweighted network is equal to the path obtained by the breadth-first search starting from one node to the other node. But there is a main difference: a node's closeness centrality only needs to perform the breadth-first search starting from this node while its betweenness centrality needs to perform the breadth-first search starting from all the other nodes. Anyway, both of these two centralities of each node can be obtained after performing the breadth-first search starting from each node. Thus, the core idea of the parallel computing of the two centralities is to perform the breadth-first search (starting from each node) in parallel. To do this, we allocate all the nodes into different blocks according to a mapping function set up in advance, and then let each block perform the breadth-first search from the allocated nodes in parallel. Based on this idea, the betweenness centrality and closeness centrality the can be obtained in parallel. Next, we first describe the parallel computing of the betweenness centrality using CUDA and then the parallel computing of the closeness centrality, since the later one can be obtained in the first step of the former one.

The parallel computing of the betweenness centrality. It is easy to understand that the betweenness centrality of node v_i can be decomposed into to many parts, and each part is the ratio of the shortest paths that pass node v_i starting from a node s to all of the other nodes (denoted as $\delta_s(v_i)$), i.e., $C_{bc}(v_i) = \sum_{s \neq v_i} \delta_s(v_i)$. As shown in [21], the betweenness centrality can be calculated by the approach of reverse deduction, since it has been proved that $\delta_s(v_i)$ can be calculated as:

$$\delta_s(v_i) = \sum_{w:v_i \in pred(s,w)} \frac{\sigma_{sv_i}}{\sigma_{sw}} (1 + \delta_s(w)), \tag{4}$$

where $pred(s,w)$ is the set of precursor nodes of w. It means that the betweenness centrality of node v_i can be calculated by finding the precursor nodes in the shortest paths that pass node v_i, starting from other nodes. It is therefore that the betweenness centrality can be calculated by the following steps: 1) performing the breadth-first search starting from each node s ($s \neq v_i$); 2) deducing reversely from the farthest node from s to obtain $\delta_s(v_i)$ according to Eq (4); 3) accumulating $\delta_s(v_i)$ for each starting node s to obtain the betweenness centrality of each node v_i. To perform the above steps in parallel, we can allocate all the nodes into different blocks and let each block to perform the breadth-first search and the reverse deduction from the nodes allocated to it, i.e., obtain $\delta_*(v_i)$ for each node v_i where $*$ denotes any node that is allocated to it. The betweenness centrality of each node v_i is then obtained by accumulating $\delta_*(v_i)$ from each block. It should be noted that we do this at block granularity, rather than thread granularity, because that the later one would consume a large amount of memory. Let's look back to Eq (4), there is a $\delta_s(v_i)$ for each node v_i starting from each node s. If here we allocate each node to a thread, then each thread would keep memory as large as *n* floating-point numbers (where *n* is the number of nodes in the network), resulting in a huge amount of memory. But in each block the calculation of $\delta_*(v_i)$ can be done in parallel, since the breadth-first search is performed layer by layer and the search in the same layer can be done in parallel. Assuming node s is allocated to block *j* with *blockdim* threads, the $\delta_s(v_i)$ for each node v_i is calculated in parallel by the following 4 steps. Firstly, some variables are required to store important information: a variable *level* represents the current level of the breadth-first search and is initialized to 0; three arrays, denoted as *dist*, *sigma*, *delt*a, with size *n* store the length of the shortest paths, the number of shortest paths and the $\delta_s(*)$, respectively, and the *dist* is initialized to -1 for each node except that $dist[s] = 0$, the *sigma* is initialized to 0 for each node except that $sigma[s] = 1$ and *delta* is initialized to 0 for each node. These variables are all stored in the shared memory of block *j* and can be accessed by all the threads in this block. Secondly, the *n* nodes are allocated to the *blockdim* threads of this block according to $nodeid = threadid + blockdim * i$ where $i = \{0,1,\cdots\}$ and $nodeid$ and $threadid$ represent the node id and thread id, respectively. For example, nodes with *id* of 0 and $blockdim$ are allocated to thread 0, nodes with *id* of 1 and $blockdim + 1$ are allocated to thread 1. Thirdly, the breadth-first search starting from s is performed in parallel: for the neighbor nodes of the nodes at current level, if they are not visited by the breadth-first search, i.e., their *dist* values are -1, then their corresponding threads change their *dist* values to $level + 1$ in parallel; moreover, their *sigma* values are added by the *sigma* values of their neighbor nodes at current level; then the current level is changed to the next level, i.e., $level = level + 1$. By repeating this process until the last level, the breadth-first search is done in parallel. Fourthly, the reverse deduction as shown in Eq (4) starting from the nodes at the last level is performed in parallel to obtain the *delta* values of each node (this step is similar to the parallel breadth-first search, except that this step starts from the last level to s). After all the blocks finish the above four steps, the betweenness centrality of each node can be obtained by accumulating $\delta_*(*)$ from each block.

For illustration, we list an example for the parallel computing of the breadth-first search and the reverse deduction since they are the most two important steps, see Figure 2. There are 8 nodes in the network (see Figure 2(a)), and two blocks with four threads for each block (see Figure 2(b)). Nodes with *id* of 0, 2, 4, 6 and 8 are allocated to block 0 and the remained nodes are allocated to block 1. Figure 2(c),(d) respectively show the steps of breadth-first search from node 0 and the corresponding reverse deduction in block 0 (the two steps with respect to other nodes are the same). To calculate the $\delta_0(v_i)$ in parallel by block 0, nodes are allocated to the four threads in block 0, e.g., nodes 0 and 4 are allocated to thread 0 and nodes 1 and 5 to thread 1, and each thread only handles the nodes that allocated to it. At first, node 0 is set to be at level 0 and the breadth-first search begins here: thread 0 does nothing about node 0 since all of its neighbor nodes have values of -1 on *dist*; threads 1, 2 and 3 visit nodes 1, 2 and 3 respectively, resulting in these nodes being set to be at level 1 and their *dist* values and *sigma* values are updated; subsequently, all the threads move to level 2 and visit nodes 4, 5, 6 and 7 and update their *dist* values and *sigma* values. Next, the reverse deduction starts from level 2, and nodes 4, 5, 6 and 7 are respectively handled by threads 0, 1, 2 and 3 to update the *delta* values of nodes 1, 2 and 3; then all the threads move back to level 1 and the process of reverse deduction is end since the starting node 0 is at level 0. It is noted that all the threads run in parallel, but a thread can move to the current level only if all the nodes at the previous (next) level are visited (handled). It means that synchronization is required among the four threads in block 0. Fortunately, CUDA provides synchronization mechanism, which is guaranteed by the function "__syncthreads ()".

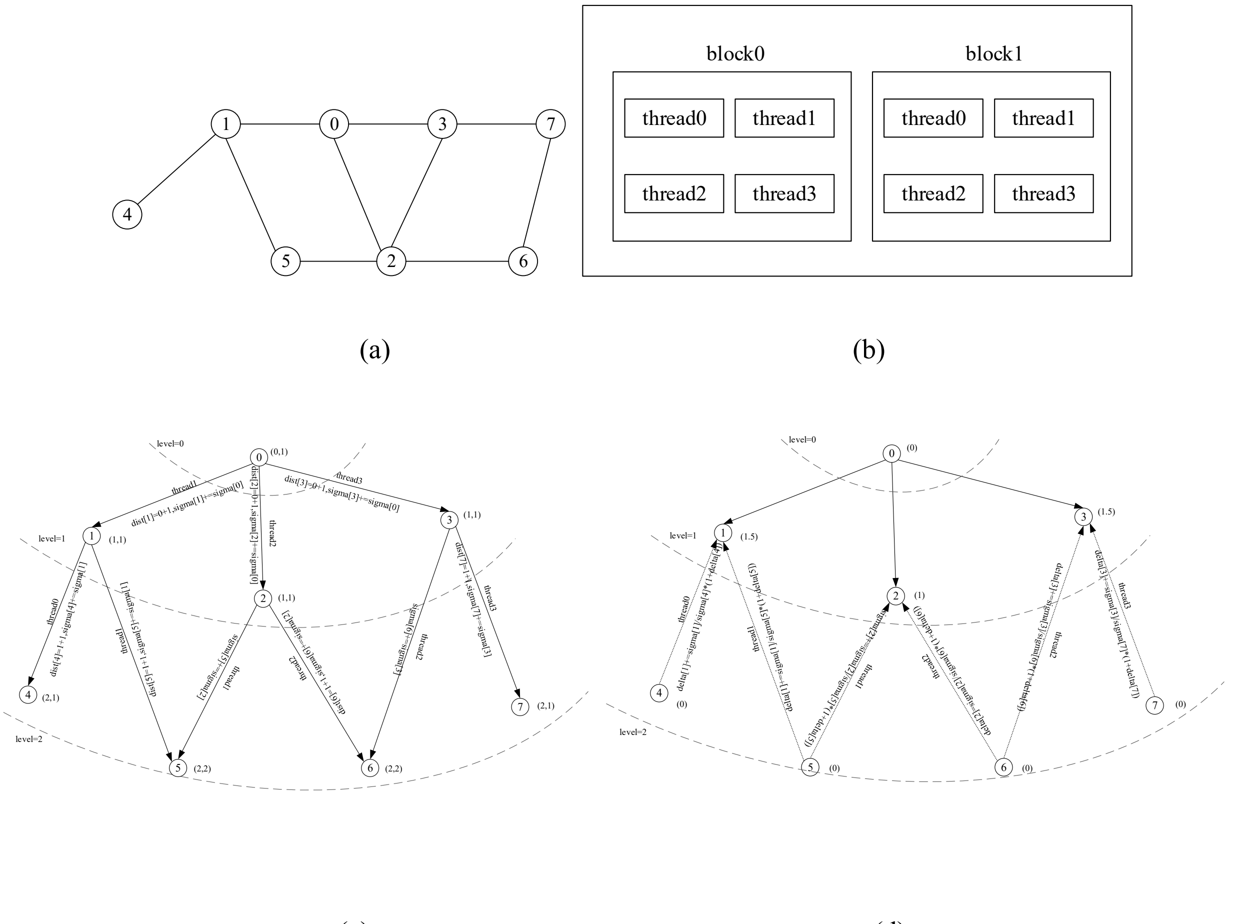


Figure 2. An example for the parallel computing of betweenness centrality. There are 8 nodes in the network (a), and two blocks with four threads for each block (b); the breadth-first search starting from node 0 (c) and the corresponding reverse deduction (d) are paralleled performed in block 0. The two numerical values in parentheses near the node in (c) are the *dist* and *sigma* values and the one in (d) is the *delta* value.

The parallel computing of the closeness centrality. It is easy to understand that a node's closeness centrality can be obtained by adding the *dist* values of other nodes after the breadth-first search starting from it. Thus, the parallel computing of the closeness centrality follows 3 steps: firstly, allocate all the nodes into different blocks; secondly, each block performs the breadth-first search starting from each of the allocated nodes in parallel; thirdly, for each starting node the *dist* values are accumulated to obtain its closeness centrality according to Eq (2). The first two steps are totally the same as in the parallel computing of the betweennees centrality, so the details are referred to that part.

Algorithm 1 *Parallel Algorithm I* (only the computation within one block is shown)

```
Input: $G = (V, E)$: the network with nodes numbered from 0 to $n-1$; $s$: the starting node.
Output: cc: the closeness centrality of $s$; delta: an array that stores the $\delta_s(*)$.
S1: Initialization:
  1) Define variables in shared memory: an array named as dist storing the distance from the
     starting node; an array named as sigma storing the number of shortest paths; an array named
     as delta storing the $\delta_s(*)$; an variable named as level indicating the current level at which
     the breadth-first search is; an variable named as repeat controlling the execution of threads.
  2) Let $dist[s] = 0$, $sigma[s] = 1$, and $dist[v] = -1$, $sigma[v] = 0$ for $v \neq s$, and
     $delta[v] = 0$ for each node $v$, and $level = 0$, $repeat = True$.
S2: Breadth-first search starting from node s:
  While $repeat = True$, do:
      If threadid == 0 then repeat = False
      For v = threadid to n-1with interval of threadid, do: //v is the node allocated to threadid.
          For each adjacent node adj of v, do:
              If dist[adj] == level and dist[v] == -1 then dist[v] = level + 1 and repeat = True
              If dist[adj] == level and dist[v] == level + 1 then sigma[v] += sigma[adj]
          End for
      End for
      If threadid == 0 then level += 1 // move to the next level
      __syncthreads ()
  End while
S3: Reverse Deduction:
  While level > 1 do:
      For v = threadid to n-1with interval of threadid, do:
          If dist[v] == level then do:
              For each adjacent node adj of v, do:
                  If dist[adj] + 1 == dist[v] then do:
                      Atomicadd(delta[adj], sigma[adj]/sigma[v]*(1 + delta[v]))
                  End if
              End for
          End if
      End for
      If threadid == 0 then level -= 1
      __syncthreads ()
S4: Compute s's closeness centrality based on dist according to Eq (2) and store it in cc.
```

From the descriptions above, it can be seen that the parallel computing of the betweenness centrality and the closeness centrality can be performed in one implementation. It is so that we implement the parallel computing of these two centralities in one parallel algorithm, named as *Parallel Algorithm I*, which is shown in Algorithm 1. It is noted that only the computation within one block allocated with one node is shown in Algorithm 1. In the case where the block is allocated with more than one node, the steps of S2–S4 are required to execute for each allocated node. And all the blocks are executed in parallel to obtain the betweenness centrality and the closeness centrality. See Algorithm 1, the first thing is the initialization which defines necessary variables such as *dist*, *sigma*, *delta*, *level* and *repeat* and assign initial values to them (see S1). The variable *repeat* is used to control the execution of each thread, which we will describe later. Then, the breadth-first search is performed starting from the starting node s in parallel: each thread iteratively checks the nodes that are allocated to it, and visit any of them if it has neighbors at the current level (see the outer for loop in S2). This process is repeated several times until each node is visited (corresponds to the case where *repeat* = *False*). Actually, one execution of this process means that the nodes at the same level have been visited by the breadth-first search. Visiting a node means that its *dist* value and *sigma* value are updated and the variable *repeat* is assigned to be True once a new node is visited (see the two if statements in the inner loop of S2). Besides, a thread that changes the variables *repeat* and *level* is required, which is set to be thread with id 0 (see the first and the last if statements). __syncthreads () is used to synchronize the threads in this block. After the breadth-first search is done, the reverse deduction is performed in S3, which calculates the number of the shortest paths of each node from the starting node s, stored in *dist*, and $\delta_s(*)$, stored in *delta*. Finally, the closeness centrality of node s is obtained based on the *dist* according to Eq (2) (see S4), which can be also done by one thread, e.g., thread with id 0.

Time complexity. Suppose there are n_b blocks with *blockdim* threads in each block. For a node in a block, the breadth-first search takes time complexity of $O(\lceil\frac{n}{blockdim}\rceil \cdot level_{max} \cdot \bar{d})$ since each thread handles up to $\lceil\frac{n}{blockdim}\rceil$ nodes. In addition, each of these nodes is processed at most $level_{max}$ rounds, and all the neighbors of a node, which are $\bar{d}$ on average, are required to be checked in each round of processing. The accumulation of *dist* values over all the nodes takes time complexity of $O(\frac{n}{blockdim} + blockdim)$. This is because that the *dist* values of the nodes handled by the same thread can be accumulated in this thread in $O(\frac{n}{blockdim})$, and summing the cumulative values from different threads takes $O(blockdim)$ since this operation is required to be performed serially. The time complexity of the reverse deduction is approximated to $O(\lceil\frac{n}{blockdim}\rceil \cdot level_{max} \cdot \bar{d})$ since it is the reverse process of the breadth-first search. All the threads can add the *delta* values to the resulting betweenness centrality in parallel since the nodes handled by different threads are different, which results in time complexity of $\mathrm{O}(\frac{n}{blockdim})$. All the blocks are running in parallel, so the time complexity of *Parallel Algorithm I* is approximated to $O(\lceil\frac{n^2}{n_b \cdot blockdim}\rceil \cdot level_{max} \cdot \bar{d})$. It should be noted that here we assume that the threads with the same id (e.g., thread0) from different blocks do not update a node's betweenness centrality simultaneously, which is reasonable since a node handled by thread0 is probably at different levels in different blocks. It can be inferred that the corresponding serial algorithm takes the time complexity of $O(\bar{d}n^2)$, it is thus that the parallel algorithm is much faster than the serial one, which is also demonstrated by the experiments (see the experimental part). This is because that n_b, the number of blocks, and *blockdim*, the number of threads in each block, are usually not small, while $level_{max}$, the maximal level, and $\bar{d}$, the average number of neighbors, are commonly small since most real networks display the phenomenon of small world and are scale-free.

4.2.2. The parallel calculation of PageRank centrality

```
Algorithm 2 Parallel Algorithm II
Input: G = (V, E): the network with nodes numbered from 0 to $n-1$; $\alpha$: a scaling factor; MaxT: the number of iterations.
Output: pr: an array used to store the PageRank centrality of each node.
S1: Initialization:
  1) Define global variables: an array named as pr storing the PageRank centrality; a variable named as pr_sum storing the sum of PageRank centralities of all the nodes.
  2) Let $pr[v] = 1/n$ for each node v and pr_sum = 0
S2: Calculation of the PageRank centrality:
  For t = 1 to MaxT with interval of 1, do:
    1) Update PageRank centrality:
      For $v = blockid * blockdim + threadid$ to n-1with interval of $n_b \cdot blockdim$, do:
      //v is the node allocated to threadid in blockid, $n_b$ is the number of blocks
          newRank = 0
          For each adjacent node adj of v, do:
              newRank += $pr[adj]/d_{adj}$
          End for
          $newRank = \alpha * newRank + (1-\alpha)/n$
          atomicAdd(pr_sum, newRank) //add newRank to pr_sum
          $pr[v] = newRank$
      End for
      __syncthreads () // threads synchronization, all the threads have updated the allocated nodes
    2) Normalize the PageRank centrality:
      For $v = blockid * blockdim + threadid$ to n-1with interval of $n_b \cdot blockdim$, do:
          $pr[v] = pr[v] / pr\_sum$
      End for
      __syncthreads ()
  End for
```

To update the PageRank centrality of a given node v_i, only its neighbors and the PageRank centralities of the neighbors are required (see Eq (3)). Thus, to parallelize the PageRank centrality, we can directly allocate all the nodes to different threads, rather than blocks, and then each thread performs the update operation shown in Eq (3) over the nodes allocated to it. The details of the parallel computing of PageRank centrality are referred to *Parallel Algorithm II*, which is shown in Algorithm 2. The first thing is the initialization which defines necessary variables including *pr*, storing the PageRank centrality for each node, and *pr_sum,* storing the sum of the PageRank centralities of all the nodes (see S1). Then, each thread calculates the PageRank centrality of the nodes that are allocated to it in parallel: the node id of the allocated nodes can be obtained according to the block id and thread id (see the first for statement in the first substep of S2); for each allocated node, its new PageRank centrality, stored in the local varibale *newRank*, is obtained according to Eq (3); the new PageRank centrality is also added to the *pr_sum*, which is used to normalize the PageRank centrality in the second substep of S2; besides, all the threads need to synchronize with each other in the two substeps in S2, which is guaranteed by the function "__syncthreads ()". The PageRank centralities of all the nodes are obtained after the two substeps in S2 are successively repeated *MaxT* times.

Time complexity. Suppose there are n_b blocks with *blockdim* threads in each block, which results in $n_b \cdot blockdim$ threads. For a thread, it takes time complexity of $O(\bar{d})$ on average to compute the new PageRank value of a node allocated to it, since each node has $\bar{d}$ neighbors on average. All the threads run in parallel, thus the time complexity of *Parallel Algorithm II* is approximated to $O(\left\lceil\frac{n}{n_b \cdot blockdim}\right\rceil \cdot \bar{d})$. It can be inferred that the corresponding serial algorithm takes the time complexity of $O(\bar{d}n)$, it is thus that the parallel algorithm is faster than the serial one, which is also demonstrated by the experiments (see the experimental part).

5. Experimental result and analysis

In this section, we conduct experiments to evaluate the parallel algorithms designed in this paper. Before presenting the experimental results, we briefly describe the used networks and the experimental settings.

5.1. Datasets and experimental settings

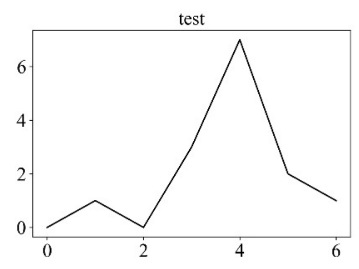
The networks used in the experiments are shown in Table 1. The first one, i.e., test, is a synthetic network and the remained ones are real networks from the Stanford Network Analysis Platform[1] and the Open Data Visualization website[2]. The size of the networks ranges from tens to hundreds of thousands of nodes and tens to millions of edges. According to the size, the networks are classified into small networks (including test, email-enron and email-univ) and large networks (including muase-facebook, wave and com-amzon). In addition, we list the average degree and the degree assortativity of each network (see the last two columns in Table 1) and show the degree distribution of each network, which is shown in Figure 3. From Table 1 it can be seen that the average degree of each network is not large, and the used networks are diverse on the degree assortativity. From Figure 3 it can be seen that all the networks follow the power law distribution on degree except the networks of test and wave, which means that all the networks except test and wave are scale-free. Overall, the used networks are diverse on topology.

[1] http://snap.stanford.edu/data/index.html

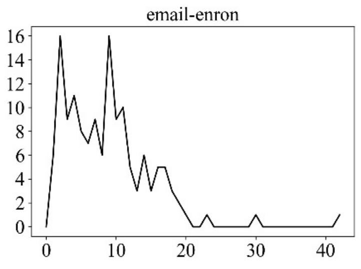
[2] https://networkrepository.com, last accessed 2021/05/23

Table 1. Dataset.

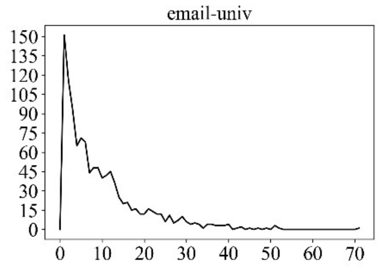
Dataset	Number of nodes	Number of sides	Average Degree	Degree Assortativity
test	14	27	3.86	-0.104
email-enron	143	623	8.72	-0.091
email-univ	1133	5451	9.62	0.033
musae-facebook	22,470	171,002	15.20	0.040
wave	156,317	1,059,331	13.25	-0.38
com-amazon	334,863	925,872	5.53	-0.62



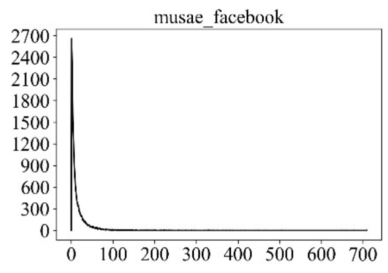
(a)



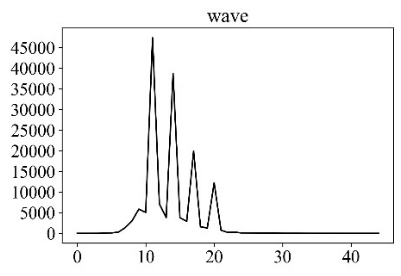
(b)



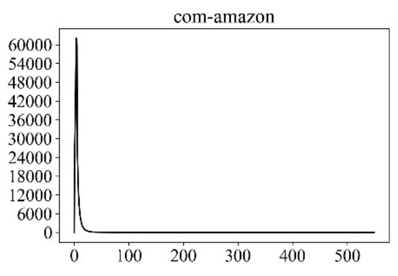
(c)



(d)



(e)



(f)

Figure 3. The degree distribution graph of test (a), the degree distribution graph of email-enron (b), the degree distribution graph of email-univ (c), the degree distribution graph of musae-facebook (d), the degree distribution graph of wave (e) and the degree distribution graph of com-amazon (f).

Our focus in this paper is to design parallel algorithms for three widely used centrality metrics with the aim of speeding up the calculation. Therefore, we evaluate the designed parallel algorithms by comparing their running times with the ones cost by the serial algorithms. The number of iterations, the scaling factor for PageRank centrality are respectively set to be 100 and 0.85. All the implementations[3] are run on a NVIDIA GeForce GTX 1050 (2G).

[3] The implementations of the two parallel algorithms have been uploaded to http://github.com/Huyanmei123/Parallel-algorithms

5.2. Experimental results and analysis

5.2.1. Experimental results over small networks

The running times consumed by different algorithms over the small networks are shown in Figure 4. It is noted that the calculation of the betweenness centrality and the closeness centrality are performed in one implementation, including the parallel computing, which is implemented in *Parallel Algorithm I*, and the serial computing. It can be seen that, in general, the parallel algorithms have no significant advantage over serial algorithms; in details, the *Parallel Algorithm I* costs more running time over the network email-enron, and the *Parallel Algorithm II* costs more running times over all the small networks. This is because that these networks are too small and the calculation of node centrality in these small networks only costs a very little time, while the parallel algorithms need extra time to copy data between main memory and GPU memory, and manage the GPU and the threads in it, and the extra time caused by GPU probably is more than the computing time.

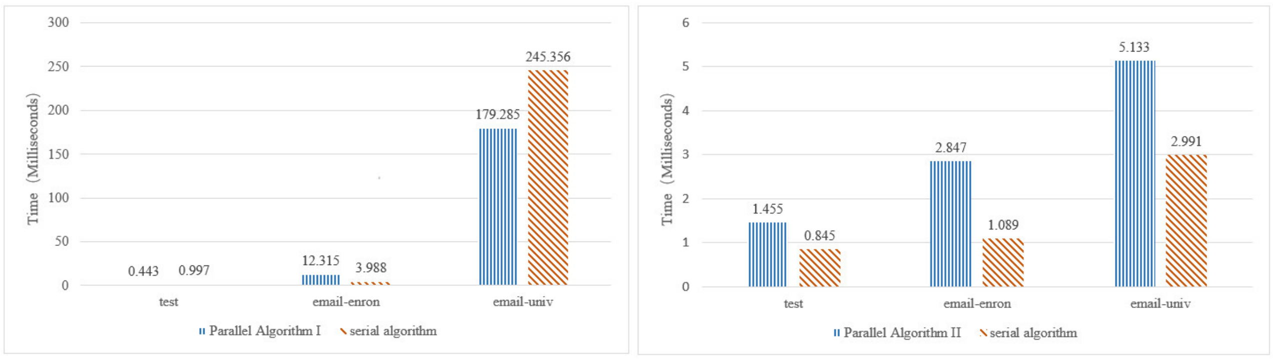


Figure 4. The running times of different algorithms over small networks.

5.2.2. Experimental results over large networks

The running times consumed by different algorithms over the large networks are shown in Figure 5. It can be seen that the parallel algorithms significantly speed up the calculation of node centrality in large networks. Moreover, the time required by serial computing of the betweenness centrality and the closeness centrality increases dramatically when the network size becomes large, but the time required by *Parallel Algorithm I* increases much slower. See the left figure in Figure 5, the running time by serial algorithm over com-amazon (more than 4 days) is much more than that over wave (about 3.6 hours), while *Parallel Algorithm I* does not take that much extra time over com-amazon (about 5.1 hours) compared with wave (about 1.7 hours). In addition, although the *Parallel Algorithm II* speeds up the calculation of PageRank centrality, the difference between the *Parallel Algorithm II* and the corresponding serial algorithm is relatively small (see the right figure in Figure 5); this is because that the serial algorithm is implemented by the PageRank function in igraph [30], which has been accelerated by prpack and arpack[4]. Anyway, from the results we can see that the parallel algorithms indeed speed up the calculation of node centralities in large networks which are common in real world.

[4] https://www.caam.rice.edu/software/ARPACK

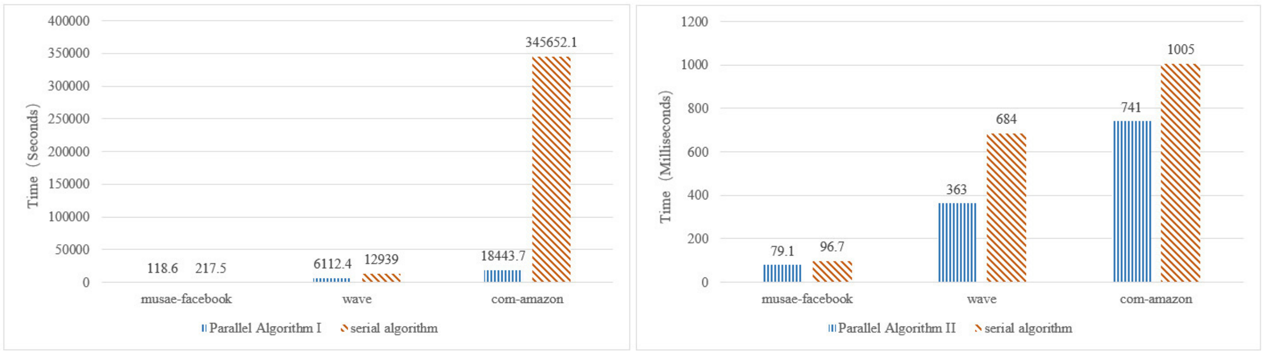
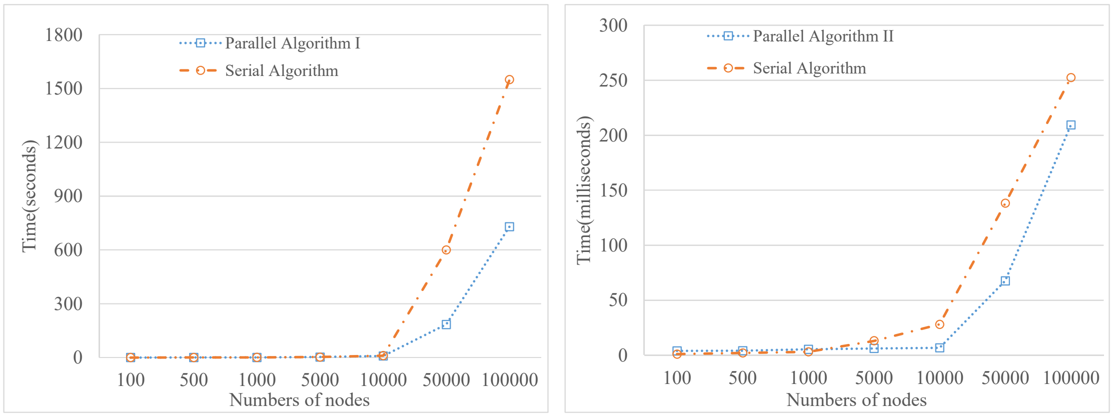


Figure 5. The running times of different algorithms over large networks.

5.2.3. Experimental results over networks with increasing scale



Figuer 6. The Comparison of the running times by the parallel algorithms (left for *Parallel Algorithm I* and right for *Parallel Algorithm II*) and the corresponding serial algorithms.

To further test the designed parallel algorithms, we compare the parallel algorithms with the corresponding serial algorithms on networks with increasing scale. Particularly, we first use BA model to generated several scale-free synthetic networks with 100, 500, 1000, 5000, 10,000, 50,000 and 100,000 nodes, respectively. Then, we run the parallel algorithms and the corresponding serial algorithms on these synthetic networks. The resulting running times are shown in Figure 6. From the figures it can be seen that the parallel algorithms are much faster than the corresponding serial algorithms on large networks. For the *Parallel Algorithm I*, it costs running times almost equal to the serial algorithm on the networks with nodes no more than 10,000; but with the increase of the network scale it becomes faster and faster than the serial algorithm. For *Parallel Algorithm II*, it costs running times almost equal to the serial algorithm on the smallest network, and on other networks it is faster than the later one; but the superiority is not that huge as *Parallel Algorithm* I, which may be because that the serial computation of the PageRank centrality is not that slow compared with the ones of betweenness centrality and closeness centrality and the used serial implementation is accelerated in the igraph package. Anyway, the parallel algorithms are faster than the corresponding serial algorithms with the increase of the network scale, which is consistent with the result before. This result means that parallel algorithms have superiority over serial algorithms on large networks, but on small networks the former ones have no superiority, which may be because that the parallel algorithms needs extra time to manage the blocks and threads and exchange data between GPU and CPU.

5.2.4. Analysis of different centrality metrics

To explore the performance of different centrality metrics and observe the relationships between different centrality metrics, we take the smallest network, i.e., test, as an example, see Figure 7(a) for the network structure of test. Table 2 lists the centrality values of each node. For better illustration, we visualize the network test by scaling each node proportional to the centrality value, see Figure 7(c),(d). It can be seen that node 6, which is in the overlap between two communities, obtains the highest value in terms of the closeness centrality and the betweenness centrality, especially that it achieves 42.666 on the betweenness centrality. This indicates that node 6 is a crucial node in the network test, as evidenced by its location: node 6 is the sole node that links the two communities and connects to numerous members of those communities.

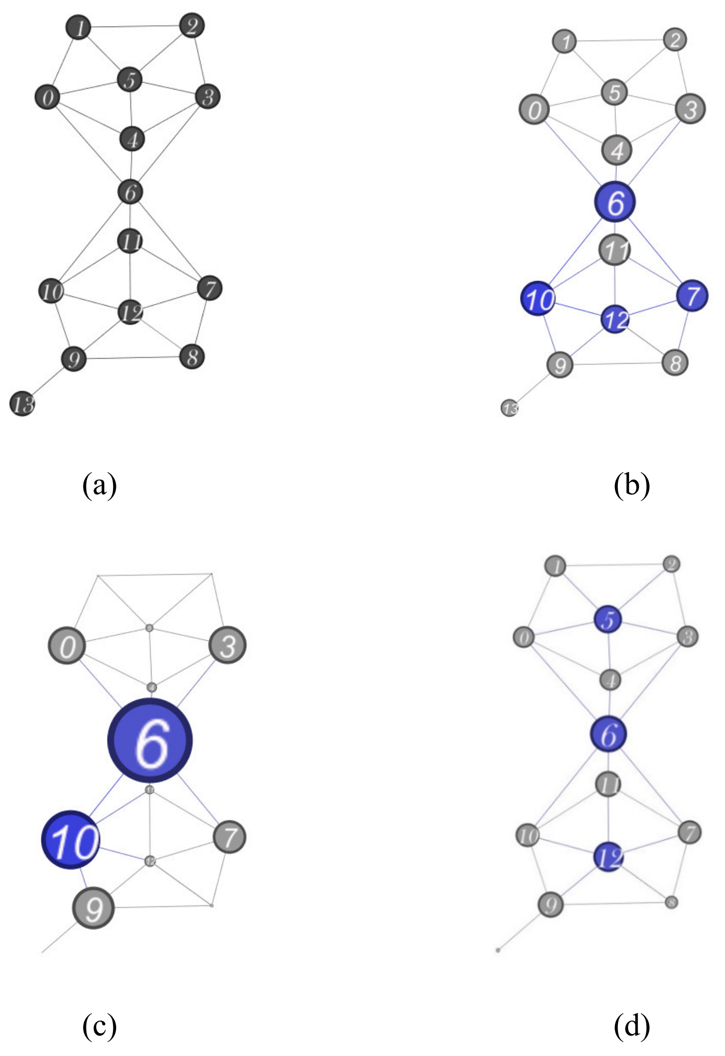


Figure 7. The network structure of test (a), the visualization of test with each node sized proportional to the closeness centrality value (b), the betweenness centrality value (c) and the PageRank centrality value (d).

Table 2. The centrality values of the nodes in the network test.

Nodes	CC	BC	PR
0	0.481	11.166	0.072
1	0.371	0.5	0.057
2	0.371	0.5	0.057
3	0.481	11.166	0.072
4	0.481	3	0.071
5	0.394	2.333	0.089
6	0.619	42.666	0.103
7	0.500	9.833	0.073
8	0.406	1	0.059
9	0.419	12.5	0.083
10	0.520	17.333	0.074
11	0.500	2.666	0.072
12	0.433	3.333	0.092

In addition, nodes 7 and 10 also have high closeness centrality values since they are very close to other members in the same community and are also connected to node 6 which is further close to other members of the upper community; node 10 has high betweenness centrality value while node 7 has lower betweenness centrality value, this is because that node 10 is connected to nodes 6 and 13 while node 7 is connected to node 6 but not node 13, which makes the former one is more important to the information propagation between nodes in the upper community and the bottom community; node 6 has the highest PageRank value than other nodes.

To further explore the relationships between different centrality metrics, we calculate the Pearson correlation coefficients between different centrality values over different networks, see Table 3 for the results. We can observe that the three centrality metrics are highly correlated over the small networks, especially in the test and email-univ; however, the three centrality metrics are less correlated over the large networks, and in certain situations, the correlation is quite low. Moreover, the PageRank centrality and the betweenness centrality are the most correlated; although the closeness centrality and the betweenness centrality are both based on the shortest paths between, they are only weakly related in most situations.

Table 3. The Pearson's correlation coefficients between different centralities.

Networks	CC-BC	CC-PR	BC-PR
test	0.798216	0.699796	0.607519
email-enron	0.553832	0.693872	0.85
email-univ	0.670686	0.814789	0.8927
musae-facebook	0.205396	0.444965	0.600013
wave	0.17173	0.040423	0.193564
com-amazon	0.17056	0.141363	0.521867

6. Conclusions

Node centrality is an important problem in network analysis and can be applied to many practical applications such as finding the most important person in Weibo and seeking the fittest testers for a new product. However, with the increase of network data, calculating node centrality by serial computing cannot meet the requirements of practical applications. On the other hand, GPU, as a good alternative to parallelize many tasks, has developed rapidly and has the ability to handle large networks. Therefore, we propose to apply GPU to parallelize the calculation of three widely used centralities and conduct experiments to evaluate the proposed parallel algorithms over several networks. The experimental results show that over small networks the parallel algorithms does not have advantage, and in some cases they even consumes more time than serial algorithms; while over large networks, the parallel algorithms can speed up the calculation significantly. Moreover, we also analyze the correlation between different centralities. The analysis shows that the closeness centrality and the betweenness centrality are weakly correlated, although both of them are based on the shortest path.

Compared with related work in the literature, we design and implement the parallel algorithms for different centralities using GPU, and conduct experiments over larger networks. However, the parallel algorithms are naïve, we will improve the parallel algorithms and conduct more experiments over more networks to deeply understand the centralities in future work.

Acknowledgments

This work is supported by Natural Science Foundation of China (No. 61802034), Sichuan Science and Technology Program (No. 2021YFG0333), National Key Research and Development Program of China (No. 2019YFC1509602) and the Digital Media Science Innovation Team of CDUT (Grant 10912-kytd201510).

Conflict of interest

All authors declare no conflicts of interest in this paper.

References

1. N. Safari-Alighiarloo, M. Taghizadeh, M. Rezaei-Tavirani, Protein-protein interaction networks (PPI) and complex diseases, *Gastroenterol. Hepatol. Bed. Bench.*, **7** (2014), 17–31.
2. B. Duo, Q. Wu, X. Yuan, Anti-jamming 3D trajectory design for UAV-enabled wireless sensor networks under probabilistic LoS channel, *IEEE Trans. Veh. Technol.*, **69** (2020), 16288–16293. https://doi.org/10.1109/TVT.2020.3040334
3. R. Zafarani, M. A. Abbasi, H. Liu, *Social media mining: an introduction*, Cambridge University Press, (2014), 41–49. https://doi.org/10.1017/CBO9781139088510
4. Y. LeCun, Y. Bengio, G. Hinton, Deep learning, *Nature*, **521** (2015), 436–444. https://doi.org/10.1038/nature14539

5. X. Li, M. Yin, Hybrid differential evolution with biogeography-based optimization for design of a reconfigurable antenna array with discrete phase shifters, *Int. J. Antennas. Propag.*, (2011), 685629. https://doi.org/10.1155/2011/685629
6. X. Li, M. Yin, Design of a reconfigurable antenna array with discrete phase shifters using differential evolution algorithm, *Prog. Electromagn. Res.*, **31** (2011), 29–43. http://dx.doi.org/10.2528/PIERB11032902
7. X. Li, S. Ma, Multi-objective memetic search algorithm for multi-objective permutation flow shop scheduling problem, *IEEE Access*, **4** (2016), 2154–2165. https://doi.org/10.1109/ACCESS.2016.2565622
8. R. Cypher, J. Sanz, *The SIMD model of parallel computation*, Springer, 1994. http://dx.doi.org/10.1007/978-1-4612-2612-3
9. D. A. Bader, K. Madduri, Designing multithreaded algorithm for breadth-first search and st-connectivity on the Cray MTA-2, in *International Conference on Parallel Processing*, IEEE, (2006), 523–530. https://doi.org/10.1109/ICPP.2006.34
10. A. Siriam, K. Gautham, K. Kothapalli, Evaluating centrality metrics in real-world networks on GPU, 2009. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.622.9442
11. A. McLaughlin, D. A. Bader, Scalable and high performance betweenness centrality on the GPU, in *International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, (2014), 572–583. https://doi.org/10.1109/SC.2014.52
12. Y. Jia, V. Lu, J. Hoberock, et al., Edge v.s node parallelism for graph centrality metrics, in *GPU Computing Gems* (eds. W. W. Hwu), Elsevier, (2012), 15–28. https://doi.org/10.1016/B978-0-12-385963-1.00002-2
13. H. Yin, A. R. Benson, J. Leskovec, The local closure coefficient: a new perspective on network clustering, in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, ACM, (2019), 303–311. https://doi.org/10.1145/3289600.3290991
14. Y. M. Hu, B. Yang, H. S. Wong, A weighted local view method based on observation over ground truth for community detection, *Inform. Sci.*, **355** (2016), 37–57. https://doi.org/10.1016/j.ins.2016.03.028
15. Y. M. Hu, B. Yang, Enhanced link clustering with observations on ground truth to discover social circles, *Knowl-Based. Syst.*, **73** (2015), 227–235. https://doi.org/10.1016/j.knosys.2014.10.006
16. B. Elbirt, *The nature of networks: a structural census of degree centrality across multiple network sizes and edge densities*, Ph.D. thesis, State University of New York at BuffaloIn, 2007.
17. K. F. Cheung, M. G. H. Bell, J. J. Pan, An eigenvector centrality analysis of world container shipping network connectivity, *Transport. Res. E-Log.*, **140** (2020), 101991. https://doi.org/10.1016/j.tre.2020.101991
18. T. Ioanna, B. Y. Ricardo, B. Francesco, Temporal betweenness centrality in dynamic graphs, *Int. J. Data Sci. Anal.*, **9** (2020), 257–272. https://doi.org/10.1007/s41060-019-00189-x
19. I. G. Adebayo, Y. X. Sun, A novel approach of closeness centrality measure for voltage stability analysis in an electric power grid, *Int. J. Emerg. Electr. Power Syst.* **3** (2020). http://doi.org/10.1515/ijeeps-2020-0013
20. A. Hashemi, M. B. Dowlatshahi, H. Nezamabadi-pour, MGFS: A multi-label graph-based feature selection algorithm via PageRank centrality, *Expert Syst. Appl.*, **142** (2019), 113024. https://doi.org/10.1016/j.eswa.2019.113024

21. U. Brandes, A faster algorithm for betweenness centrality, *J. Math. Sociol.*, **25** (2001), 163–177. https://doi.org/10.1080/0022250X.2001.9990249
22. G. Zhao, P. Jia, A. Zhou, InfGCN: Identifying influential nodes in complex networks with graph convolutional networks, *Neurocomputing*, **414** (2020), 18–26. https://doi.org/10.1016/j.neucom.2020.07.028
23. S. Buß, H. Molter, R. Niedermeier, Algorithmic aspects of temporal betweenness, in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ACM, (2020), 2084–2092. https://doi.org/10.1145/3394486.3403259
24. E. Y. Yu, Y. P Wang, Y. Fu, Identifying critical nodes in complex networks via graph convolutional networks, *Knowl-Based. Syst.*, **198** (2020), 105893. https://doi.org/10.1016/j.knosys.2020.105893
25. S. Ahamed, M. Samad, Information mining for covid-19 research from a large volume of scientific literature, preprint, arXiv:2004.02085. http://arxiv.org/abs/2004.02085
26. L. E. Suárez, B. A. Richards, G. Lajoie, Learning function from structure in neuromorphic networks, *Nat. Mach. Intell.*, **3** (2021), 771–786. https://doi.org/10.1038/s42256-021-00376-1
27. S. Brin, L. Page, The anatomy of a large-scale hypertextual Web search engine, *Comput. Support. Coop. Work*, **30** (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X
28. U. Brandes, C. Pich, Centrality estimation in large networks, *Int. J. Bifurcat. Chaos.*, **17** (2007), 2303–2318. https://doi.org/10.1142/S0218127407018403
29. J. Liu, Z. Ren, Q. Guo, Node importance ranking of complex networks, *Acta Phys. Sinica.*, **62** (2013), 178901. https://doi.org/10.7498/aps.62.178901
30. G. Csárdi, T. Nepusz, The igraph software package for complex network research, *Int. J. Complex Syst.*, **1695** (2006), 1–9. http://igraph.sf.net