Future Generation Computer Systems 106 (2020) 1–13

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Efficient parallel algorithm for detecting influential nodes in large biological networks on the Graphics Processing Unit

Lei Xiao, Shuangyan Wang, Gang Mei *

School of Engineering and Technology, China University of Geosciences (Beijing), 100083, Beijing, China

ARTICLE INFO

Article history:
Received 11 July 2019
Received in revised form 30 November 2019
Accepted 25 December 2019
Available online 2 January 2020

Keywords:
Biological networks
Influential nodes
Degree Centrality
Clustering Coefficient
H-Index
Parallel algorithm

ABSTRACT

In biological networks, some nodes are more influential than others. The most influential nodes are those whose elimination induces a network collapse, and detecting these nodes is crucial in many circumstances. However, this is a difficult task when the size of the biological networks is large. In this paper, we have designed and implemented an efficient parallel algorithm for detecting influential nodes for large biological networks by exploiting a Graphics Processing Unit (GPU). The essential concept behind the proposed parallel algorithm is that several computationally expensive procedures in detecting influential nodes are redesigned and transformed into quite efficient GPU-accelerated primitives such as parallel sort, scan, and reduction. Four local metrics, including the *Degree Centrality* (DC), *Companion Behavior* (CB), *Clustering Coefficient* (CC), and *H-Index*, are used to measure the nodal influence. To evaluate the efficiency of the proposed parallel algorithm, five large real biological networks are employed in the experiments. The experimental results show that (1) the proposed parallel algorithm can achieve speedups of approximately 48∼94 over the corresponding serial algorithm; (2) compared to a baseline parallel algorithm developed on a multi-core CPU, the proposed parallel algorithm yields speedups of 5∼9 for DC and H-Index, while it is slightly slower for CB and CC due to the uneven degree distribution; and (3) when using DC and H-Index, the proposed parallel algorithm is capable of detecting the influential nodes in a large biological network consisting of 150 million edges in less than 3 s.

1. Introduction

In recent years, complex network analysis has received increasing attention. Many real complex systems can be abstractly regarded as complex networks for presenting the complexities of real systems [1], such as social networks, technological networks, information networks, and biological networks. Various methods have been proposed for mining information from complex networks. In particular, detecting the influential nodes in complex networks is a topic of interest drawing much attention in this research field [2–4].

Detecting influential nodes in complex networks can be exploited to mine the features and functions of these networks [2]. Much research has been conducted to rank and identify influential nodes in the aforementioned four network categories. In the detection of influential nodes in various networks, the first critical issue is to select or define the metrics for measuring the influence of each node; the second is to employ or develop specific algorithms to effectively and efficiently determine and rank the influential nodes.

Many metrics have been proposed for detecting the influential nodes in complex networks. These metrics can be roughly divided into two categories [4]: (1) ***local*** metrics that are calculated based on the local structures of networks and (2) ***global*** metrics that are calculated based on the global structures of networks. For the first category, the most commonly used metrics include the degree centrality, clustering coefficient, and H-Index [5]. For the second category, the most commonly used metrics are the betweenness centrality [6], closeness centrality, PageRank, *k*-core [7] computed using the *k*-shell decomposition [8], and the bidirectional *k*-core (B-core) [9].

Much research has been conducted to detect influential nodes in various networks [10–12]. For example, Zhou et al. [13] designed a two-stage mining algorithm (GAUP) to mine the most influential nodes in a social network on a given topic. Zhu et al. [14] proposed a new ranking method named SpreadRank to maximize the spread of influence ranking in social networks. Rahimkhani et al. [15] proposed a new algorithm based on the linear threshold model of influence maximization by first finding the community structures and then selecting a number of representative nodes. Deng et al. [16] proposed a novel model called PAV to capture the intrinsic relationships of different objects in bibliographic information networks and applied the PAV in the ACM Digital Library.

* Corresponding author.
E-mail address: gang.mei@cugb.edu.cn (G. Mei).

https://doi.org/10.1016/j.future.2019.12.038
0167-739X/

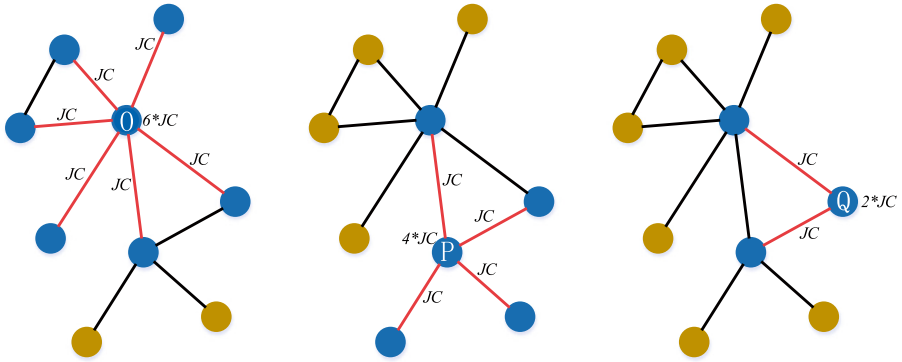


Fig. 1. Illustrations of the summations of *Jaccard* Coefficient.

Jiang et al. [17] described complex multiagent software systems as citation networks and presented a concept of extended group centrality to find the influential agent groups in complex multiagent software systems. Zareie et al. [18] ranked influential users in social networks by considering neighbor diversity and spread sphere for the spreading process. Wang et al. [19] identified the key conflict points in an aircraft state network based on complex network theory. Wang et al. [20] identified the critical points of road networks and revealed that local floods could induce large-scale abrupt failures of the entire road network.

Moreover, with the increasing size of complex networks, in particular online social networks, there are typically millions or even billions of nodes in these networks. For large complex networks, the detection of influential nodes is computationally expensive and thus requires improvement of the computational efficiency by designing theoretically fast algorithms and/or parallel algorithms. For example, Migallóna et al. [21,22] proposed parallel two-stage algorithms for solving the PageRank problem using a mixed MPI/OpenMP model. Jamour et al. [23] developed an efficient parallel algorithm for incremental betweenness centrality in large online social networks. Wang et al. [24] implemented a parallel algorithm for identifying influential nodes in dynamic social networks using OpenMP.

Most of the above research focuses on ranking and identifying influential nodes in social networks [25,26], while little work specifically focused on detecting influential nodes in biological networks. For biological networks, Mallik and Maulik [27] proposed a new framework for ranking biomolecules in a multi-informative uterine leiomyoma dataset using an eigenvector centrality-based approach. Ferraro et al. [28] found influential nodes for integration in brain networks using optimal percolation theory. Sun et al. [29] proposed a new framework to identify influential genes in protein–protein interaction networks by considering the heterogeneity of influence. Morone et al. [30] predicted the location of the most influential neural nodes involved in information processing in the brain.

In this paper, we propose an efficient parallel algorithm for detecting influential nodes for large biological networks by exploiting the massive computing capability of a modern GPU. To the best of our knowledge, this is the first work focusing on designing a GPU-accelerated parallel algorithm for detecting influential nodes in large biological networks.

The essential concept behind the proposed parallel algorithm is that several computationally expensive procedures in detecting influential nodes are redesigned and transformed into quite efficient primitives such as parallel sort, parallel scan, and parallel reduction. These parallel primitives are highly optimized, and the use of these parallel primitives can significantly improve the computational efficiency of the proposed algorithm. Moreover, four local metrics are used to measure the influences of nodes in the detection, and five large real biological networks are employed in the experiments to evaluate the performance of the proposed parallel algorithm.

The main contributions of this paper can be summarized as follows.

(1) We propose an efficient parallel algorithm on the GPU to identify influential nodes for large biological networks.

(2) We detect the influential nodes of five large real biological networks using four different local metrics by employing the proposed parallel algorithm.

The rest of this paper is organized as follows. Section 2 introduces four local metrics used to measure the influences of nodes. Section 3 describes the details of the proposed parallel algorithm for detecting the influential nodes in large biological networks. Section 4 presents five groups of experiments to evaluate the performance of the proposed parallel algorithm. Section 5 discusses the experimental results and the proposed algorithm. Finally, Section 6 draws several conclusions.

2. Background: Metrics for detecting influential nodes in large biological networks

In this work, four local metrics are used to measure the influences of nodes in biological networks, including (1) the Degree Centrality, (2) Companion Behaviors [3], (3) Clustering Coefficient, and (4) H-Index.

2.1. Metric 1: Degree centrality

The nodes which are of high Degree Centrality (DC) can significantly impact the other nodes because of their numerous connections (Eq. (1)).

$$C_i^{DEG} = deg(i) \tag{1}$$

where C_i^{DEG} denotes the DC of the node i, $deg(i)$ represents the degree of the node i.

2.2. Metric 2: Companion behaviors

The local metric Companion Behaviors (CB), was originally proposed by Wang and Mei [3] and based on the calculation of Jaccard Coefficients (JC) of edges. The JC embodies the differences of neighbors of two nodes, i.e., the relationship strength of two nodes (Eq. (2)). The relationship between two nodes could be weak if the JC of the two nodes is small. In contrast, the relationship between two nodes could be strong if the JC of the two nodes is large.

$$JC = (E|A, B) = \frac{|n_A \bigcap n_B|}{|n_A \bigcup n_B|} \tag{2}$$

where $JC = (E|A, B)$ denotes the JC of node A and node B; n_A denotes the set of neighbors of node A; n_B represents the set of neighbors of node B.

Generally, a node can significantly impact the other nodes, which are of strong relationship strength with the node. The CB of a node can be expressed as the summation of the JC of all neighbors and the node; see Eq. (3) and Fig. 1. The *Sums* of node O, node P, and node Q are $6 * JC$, $4 * JC$, and $2 * JC$, respectively.

$$CB(v_i) = \sum_{j=0}^{n} JC(v_j|v_i) \tag{3}$$

where $JC(v_j|v_i)$ indicates the JC of node i and its neighbor node j; and the $CB(v_i)$ indicates the summation of the JC of n neighbors and node i.

2.3. Metric 3: Clustering coefficient

The Clustering Coefficient (CC) C_i of node v_i is defined as the ratio of the actual number of edges E_i and the total number of possible edges $C_2^{k_i}$ between k_i neighbors of node v_i, i.e.:

$$C_i = \frac{E_i}{C_2^{k_i}} \tag{4}$$

2.4. Metric 4: H-index

The H-Index was originally proposed by Hirsch [31] in 2005 to estimate a scientist's cumulative research contributions, which is calculated based on an analysis of publication and citation data. The H-Index is originally defined as: "A scientist has index h if h of his or her papers have at least h citations each". In complex science, the H-Index is also employed as a metric to evaluate the influence of nodes in complex networks, which is redefined as: "A node has index h if h of its neighbors have at least degree h each".

3. Proposed parallel algorithm for detecting influential nodes in large biological networks

3.1. Overview of the proposed parallel algorithm

There are two main stages in the proposed parallel algorithm for detecting influential nodes in large biological networks. The first stage is to find the neighbors for each node in parallel. The key step in this stage is to list the neighbors of all nodes simultaneously using a parallel sort and parallel scan. The second stage is to calculate the CB, CC, and H-Index values of all nodes in parallel on the basis of the previously found neighbors of nodes.

The flowchart of the proposed parallel algorithm is illustrated in Fig. 2. After inputting the given network, we must first clean up the network by removing potential duplicate edges. Then, the indices of all nodes will be reordered since the original indices of the input nodes are probably noncontinuous. The noncontinuous indices of nodes cause serious difficulties in referring to the information of nodes in memory.

The first and the most important procedure in the proposed parallel algorithm is to find the neighbors of all nodes. We propose a straightforward and easy-to-implement parallel algorithm to realize this procedure. The essential concept behind this procedure is to sort all edges and then employ parallel scans to list the corresponding neighbors for each node. Details on this procedure will be introduced in the subsequent section.

After finding the neighbors of all nodes, another three metrics, i.e., the CB, CC, and H-Index, of each node will be calculated based upon the found neighbors. To calculate the CB of each node, we first calculate the JC of each edge in parallel and then accumulate the JCs of those corresponding edges of the target node. The accumulation can be very easily performed using a parallel reduction.

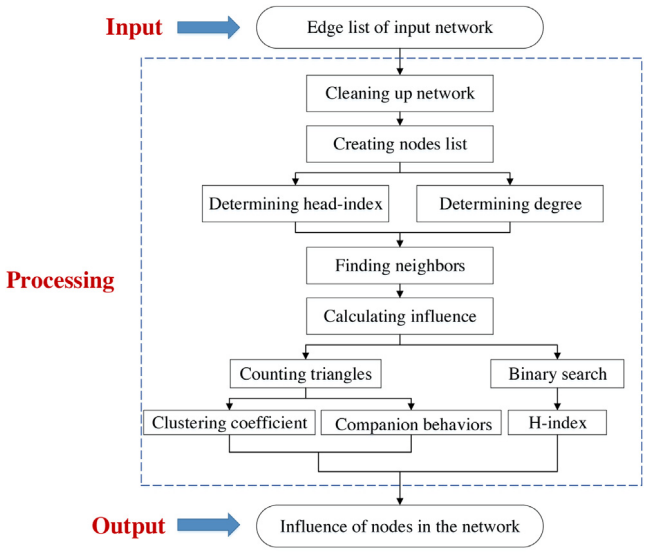


Fig. 2. Flowchart of the proposed parallel algorithms for detecting influential nodes.

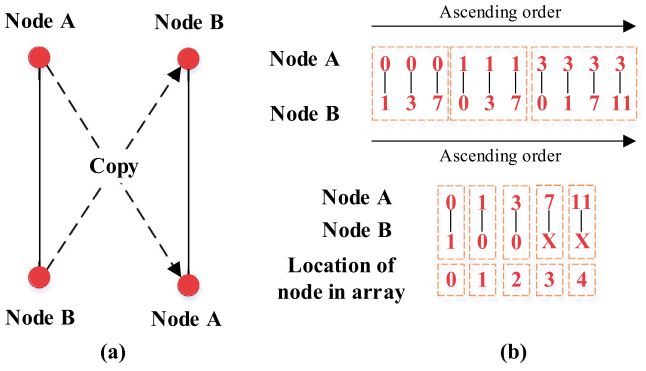


Fig. 3. Illustrations of finding real indices of nodes. (a) Copying an edge in the opposite direction. (b) Sorting edges according to nodal IDs and obtaining the real indices with the help of `unique` operation.

To calculate the CC of each node, we first count all triangles in the network and then accumulate the number of edges in the corresponding triangles for each node. The triangles are counted by looping over all edges in parallel, and the accumulation of the number of edges in the corresponding triangles is also realized by employing the parallel reduction operation. The H-Index of each node can be easily calculated by looping over all nodes in parallel.

After calculating the four metrics of each node, a complete list of the metrics of all nodes will be outputted for reference. An interested user could identify their required top-k influential nodes in the target large biological network. Note that the work in this paper develops an efficient and easy-to-implement parallel algorithm for detecting influential nodes in large biological networks rather than analyzing the characteristics, behaviors, or impact of the found influential nodes.

3.2. Procedures of the proposed parallel algorithm

3.2.1. Procedure 1: Cleaning up the input biological network

The objective of this procedure is to remove duplicate edges in the initially given biological networks. The cleaning up is

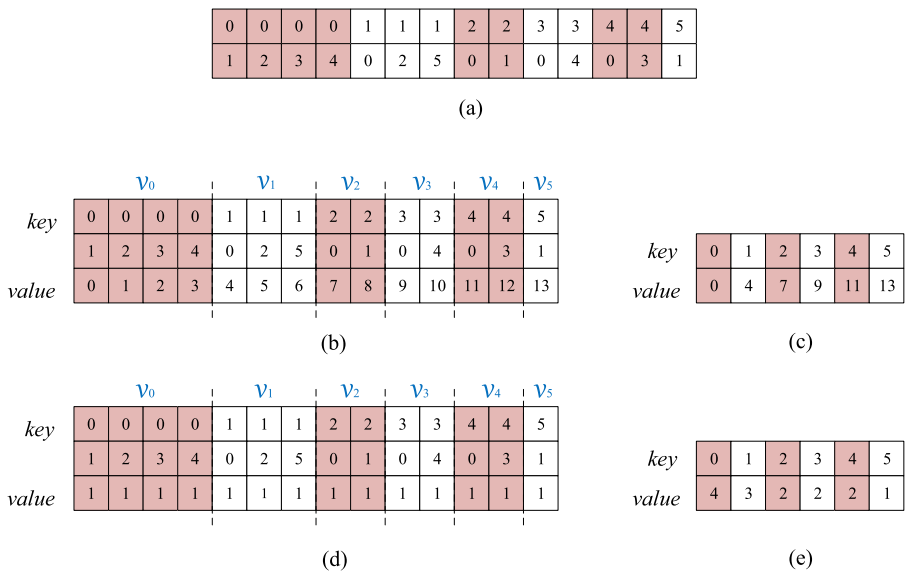


Fig. 4. Illustrations of finding neighbors of all nodes by parallel segmented scan. (a) illustrate the sorted edge list; (b) and (c) illustrate the parallel calculation of the head position of each segment; (d) and (e) illustrate the parallel calculation of the lengths of segments.

performed in parallel, and similar work has been conducted in our previous study [32].

The essential step in this procedure is to sort all edges according to the two nodal IDs of each edge. Suppose the first and the second nodes of an edge are noted as Node A and Node B, respectively, all edges are first sorted according to the ID of Node A in ascending order, and if two edges have the same IDs of A, then they are sorted according to the IDs of B. After the above sorting, duplicate edges with the same IDs of both A and B will be listed adjacently and thus can be identified by comparing the IDs of A and B for each pair of adjacent edges.

The above procedure can be easily conducted in parallel. First, the sorting of all edges can be easily parallelized on the GPU by invoking existing parallel primitives provided by CUDA libraries such as *thrust* [33]. Second, the identification of duplication edges by comparing the nodal IDs of adjacent edges can also be performed using scan. The parallel scan is one of the most commonly used parallel primitives, which has been well implemented on the GPU and integrated into the famous *thrust* library [33].

3.2.2. Procedure 2: Finding the real indices of all nodes

After removing duplicate edges, all nodes need to be uniquely indexed for further calculation. This is because the IDs of the nodes may not be continuously numbered. Thus, the IDs of nodes cannot be directly used as indices for referring to computer memory. In this case, all nodes must be uniquely indexed, but their IDs will remain [34].

The finding of the real indices of all nodes is also straightforward (Fig. 3). It is obvious that any nodes with the same IDs should have the same indices. After sorting all edges, the IDs of Node A of all edges are listed in ascending order. By simply comparing the IDs of Node A, unique indices can be sequentially assigned. This procedure can be easily realized in sequence: the index of the first Node A will be designated as 0, and if the second Node A has the same ID as that of the first A, then its index will also be 0; otherwise, its index will increase by 1 (i.e., $0 + 1$).

To perform the above procedure in parallel, the parallel scan is also conducted by employing a helper array with values of the IDs of Node A. The indices of all nodes can be obtained by performing a parallel `unique` operation. Those nodes with the same IDs will have the same indices.

3.2.3. Procedure 3: Finding the neighbors of all nodes

The degree of a node is the number of its neighbors. Additionally, on the basis of the sorted list of edges, a parallel scan is performed to find the neighbors of all nodes on the GPU. The essential ideas are illustrated in Fig. 4.

After sorting all edges first according to the IDs of Node A and then according to those of Node B, those edges having the same first nodes (e.g., the node with the ID 84) are listed adjacently and continuously. That is, those edges are in a segmented and continuous list. For each segment, i.e., those edges having the same first nodes, all the first nodes are the same; for example, the node with the ID 84, its neighbors are those second nodes of the edges in the above segment.

The task of finding the neighbors of all nodes is then mapped to determine those segments. This is to determine the head position and the length of each segment precisely. For all segmented lists of edges, the head position of each segment can be obtained by performing a segmented scan operation; see Fig. 4(b) and (c). The lengths of segments can be easily obtained by calculating the distance between any pair of adjacent head positions; see Fig. 4(d) and (e).

The segmented scan operation can be easily parallelized by invoking the corresponding parallel primitive integrated in the CUDA package. After that, the head positions of all segments can be obtained, and then a specific CUDA kernel is designed to calculate the lengths of all segments in parallel. Within the CUDA kernel, each thread is responsible for calculating the length of a segment.

3.2.4. Procedure 4: Calculating the JC values of all edges

After finding the neighbors of all nodes, the JC values of all edges can be calculated according to Eq. (2). Obviously, there is no data dependency between the calculations of the JC values of any two edges. That is, all JC values can be calculated simultaneously. Therefore, the calculation of the JC values of all edges can be easily parallelized on the GPU. A specific CUDA kernel is designed, and within the kernel, each GPU thread takes the responsibility to calculate the JC value of an edge. All JC values can be theoretically calculated concurrently.

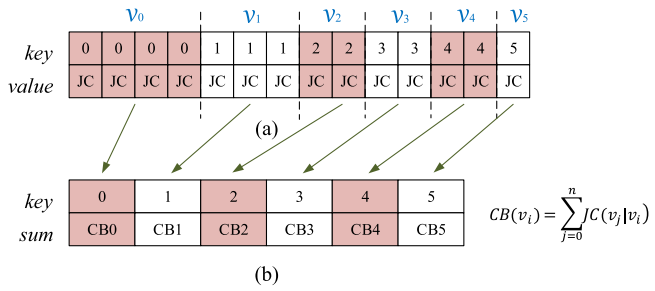


Fig. 5. Illustration of calculating CBs in parallel.

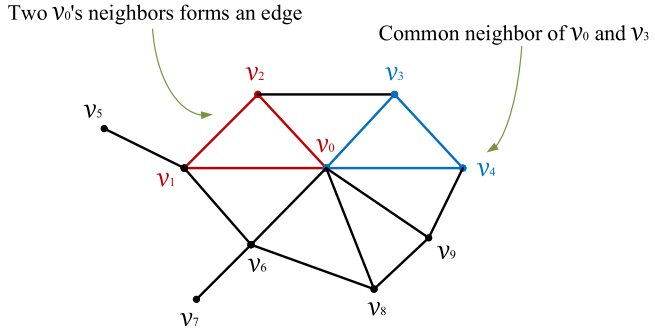


Fig. 6. Illustration of triangle structure in networks.

3.2.5. Procedure 5: Calculating the CB values of all nodes

The CB value of a node is also a local metric for measuring the nodal influence, which can be calculated by accumulating the JC values of those edges linking to the node (Eq. (3)). The accumulation of JC values cannot be directly parallelized due to race conditions. In this paper, we specifically design a parallel strategy for accumulating the JC values. The essential idea behind the parallel strategy is illustrated in Fig. 5.

For the sorted edges, the JC value of each edge was calculated in parallel by invoking a specifically designed CUDA kernel. Furthermore, the neighbors of each node have been recorded via the parallel segmented scan. That is, those edges linking to the same node have been collected in a continuous segment. Inspired by the concept of finding the real indices of nodes, the parallel segmented reduction can also be applied for the sorted edge list to accumulate the JC values of those edges located in the same segment.

More specifically, those edges located in the same segment are those that are linked to the same node. By performing a segmented reduction, the JC values of those edges located in the same segment can be accumulated. The results of the accumulation are exactly the required CB value of the node. Moreover, to improve the computational efficiency, the parallel segmented reduction can be utilized by invoking the corresponding parallel primitives integrated into the CUDA package to achieve the accumulation.

3.2.6. Procedure 6: Calculating the CC values of all nodes

The CC value of a node is also a local metric for measuring the nodal influence, which can be calculated by counting the triangles incident to the node, for example, the triangle incidents to node v_0 in Fig. 6. In this paper, the CC values of all nodes are calculated by (1) counting triangles in the biological network and (2) accumulating the number of triangles onto those nodes to which the triangles are incident. Note that both of the above two steps are conducted in parallel on the GPU.

The step of counting triangles in a biological network is also realized on the basis of the sorted edge list. After finding the neighbors of each node, for each edge in the sorted edge list, the finding of triangles is exactly to find the common neighbors for both the first and the second nodes of the edge. More specifically, suppose an edge e has two nodes n_a and n_b, if the nodes n_a and n_b have the same neighbor termed as n_c, then the three nodes n_a, n_b, and n_c can form a triangle in the biological network.

This step is suitable to be parallelized on the GPU since the finding of triangles for each edge can be performed independently. A CUDA kernel is specifically designed to count the number of triangles for all edges. Within the kernel, each GPU thread is invoked to count the number of triangles (i.e., the number of common neighbors of the two nodes of the same edge) for an edge. Note that only the number rather than the IDs of the common neighbors must be saved after counting.

After counting the number of triangles for each edge, the further step is to accumulate the numbers of triangles onto the corresponding nodes to calculate the CC values. Inspired by the idea of using the parallel segmented reduction to calculate the CB values of nodes based on JC values of edges, the accumulation is also performed in parallel using the segmented reduction. More specifically, in the segmented reduction, the keys are the IDs of the nodes, and the values are the number of triangles. Note that an edge needs to be copied and mapped twice because it has two nodes. After performing the segmented reduction in parallel according to the keys and values, the number of triangles incident to each node can be achieved, and finally, the CC value of the node can be calculated.

3.2.7. Procedure 7: Calculating the H-Index of all nodes

The calculation of the H-Index of all nodes is straightforward. After finding the neighbors of all nodes, for each node, all of its neighbors are first sorted according to the degree. Then, for the sorted list of neighbors, the binary search is performed to find the H-Index value. Obviously, there are no data dependencies between the determinations of the H-Index for any two nodes. That is, the calculation of the H-Index for a node can be carried out independently and simultaneously. A CUDA kernel is also designed to calculate the H-Index values of all nodes. Within the kernel, each thread is responsible for first sorting the neighbors according to the degree and then determining the H-Index value using a binary search.

3.3. Computational complexity of the proposed parallel algorithm

There are only unique and sort operations in Procedure 1. The running time is $O(n\log n)$, where n denotes the number of edges. With a unique operation and a single loop, it cost $O(n)$ to find real indices. The computational complexity of finding neighbors is $O(n\log n)$ due to the sorting. With a merge sort in the procedure, calculating the CB of a single node will run in $O(m\log m)$, where m denotes its degree. Similar to calculating CB, the running time of obtaining CC of a specific node is $O(m\log m)$. For the H-Index, sorting must be performed before the binary search, and the computational complexity is $O(n\log n)$.

3.4. Considerations for the parallelization

3.4.1. Use of data layouts: AoS and SoA

The data layout is the form of organizing data in computer memory. There are two typical data layouts: AoS and SoA; see the illustrations in Fig. 7. In GPU computing, different choices of the data layouts may have a strong influence on computational efficiency. In this paper, to develop the most efficient parallel algorithms and to examine the impact of data layouts, all GPU implementations of the proposed parallel algorithms are developed using both AoS and SoA.

```
Struct
__align(8)__
Edge {
     int NodeA;
     int NodeB;
};
Struct Edge
myEdges[N];

Struct __
align(16) Node{
     int ID;
     int degree;
     int head_index;
     float value;
};
Struct Node
myNodes[N];
```

(a) **AoS**

```
Struct
Edge
{
     int NodeA[N];
     int NodeB[N];
};
Struct Edge
myEdges;

Struct
Node{
     int ID[N];
     int degree[N];
     int head_index[N];
     float value[N];
};
Struct Node
myNodes;
```

(b) **SoA**

Fig. 7. Data layouts: AoS and SoA.

3.4.2. Use of efficient parallel primitives

The objective of this work is to design efficient parallel algorithms for detecting influential nodes in biological networks. Several very important and essential ideas are based on the use of parallel primitives such as parallel sort, parallel scan, and parallel reduction. The use of these efficient parallel primitives can help improve the computational efficiency of the proposed parallel algorithm. Therefore, when designing the parallel algorithms, the workflow and ideas of the algorithms are carefully considered to utilize those efficient parallel primitives as much as possible.

3.5. Implementation details

To comparatively evaluate the performance of the proposed parallel algorithm, we design two baselines, including (1) a serial version that will be introduced in Section 3.5.1, and (2) a parallel version implemented on multi-core CPU that will be described in Section 3.5.2. Moreover, the proposed parallel algorithm developed on the many-core GPU will be introduced in Section 3.5.2 as well.

3.5.1. The serial implementation

The serial implementation of the algorithm for detecting influential nodes in biological networks is straightforward. The key procedure is to find the neighbors of each node. This can be simply realized by serially looping over all edges of the network; for one node of an edge, the other node is one of its neighbors. Because the number of neighbors for each node cannot be determined before the finding, a dynamic array, `std::vector<int> neighbors`, must be allocated for each node to store the dynamically found neighbors.

First, we copy all edges by swapping their nodes and store them with the original edge so that all nodes can be found from each side. Then, the edges' vector will be sorted by the `std::sort()` function to eliminate duplicate edges. Third, `std::unique()` will be used to delete adjacent edges with the same nodes. The next procedure is to find read indices of nodes alongside their neighbors. In serial implementation, an `int` vector is arranged to store the ID of all nodes. Similar to the former procedure, `std::sort()` and `std::unique()` will be called to clean the list. Then, we perform a loop over all the edges to count the number of nodes with different IDs. For the finding neighbors, the serial implementation is different from the proposed method in Section 3.2.3. It is obvious that one node of an edge should be the neighbor of the other one. We only loop over all edges and store each node to the other's neighbor array. After finding the neighbors of each node, it is able to calculate the influential metrics.

The JC value of each edge can also be easily obtained by serially looping over all edges. For each edge, because the neighbors of its two nodes have been found, the common (i.e., the shared) neighbors of the two nodes of the same edge can also be identified. In this case, the number of common neighbors can be achieved. Meanwhile, the number of neighbors of the two nodes is obtained. Therefore, the JC value of the edge is easily calculated according to Eq. (2).

The metric CB value of a node is the sum of the JCs of those edges that share the node; see Fig. 1. In the above procedures, the JC values of all edges were calculated. Thus, the CB values of all nodes can be calculated by serially looping over all edges and adding the JC value of each edge onto the corresponding node. More specifically, in the serial looping over all edges, for each edge, its JC value is accumulated onto both of its two nodes. After looping over all edges, the CB values of all nodes are obtained.

The calculation process of CC is similar to that of CB. First, we use a loop over all edges to count triangles in the network. The basic idea of how to use triangles to calculate CC has been mentioned before. In the loop, we use a two-pointer merge algorithm [35], which yields significant acceleration compared with the brute force method. The accumulation of the number of triangles in serial version is almost the same as CB mentioned above, while the only difference is a special condition to avoid dividing by zero.

When calculating the metric H-Index, an additional array is allocated to store the neighbors' degree of each node. The function `std::sort()` is used to ensure that all the degrees are listed in order. Finally, we use a binary search to find the H-Index rapidly.

3.5.2. The parallel implementation

The parallel implementation of the proposed algorithm for detecting influential nodes in a large biological network is heavily dependent on the use of those efficient parallel primitives provided by the library *thrust*, including `thrust::sort()`, `thrust::unique_by_keys()`, and `thrust::reduction_by_keys()`. The parallel primitives are very efficient and easy to use in practice.

In the parallel implementation, one of the essential ideas to find the neighbors of all nodes is to first sort all edges according to their two nodes' indices and then to use the segmented scan to list the neighbors of each node in a continuous segment of an array. This solution is very similar to our previous research work [36,37]. The abovementioned sort of all edges are implemented using the efficient primitive `thrust::sort()`. Note that before sorting all edges, those edges should be stored in an array allocated by `thrust::host_vector(numEdge)` and then copied to `thrust::device_vector(numEdge)`. The segmented scan will then be performed by employing the primitive `thrust::unique_by_keys()`.

The basic idea of parallel implementation is almost the same wherever it is on the multi-core CPU or the many-core GPU. Therefore, the two parallel algorithms will be described together unless there are some differences. There are no obvious differences between the parallel implementation and the serial implementation in the cleanup procedure. However, note that the vectors to store edges are converted from `host_vector` into `device_vector`, which means the parallel primitives provided by *thrust* library will be called on the GPU rather than on the CPU. We find that in the parallel algorithm developed on the multi-core CPU, sorting with structure using Intel Threading Building Blocks (TBB) is always faster than the *thrust*. Therefore, in the rest

of this implementation, TBB will be used instead of *thrust* library in sorting.

In the next procedure, we use a new approach to find real indices of all nodes; see Section 3.2.2. We first use the second nodes of the edge as the key. Then, we use `thrust::unique()` to eliminate the same key so that the length of key vector will be reduced from the length of edge vector to the number of nodes. Finally, the real indices of nodes can be obtained by a specific CUDA kernel (or a `for` loop in parallel implementation developed on the multi-core CPU).

The most obvious difference between the serial and parallel versions is the way to find neighbors. Because the number of neighbors for each node cannot be determined before finding, we need to allocate dynamic arrays to store neighbors. Sorting structures with dynamic arrays are not allowed in *thrust* library; therefore, a new way to store neighbors is needed.

As described in Section 3.2.3, to find the number of neighbors, we use an auxiliary array initialized with 1 as the value, while the keys are first nodes' indices of all edges. Then, the *thrust* primitive `reduction_by_keys()` is called to accumulate all values with the same key. This means that edges with the same first node will be counting (i.e., the degrees of nodes). Similarly, we create the key by the same way while the value array is filled by a sequence number from 0 to the size of the array. The function `thrust::unique_by_keys()` will be used to remove those duplicate values with the same key. It is clear that only the location where a node first appears can remain. Finally, both the head position and the length of each segment are obtained.

After finding the neighbors of each node, the metric CB value of the node will be calculated in parallel. First, the JC value of each edge must be calculated. This step can be easily performed in parallel. A specific CUDA kernel is designed in which each GPU thread is responsible for calculating the JC of an edge, i.e., to count the common/shared neighbors for the two nodes of the same edge. Note that to be used in the CUDA kernel, the point of the arrays allocated by `thrust::device_vector` needs to be first obtained and then transformed into the kernel function.

The CB value of an edge can be achieved by accumulating the JC values of all its connected edges. This step of accumulation for all edges can be efficiently realized via the parallel segment reduction and invoking the parallel primitive `thrust::reduction_by_keys()`. More details are illustrated in Fig. 5 and described in the references [36,37].

After finding the neighbors of each node, the metric CC value of the node can also be calculated in parallel. First, all triangles in the biological network are found by looping over all edges in parallel. For an edge, if the two nodes if the edge, e.g., node A and node B, have the same neighbor node C, then the three nodes A, B, and C form a triangle structure in the network. This step can be easily performed in parallel. A specific CUDA kernel is designed in which each GPU thread is responsible for finding the triangles for each edge.

After finding all triangles in the network, for each node, the edges formed by a pair of its neighbors can be counted by looping over all triangles in parallel. For a triangle, any two nodes of its three nodes are the neighbors of the remaining node. By looping over all edges in parallel, the number of edges formed by two neighbors of a specific node can be achieved. For all nodes, the above step can be realized in parallel via the parallel segment reduction using the parallel primitive `thrust::reduction_by_keys()`.

There are no race conditions in calculating the H-Index; as a result, it can be well parallelized in a parallel architecture. However, there is still a difference between the parallel algorithm developed on the multi-core CPU and the proposed parallel algorithm developed on the many-core GPU. The additional array to store neighbors' degrees can also be stored in a dynamic array separately or in a huge array as a whole that is applied in GPU implementation. Using dynamic arrays only needs to sort some integers, while the other method needs to sort the structure, therefore it can achieve higher efficiency when implements on the multi-core CPU. However, using a segmented scan operation is more suitable for the proposed parallel algorithm developed on the many-core GPU. Those parallel implementations are well parallelized with helpful primitives.

Table 1
Specifications of the workstation computer for performing benchmark tests.

Specifications	Details
CPU	Intel Xeon Gold 5118 CPU
CPU Frequency (GHz)	2.30
CPU RAM (GB)	128
CPU core	48
GPU	Quadro P6000
GPU memory (GB)	24
CUDA cores	3840
OS	Windows 10 Professional
Compiler	VS2015 Community
CUDA version	v9.0

Table 2
Five real biological networks used for the experiments.

Real biological network	Number of nodes	Number of edges
ppi-walks	44 697	8 730 249
bio-heart_top	25 825	121 110 631
bio-skeletal_muscle_top	25 825	123 841 014
bio-embryo_top	25 825	146 713 307
bio-blood_plasma_top	25 825	156 621 439

4. Results

4.1. Experimental design

In this section, the experimental environment and the testing data employed for conducting the experimental benchmarks are introduced. Note that, two baseline algorithms are used, including (1) the serial version, and (2) the parallel version develop on the multi-core CPU.

4.1.1. Experimental environment

To evaluate the performance of the proposed GPU-accelerated parallel algorithm, we conduct five groups of benchmark tests on a workstation computer. The specifications of the employed workstation computer are listed in Table 1.

4.1.2. Testing data

To evaluate the computational efficiency of the proposed parallel algorithm, five groups of experimental tests are conducted for real biological networks. These employed networks listed in Table 2 are downloaded from: Stanford Network Analysis Project [38].

4.2. Experimental results

In this subsection, the computational efficiency of the proposed parallel algorithm for detecting influential nodes in five real biological networks is presented. For the five large real biological networks, the ranking of influential nodes is conducted according to four local metrics, i.e., DC, H-Index, CB, and CC. The computational efficiencies for the above four metrics are listed in Tables 3–6. Please note that due to the large size of the real

Table 3
Computational efficiency of ranking influential nodes according to the metric DC.

Real biological network	Serial time (ms)	Parallel time (ms)			Speedup		
		CPU	GPU-AoS	GPU-SoA	CPU	GPU-AoS	GPU-SoA
ppi-walks	8161	791	153	113	10.32	53.34	72.22
bio-heart_top	167 384	11 346	2409	1785	18.05	69.48	93.77
bio-skeletal_muscle_top	164 019	9632	2436	1822	17.03	67.33	90.02
bio-embryo_top	195 108	11 641	2916	2170	16.76	66.91	89.91
bio-blood_plasma_top	214 675	13 128	3131	2329	16.35	68.56	92.17

Table 4
Computational efficiency of ranking influential nodes according to the metric H-Index.

Real biological network	Serial time (ms)	Parallel time (ms)			Speedup		
		CPU	GPU-AoS	GPU-SoA	CPU	GPU-AoS	GPU-SoA
ppi-walks	8430	857	176	123	9.84	47.90	68.54
bio-heart_top	183 165	10 288	2812	2231	17.80	65.14	82.10
bio-skeletal_muscle_top	179 735	10 620	2872	2317	16.92	62.58	77.57
bio-embryo_top	214 687	12 834	3429	2759	16.73	62.61	77.81
bio-blood_plasma_top	233 273	14 422	3801	2897	16.17	61.37	80.52

Table 5
Computational efficiency of ranking influential nodes according to the metric CB.

Real biological network	Serial time (ms)	Parallel time (ms)			Speedup		
		CPU	GPU-AoS	GPU-SoA	CPU	GPU-AoS	GPU-SoA
ppi-walks	24 730	2344	389	267	10.55	63.57	92.62
bio-heart_top	N/A	608 363	1035 150	716 964	N/A	N/A	N/A
bio-skeletal_muscle_top	N/A	616 424	1157 150	785 040	N/A	N/A	N/A
bio-embryo_top	N/A	899 836	1672 770	1567 900	N/A	N/A	N/A
bio-blood_plasma_top	N/A	916 409	1805 370	1591 310	N/A	N/A	N/A

Table 6
Computational efficiency of ranking influential nodes according to the metric CC.

Real biological network	Serial time (ms)	Parallel time (ms)			Speedup		
		CPU	GPU-AoS	GPU-SoA	CPU	GPU-AoS	GPU-SoA
ppi-walks	25 334	2173	372	269	11.66	68.10	94.18
bio-heart_top	N/A	625 313	1106 490	732 834	N/A	N/A	N/A
bio-skeletal_muscle_top	N/A	614 399	1086 200	763 434	N/A	N/A	N/A
bio-embryo_top	N/A	870 565	1676 900	1532 960	N/A	N/A	N/A
bio-blood_plasma_top	N/A	908 909	1776 160	1380 840	N/A	N/A	N/A

biological networks, the serial computational time for both the metrics CC and CB is not benchmarked; see Tables 5 and 6.

According to the benchmark results, the speedups of the GPU-accelerated parallel algorithm over the corresponding serial algorithm are approximately 48∼94. For DC and H-Index, the parallel implementation developed on the many-core GPU always achieves the highest speedups. However, the baseline developed on the multi-core CPU is slightly faster than the proposed implementation for CB and CC except the first testing network. The above result will be discussed in Section 5.1.

Comparing the computational efficiency when using different data layouts, i.e., AoS and SoA, the version of GPU implementation when using the layout SoA is typically 30% faster than the version when using the layout AoS. In GPU computing, different choices of the data layouts may have a strong influence on computational efficiency. However, there is no evidence that one layout is always better than the other. Different data layouts may be better in different applications. In this paper, to develop the most efficient parallel algorithms and to examine the impact of data layouts, all GPU implementations of the proposed parallel algorithms are developed using both AoS and SoA. The benchmark results indicate that the SoA is better than AoS in the proposed parallel algorithm.

5. Discussion

5.1. Impact of the degree distribution on the efficiency

Benchmark results indicate that, for DC and H-Index, the proposed parallel algorithm developed on the many-core GPU performs much better than the baselines. The speedups of the GPU-accelerated parallel algorithm over the corresponding serial algorithm are approximately 48∼94. More specifically, sorting especially with structures is the most time-consuming step in these procedures. During parallel sorting, workloads distributed to each thread or core are quite balanced, which means that it is suitable to be parallelized. As discussed above, with more available cores, GPU could provide more speedups over multi-core CPU in this case.

However, for the CB and CC, the parallel algorithm developed on the multi-core CPU is slightly faster than the one on the many-core GPU for the four testing datasets. This is because the maximum degrees of these networks are quite large and the degree distributions of these test data are quite uneven. Unlike the process of finding neighbors, when calculating the CB and CC, it needs to use merge sorting in each thread of the kernel, which cannot be parallelized. The workload between GPU threads will become unbalanced while there are several nodes with extremely large degrees, which would significantly decrease the computational efficiency due to the synchronization of a warp of GPU threads. Figs. 8(b)–8(e) show that the degree distributions

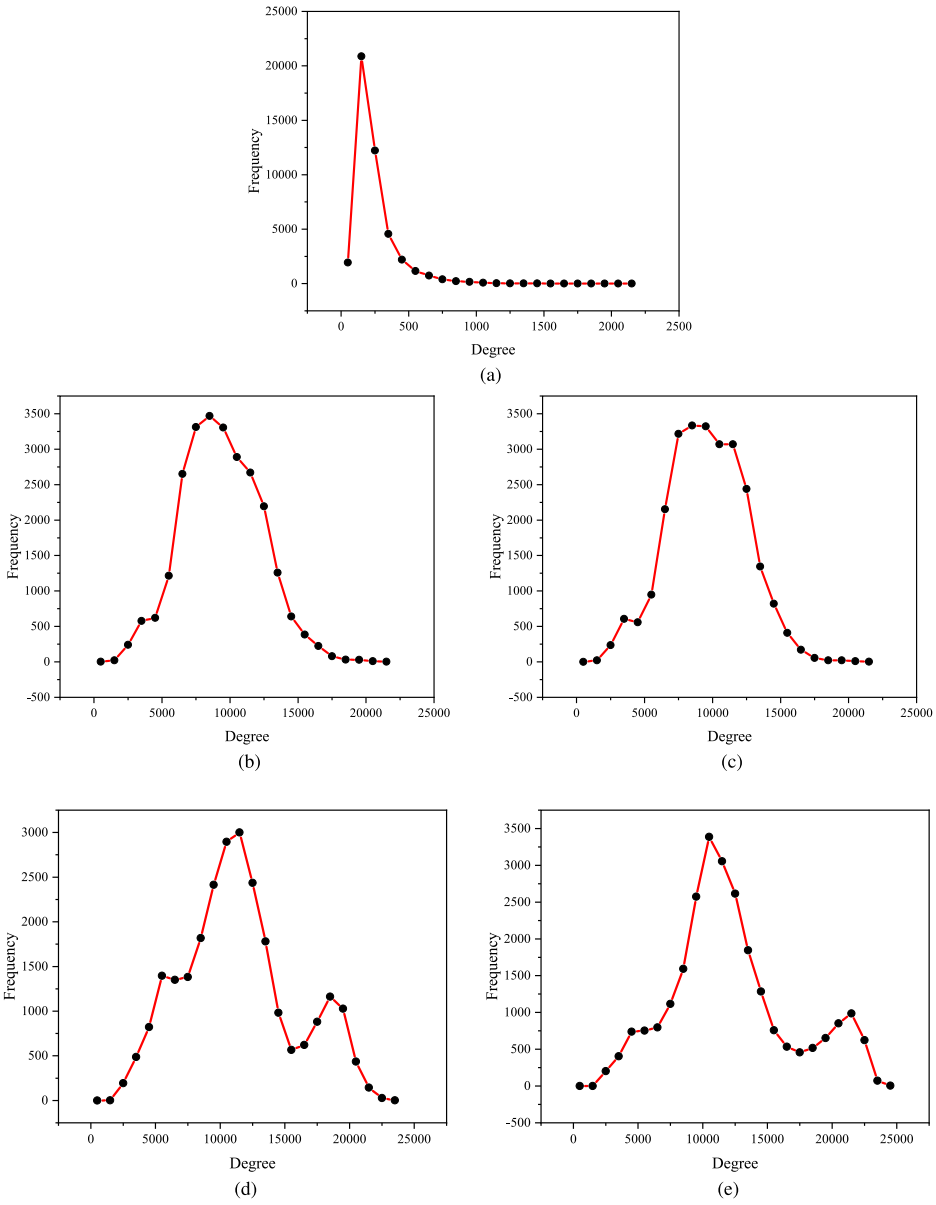


Fig. 8. Degree distributions of testing data. (a) ppi-walks. (b) bio-heart_top. (c) bio-skelet-al_muscle_top. (d) bio-embryo_top. (e) bio-blood_plasma_top.

of these networks are similar to the normal distribution. Moreover, the maximum degrees of these networks are more than 20,000. Therefore, an unbalanced workload occurs which means the computational efficiency of the proposed parallel algorithm developed on the many-core GPU will significantly decrease.

In contrast, there is a dynamic schedule option in modern CPU, which means the workloads distributed to each core could be balanced [39]. Therefore, it is reasonable that the parallel algorithm developed on the multi-core CPU perform better than the proposed parallel algorithm developed on the many-core GPU for these testing datasets.

The performance gains of parallelization in GPU computing can be fully exploited in the situation where the degree distribution of network is even and the maximum degree of nodes in the network is not extremely large. The testing biological network `ppi-walks` conforms to the above requirements; see its degree distribution in Fig. 8(a). The benchmark result of this network indicates that: (1) the speedups for DC and H-Index are 68∼72; (2) the speedups for CB and CC are 92∼94, which verifies the above explanation.

Table 7
Two synthetic networks used for the verification experiments.

Synthetic network	Number of nodes	Number of edges
A	2500 000	25 000 000
B	5000 000	50 000 000

For further verification, we additionally use two benchmarks of synthetic networks consisting with 25 million and 50 million edges by employing the strategy of vertex selecting-and-pairing [34]; see Table 7. The degree distributions of networks and benchmark results are presented in Fig. 9 and Tables 8–11.

From the benchmark results, it can be seen that the proposed parallel algorithm developed on the many-core GPU generally performs better than the parallel algorithm developed on the multi-core CPU for CB and CC. More specifically, when the degree distribution is relatively uniform and the maximum degree of the network is not too large (i.e., the workload is balance), the massively parallel computing capability of the GPU can be better exploited.

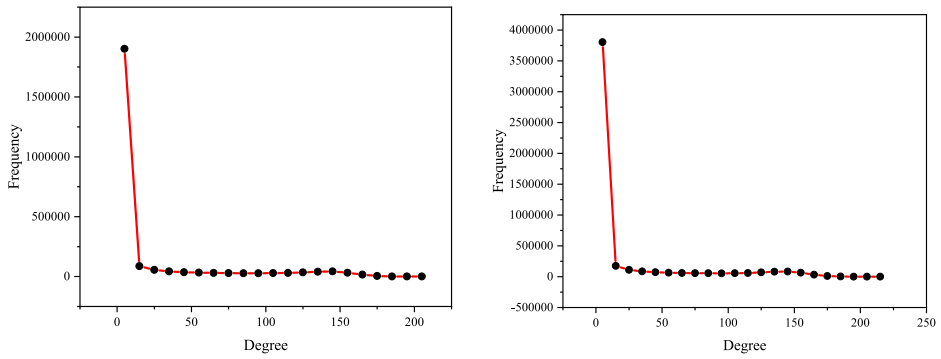


Fig. 9. Degree distributions of synthetic networks. (a) Network A consisting with 25 million edges. (b) Network B consisting with 50 million edges.

Table 8
Computational efficiency of ranking influential nodes according to the metric DC.

Synthetic network	Serial time (ms)	Parallel time (ms)			Speedup		
		CPU	GPU-AoS	GPU-SoA	CPU	GPU-AoS	GPU-SoA
A	43 252	3040	515	438	14.22	83.98	98.75
B	95 799	5661	1050	963	16.92	91.24	99.48

Table 9
Computational efficiency of ranking influential nodes according to the metric H-Index.

Synthetic network	Serial time (ms)	Parallel time (ms)			Speedup		
		CPU	GPU-AoS	GPU-SoA	CPU	GPU-AoS	GPU-SoA
A	49 381	3834	661	598	12.88	74.71	82.58
B	103 149	7596	1302	1160	13.58	79.22	88.92

Table 10
Computational efficiency of ranking influential nodes according to the metric CB.

Synthetic network	Serial time (ms)	Parallel time (ms)			Speedup		
		CPU	GPU-AoS	GPU-SoA	CPU	GPU-AoS	GPU-SoA
A	75 232	6204	2065	1676	12.13	36.43	44.89
B	152 589	11 838	4221	3237	12.89	36.15	47.14

Table 11
Computational efficiency of ranking influential nodes according to the metric CC.

Synthetic network	Serial time (ms)	Parallel time (ms)			Speedup		
		CPU	GPU-AoS	GPU-SoA	CPU	GPU-AoS	GPU-SoA
A	77 429	6074	2072	1628	12.75	37.37	47.56
B	158 305	11 346	4061	3218	13.95	38.98	49.19

5.2. Advantages of the proposed parallel algorithm

In this paper, we designed and implemented parallel algorithms for detecting influential nodes in biological networks by exploiting the massively parallel computing capability of the GPU. There are several strengths of the proposed parallel algorithms, among which the most obvious advantages are efficiency and simplicity.

5.2.1. Competitive efficiency of the proposed parallel algorithm

The proposed parallel algorithms are computationally efficient. The experimental results indicate that the proposed parallel algorithms can achieve competitive computational efficiency. The most critical cause for achieving high computational efficiency is that several computationally expensive procedures in sequential are redesigned and transformed into quite efficient primitives such as parallel sort, parallel scan, and parallel reduction. The use of these primitives can significantly improve the computational efficiency of the proposed algorithms.

There are two main stages in the proposed parallel algorithm for detecting influential nodes in large biological networks including (1) finding the neighbors of all nodes and (2) the calculation of influential metrics.

The proposed strategy of finding real indices of nodes is quite efficient when compared with related work. Polak [35] used multiple threads to examine the adjacency edges whether their first nodes are different. More specifically, primitive `thrust::maximum` is used to calculate numbers of vertices. However, there may be unexpected situations when checking nodes with an empty adjacency list that often occurs in an uncleaned network in which some nodes are missing. In our implementation, with an auxiliary key array, all real indices are obtained according to their location by a well-parallelized `unique` primitive, and the concerns above can be effectively and efficiently avoided.

It is quite simple to find degree (i.e., the number of one-ring neighbors of a node) in network or similar data structures with nodes connected by an edge. The essential idea is that each node of an edge is the neighbor of another [40,41]. However,

race condition appears in a parallel situation when neighbors are finding by different threads simultaneously that their indices may be written in the same position. Compared with the most popular solution coloring, our parallel solution avoids the preprocessing of edge i.e., the coloring, and the implementation is also made straightforward by using efficiency primitives. Moreover, the multi-core CPU implementation of this procedure can achieve speedups of approximately 15, while the GPU implementation yields higher speedups of approximately 48∼94.

The calculation of CB and CC consists of two major steps: (1) obtaining the target value and (2) accumulating the value in a particular way onto corresponding node. The first step is well employed with merge sorting, which significantly reduces the running time compared with brute force method. Unnecessary reads are also avoided following Polak's optimization, that is, the indices are read outside the loop [35]. The accumulation are also performed in parallel using the segmented reduction. With the help of the well-optimized library *thrust*, it runs approximately 20% faster than using a raw CUDA kernel.

For the metric H-Index, parallel calculation can be cast with a specific CUDA kernel in the situation where there are no data dependencies. A binary search is the key procedure for calculating the H-Index, and sorted arrays are needed as input. Compared with the sequential implementation, the degrees of node neighbors are stored as an adjacency array that can be parallel sorted outside the CUDA kernel, which significantly improves the efficiency.

5.2.2. Satisfied simplicity of the proposed parallel algorithm

Moreover, the proposed parallel algorithms are simple and easy to implement. As analyzed above, several computationally expensive procedures in sequence are redesigned and transformed into efficient primitives. These parallel primitives have been designed and optimized, and most importantly, they have been well integrated into the CUDA package. Therefore, it is quite convenient and easy for us to implement the proposed algorithms.

For any parallel algorithm, its correctness must first be verified. Second, the more efficient the algorithm is, the better it is. Third, it will be better that the algorithm is simple to implement and easy to use. Complicated algorithms are not welcome in practice. Fortunately, the proposed parallel algorithm is simple and easy to use.

5.3. Shortcomings of the proposed parallel algorithm

In the proposed parallel algorithm, after finding the neighbors of all nodes, the triangles in the biological network will be identified to calculate the CC and CB. When identifying the triangles, if the node of an edge has a very large degree, then it requires a long computational time to compare and find the common neighbors to form triangles. That is, it is computationally expensive to count triangles when the node of an edge has a large number of nodes.

In GPU computing, it is the best that all threads have almost the same workload due to the thread synchronization. Then, the performance gains of parallelization can be fully exploited. This is referred to as workload balance. However, when counting triangles in the biological network, if there are several nodes with very large degrees, i.e., the hubs, then the workload between GPU threads will become unbalanced. The unbalanced workload will significantly decrease the computational efficiency. This is because a warp of GPU threads must be synchronized to wait for the warp of GPU threads to finish the same work. A small warp of GPU threads typically finishes their work earlier than when invoking a large warp of GPU threads. For the above reasons, in the proposed algorithm, the warp size of GPU threads cannot be large. However, a small warp cannot fully exploit the massively parallel computing capability of GPU. This is the main shortcoming of the proposed algorithm.

5.4. Outlook and future work

In this paper, we have designed and implemented efficient parallel algorithms for detecting influential nodes in biological networks. Experimental results have demonstrated that the proposed parallel algorithms are capable of efficiently detecting influential nodes in very large biological networks. However, there are several issues that need to be addressed and considered in the future.

The first issue is that when there are several nodes with very large degrees in the biological networks, the computational efficiency of the proposed GPU-based parallel algorithms will significantly decrease due to the unbalanced workload between GPU threads. In the future, we will specifically design parallel solutions to address the above problem.

The second issue is that in the proposed parallel algorithms for detecting influential nodes in biological networks, those metrics that are used to measure the influences of nodes are local metrics, including the degree centrality, CB, CC, and H-Index. Those local metrics are commonly used and properly evaluate the influences of nodes in many cases; however, global metrics such as betweenness centrality, closeness, and k-shell are better in some cases. Moreover, detecting influential nodes using global metrics is usually much more computationally expensive than using local metrics. Therefore, we plan to design efficient parallel algorithms for detecting influential nodes based on those global metrics.

In addition, with significant advances in communication technologies, an era of "Internet of Things" (IoT) appears [42,43]. A large amount of IoT data can be collected in various ways [44] which may be used to generate complex networks. The detection of the influential nodes in those generated complex networks can help to process and analyze the IoT data [45,46].

6. Conclusion

In this paper, we have designed and implemented an efficient parallel algorithm for detecting influential nodes for large biological networks by exploiting the GPU. Four local metrics, including the DC, CB, CC, and H-Index, have been used to measure the nodal influence. The computationally expensive procedures in detecting influential nodes have been well redesigned and transformed into quite efficient GPU-accelerated primitives such as parallel sort, parallel scan, and parallel reduction, which significantly improves the computational efficiency. Five large real biological networks have been employed in the experiments to evaluate the performance of the proposed parallel algorithm. It has been observed that (1) the proposed parallel algorithm can achieve speedups of approximately 48∼94 over the corresponding serial algorithm; (2) compared to the baseline parallel algorithm developed on the multi-core CPU, the proposed parallel algorithm yields speedups of 5∼9 for DC and H-Index, while it is slightly slower for CB and CC due to the uneven degree distributions; and (3) when using DC and H-Index, the proposed parallel algorithm is capable of detecting the influential nodes in a large biological network consisting of 150 million edges in less than 3 s.

Future work will focus on addressing the problem of the unbalanced workload in the GPU implementation; and the detection of global metrics instead of the local metrics is also planned to analyze large biological networks in practical case studies.

Data accessibility

All code has been submitted with the manuscript. The biological networks for testing can be downloaded at: https://snap.stanford.edu/.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was jointly supported by the National Natural Science Foundation of China (Grant No. 11602235), and the Fundamental Research Funds for China Central Universities (Grant Nos. 2652018091, 2652018107, and 2652018109). The authors would like to thank the editor and the reviewers for their contributions.

List of Abbreviations

AoS	Array of Structures
CB	Companion Behavior
CC	Clustering Coefficient
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DC	Degree Centrality
GPU	Graphics Processing Unit
IoT	Internet of Things
JC	Jaccard Coefficient
SoA	Structure of Arrays
TBB	Threading Building Blocks

References

[1] M.E.J. Newman, The structure and function of complex networks, SIAM Rev. 45 (2) (2003) 167–256, http://dx.doi.org/10.1137/s003614450342480.

[2] S. Aral, D. Walker, Identifying influential and susceptible members of social networks, Science 337 (6092) (2012) 337–341, http://dx.doi.org/10.1126/science.1215842.

[3] S. Wang, S. Cuomo, G. Mei, W. Cheng, N. Xu, Efficient method for identifying influential vertices in dynamic networks using the strategy of local detection and updating, Future Gener. Comput. Syst. 91 (2019) 10–24, http://dx.doi.org/10.1016/j.future.2018.08.047.

[4] H. Liao, M.S. Mariani, M.s. Medo, Y.-C. Zhang, M.-Y. Zhou, Ranking in evolving complex networks, Phys. Rep. 689 (2017) 1–54, http://dx.doi.org/10.1016/j.physrep.2017.05.001.

[5] L. Lu, T. Zhou, Q.M. Zhang, H.E. Stanley, The H-index of a network node and its relation to degree and coreness, Nature Commun. 7 (2016) 10168, http://dx.doi.org/10.1038/ncomms10168.

[6] S. Wandelt, X. Sun, M. Zanin, S. Havlin, QRE: Quick Robustness Estimation for large complex networks, Future Gener. Comput. Syst. 83 (2018) 413–424, http://dx.doi.org/10.1016/j.future.2017.02.018.

[7] F. Morone, G. Del Ferraro, H.A. Makse, The k-core as a predictor of structural collapse in mutualistic ecosystems, Nat. Phys. 15 (1) (2018) 95–102, http://dx.doi.org/10.1038/s41567-018-0304-8.

[8] M. Kitsak, L.K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H.E. Stanley, H.A. Makse, Identification of influential spreaders in complex networks, Nat. Phys. 6 (11) (2010) 888–893, http://dx.doi.org/10.1038/nphys1746.

[9] L. Zhai, X. Yan, G. Zhang, The bi-directional h-index and B-core decomposition in directed networks, Physica A 531 (2019) http://dx.doi.org/10.1016/j.physa.2019.121715.

[10] A. Sapountzi, K.E. Psannis, Social networking data analysis tools & challenges, Future Gener. Comput. Syst. 86 (2018) 893–913, http://dx.doi.org/10.1016/j.future.2016.10.019.

[11] N. Sumith, B. Annappa, S. Bhattacharya, Influence maximization in large social networks: Heuristics, models and parameters, Future Gener. Comput. Syst. Int. J. Esci. 89 (2018) 777–790, http://dx.doi.org/10.1016/j.future.2018.07.015.

[12] F. Wang, W.J. Jiang, X.L. Li, G.J. Wang, Maximizing positive influence spread in online social networks via fluid dynamics, Future Gener. Comput. Syst. Int. J. Esci. 86 (2018) 1491–1502, http://dx.doi.org/10.1016/j.future.2017.05.050.

[13] J. Zhou, Y. Zhang, J. Cheng, Preference-based mining of top- influential nodes in social networks, Future Gener. Comput. Syst. 31 (2014) 40–47, http://dx.doi.org/10.1016/j.future.2012.06.011.

[14] T. Zhu, B. Wang, B. Wu, C. Zhu, Maximizing the spread of influence ranking in social networks, Inform. Sci. 278 (2014) 535–544, http://dx.doi.org/10.1016/j.ins.2014.03.070.

[15] K. Rahimkhani, A. Aleahmad, M. Rahgozar, A. Moeini, A fast algorithm for finding most influential people based on the linear threshold model, Expert Syst. Appl. 42 (3) (2015) 1353–1361, http://dx.doi.org/10.1016/j.eswa.2014.09.037.

[16] Z.-H. Deng, B.-Y. Lai, Z.-H. Wang, G.-D. Fang, PAV: A novel model for ranking heterogeneous objects in bibliographic information networks, Expert Syst. Appl. 39 (10) (2012) 9788–9796, http://dx.doi.org/10.1016/j.eswa.2012.02.175.

[17] J.C. Jiang, J.Y. Yu, J.S. Lei, Finding influential agent groups in complex multiagent software systems based on citation network analyses, Adv. Eng. Softw. 79 (2015) 57–69, http://dx.doi.org/10.1016/j.advengsoft.2014.09.002.

[18] A. Zareie, A. Sheikhahmadi, M. Jalili, Influential node ranking in social networks based on neighborhood diversity, Future Gener. Comput. Syst. 94 (2019) 120–129, http://dx.doi.org/10.1016/j.future.2018.11.023.

[19] W. Zekun, W. Xiangxi, W. Minggong, Identification of key nodes in aircraft state network based on complex network theory, IEEE Access 7 (2019) 60957–60967, http://dx.doi.org/10.1109/access.2019.2915508.

[20] W. Wang, S. Yang, H.E. Stanley, J. Gao, Local floods induce large-scale abrupt failures of road networks, Nature Commun. 10 (1) (2019) 2114, http://dx.doi.org/10.1038/s41467-019-10063-w.

[21] H. Migallón, V. Migallón, J. Penadés, Parallel two-stage algorithms for solving the PageRank problem, Adv. Eng. Softw. 125 (2018) 188–199, http://dx.doi.org/10.1016/j.advengsoft.2018.03.002.

[22] H. Migallón, V. Migallón, J.A. Palomino, J. Penadés, A heuristic relaxed extrapolated algorithm for accelerating PageRank, Adv. Eng. Softw. 120 (2018) 88–95, http://dx.doi.org/10.1016/j.advengsoft.2016.01.024.

[23] A. Sapountzi, K.E. Psannis, Social networking data analysis tools & challenges, Future Gener. Comput. Syst. 86 (2018) 893–913, http://dx.doi.org/10.1016/j.future.2016.10.019.

[24] S. Wang, Y. Du, Y. Deng, A new measure of identifying influential nodes: Efficiency centrality, Commun. Nonlinear Sci. Numer. Simul. 47 (2017) 151–163, http://dx.doi.org/10.1016/j.cnsns.2016.11.008.

[25] Y.F. Wang, W.Y. Dong, X.S. Dong, A novel ITO Algorithm for influence maximization in the large-scale social networks, Future Gener. Comput. Syst. Int. J. Esci. 88 (2018) 755–763, http://dx.doi.org/10.1016/j.future.2018.04.026.

[26] C.W. Tsai, S.J. Liu, SEIM: Search economics for influence maximization in online social networks, Future Gener. Comput. Syst. Int. J. Esci. 93 (2019) 1055–1064, http://dx.doi.org/10.1016/j.future.2018.08.033.

[27] S. Mallik, U. Maulik, MiRNA-TF-gene network analysis through ranking of biomolecules for multi-informative uterine leiomyoma dataset, J. Biomed. Inform. 57 (2015) 308–319, http://dx.doi.org/10.1016/j.jbi.2015.08.014.

[28] G. Del Ferraro, A. Moreno, B. Min, F. Morone, U. Perez-Ramirez, L. Perez-Cervera, L.C. Parra, A. Holodny, S. Canals, H.A. Makse, Finding influential nodes for integration in brain networks using optimal percolation theory, Nature Commun. 9 (1) (2018) 2274, http://dx.doi.org/10.1038/s41467-018-04718-3.

[29] P.G. Sun, Y.N. Quan, Q.G. Miao, J. Chi, Identifying influential genes in protein–protein interaction networks, Inform. Sci. 454–455 (2018) 229–241, http://dx.doi.org/10.1016/j.ins.2018.04.078.

[30] F. Morone, K. Roth, B. Min, H.E. Stanley, H.A. Makse, Model of brain activation predicts the neural collective influence map of the brain, Proc. Natl. Acad. Sci. 114 (15) (2017) 3849–3854, http://dx.doi.org/10.1073/pnas.1620808114.

[31] J.E. Hirsch, An index to quantify an individual's scientific research output, Proc. Natl. Acad. Sci. 102 (46) (2005) 16569–16572, http://dx.doi.org/10.1073/pnas.0507655102.

[32] G. Mei, S. Cuomo, H. Tian, N. Xu, L. Peng, MeshCleaner: A generic and straightforward algorithm for cleaning finite element meshes, Int. J. Parallel Program. 46 (3) (2017) 565–583, http://dx.doi.org/10.1007/s10766-017-0507-0.

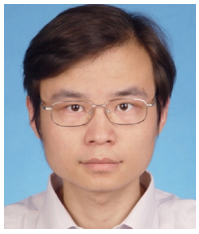
[33] N. Bell, J. Hoberock, C. Rodrigues, THRUST: a productivity-oriented library for CUDA, 2017, pp. 475–491, http://dx.doi.org/10.1016/B978-0-12-811986-0.00033-9.

[34] S. Wang, G. Mei, S. Cuomo, A simple and generic paradigm for creating complex networks using the strategy of vertex selecting-and-pairing, Future Gener. Comput. Syst. 100 (2019) 994–1004, http://dx.doi.org/10.1016/j.future.2019.05.071.

[35] A. Polak, Counting triangles in large graphs on GPU, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, 2016, pp. 740–746, http://dx.doi.org/10.1109/IPDPSW.2016.108.
[36] G. Mei, J. Zhang, N. Xu, K. Zhao, A sample implementation for parallelizing Divide-and-Conquer algorithms on the GPU, Heliyon 4 (1) (2018) e00512, http://dx.doi.org/10.1016/j.heliyon.2018.e00512.
[37] G. Mei, N. Xu, H. Tian, A parallel solution to finding nodal neighbors in generic meshes, 2016, CoRR, abs/1604.04689, arXiv:1604.04689.
[38] S.M. Marinka Zitnik, R. Sosič, J. Leskovec, BioSNAP datasets: Stanford biomedical network dataset collection, 2018, http://snap.stanford.edu/biodata.
[39] OpenMP ARB: The openMP API specification for parallel programming (2019), 2010, http://www.openmp.org/. (Accessed 4 April 2010).
[40] J. Chen, X. Jin, Z. Deng, GPU-based polygonization and optimization for implicit surfaces, Vis. Comput. 31 (2) (2015) 119–130, http://dx.doi.org/10.1007/s00371-014-0924-7.
[41] G. Mei, J.C. Tipper, N. Xu, A generic paradigm for accelerating Laplacian-based mesh smoothing on the GPU, Arab. J. Sci. Eng. 39 (11) (2014) 7907–7921, http://dx.doi.org/10.1007/s13369-014-1406-y.
[42] A.P. Plageras, K.E. Psannis, C. Stergiou, H. Wang, B. Gupta, Efficient IoT-based sensor BIG Data collection–processing and analysis in smart buildings, Future Gener. Comput. Syst. 82 (2018) 349–357, http://dx.doi.org/10.1016/j.future.2017.09.082.
[43] G. Mei, N. Xu, J. Qin, B. Wang, P. Qi, A survey of internet of things (IoT) for geo-hazards prevention: applications, technologies, and challenges, IEEE Internet Things J. (2019) 1–16, http://dx.doi.org/10.1109/JIOT.2019.2952593.
[44] K.E. Psannis, C. Stergiou, B.B. Gupta, Advanced media-based smart big data on intelligent cloud systems, IEEE Trans. Sustain. Comput. 4 (1) (2019) 77–87, http://dx.doi.org/10.1109/TSUSC.2018.2817043.
[45] V.A. Memos, K.E. Psannis, Y. Ishibashi, B.-G. Kim, B. Gupta, An efficient algorithm for media-based surveillance system (EAMSuS) in IoT smart city framework, Future Gener. Comput. Syst. 83 (2018) 619–628, http://dx.doi.org/10.1016/j.future.2017.04.039.
[46] C. Stergiou, K. Psannis, A. Plageras, Y. Ishibashi, B.-G. Kim, Algorithms for efficient digital media transmission over IoT and cloud networking, J. Multimedia Inf. Syst. 5 (1) (2018) 27–34, http://dx.doi.org/10.9717/JMIS.2018.5.1.27.

Lei Xiao is currently a Ph.D. candidate at China University of Geosciences (Beijing). His research interests are in the areas of Numerical Simulation and Computational Modeling, including Computational Geometry, FEM Analysis, and GPU Computing.

Shuangyan Wang is expected to receive her Ph.D. degree in 2020 from China University of Geosciences (Beijing). Her main research field interests are in the areas of Numerical Simulations and Computational Modeling, Graph Theory, Complex Science and Applications, Crisis Management



Gang Mei s an Associate Professor in Numerical Modeling and Simulation at China University of Geosciences (Beijing). He received his Ph.D. degree in 2014 from the University of Freiburg in Germany. He obtained both his bachelor and master degrees from China University of Geosciences (Beijing). His main research interests are in the areas of Numerical Simulation and Computational Modeling, including Computational Geometry, FEM Analysis, GPU Computing, Data Mining, and Complex Science and Applications. He has published more than 50 research articles in journals and academic conferences. He is the IEEE Member, and has served as an Associate Editor for the journal IEEE Access since 2018.