A MapReduce-Based Parallel Clustering Algorithm for Large Protein-Protein Interaction Networks

Li Liu[1], Dangping Fan[1,⋆], Ming Liu[2], Guandong Xu[3], Shiping Chen[2,4], Yuan Zhou[1], Xiwei Chen[1], Qianru Wang[1], and Yufeng Wei[5]

[1] School of Information Science and Engineering, Lanzhou University, Gansu 730000, P.R.China
{liliu,fandp10}@lzu.edu.cn
[2] School of Electrical and Information Engineering, The University of Sydney, NSW 2006, Australia
{ming.liu,shiping.chen}@sydney.edu.au
[3] Advanced Analytics Institute,University of Technology Sydney, NSW 2008, Australia
Guandong.Xu@uts.edu.au
[4] CSIRO ICT Centre, Australia
[5] The Third Peoples Hospital of Lanzhou, Gansu 730050, P.R. China

Abstract. Clustering proteins or identifying functionally related proteins in Protein-Protein Interaction (PPI) networks is one of the most computation-intensive problems in the proteomic community. Most researches focused on improving the accuracy of the clustering algorithms. However, the high computation cost of these clustering algorithms, such as Girvan and Newmans clustering algorithm, has been an obstacle to their use on large-scale PPI networks. In this paper, we propose an algorithm, called Clustering-MR, to address the problem. Our solution can effectively parallelize the Girvan and Newmans clustering algorithms based on edge-betweeness using MapReduce. We evaluated the performance of our Clustering-MR algorithm in a cloud environment with different sizes of testing datasets and different numbers of worker nodes. The experimental results show that our Clustering-MR algorithm can achieve high performance for large-scale PPI networks with more than 1000 proteins or 5000 interactions.

Keywords: PPI, Clustering, MapReduce, Edge-betweenness.

1 Introduction

Detection of physically and functionally related proteins is one of the most challeng-ing tasks for the proteomics community. Intensive computation is needed to analyze pairwise protein interaction in order to understand how proteins work together to perform their tasks. The PPI network is an important information

⋆ Corresponding author.

S. Zhou, S. Zhang, and G. Karypis (Eds.): ADMA 2012, LNAI 7713, pp. 138–148, 2012.

source, which en-codes the interaction between proteins. The network contains nodes (i.e. proteins) and edges, which represent the interactions between the proteins. When proteins are expe-rienced in the same cellular process or have the same protein complex, they are ex-pected to have strong interactions with their partners [1], and vice versa. Furthermore, modularity is mostly studied by grouping physically or functionally related proteins in a cluster such that the proteins in the same cluster share common biological features [2]. Therefore, the computation objective is to discover the clustering structures in the PPI network, i.e. to determinate a collection of sets of nodes where each node is closer to the other nodes within the same set than to nodes outside of the set.

Several graph-based clustering algorithms have been applied to PPI network to find highly connected sub-graphs. The flow-based cluster algorithm [3] was used to cluster large-size networks and the time complexity of this algorithm was approximately $O(k\,|\varepsilon|+\sum_{i=1}^{|v_c|} d_i^2)$, where the d_i is the degree of the node i in the graph, and k is a small constant which is typically set to 10. A quasi all paths-based network analysis algorithm, called CASCADE [4], was proposed to effectively detect biologically relevant clusters with the time complexity of $O(n^3 \log n + n^2 m)$, where n is the number of nodes and m is the number of edges. Girvan and Newmans Edge-betweeness algorithm [5] is one of the most popular clustering algorithms, which has been widely used to discover clustering structures in different domain networks, such as web, social networks, and PPI networks. The time complexity of this algorithm is $O(nm^2)$.These studies mainly focused on improving the clustering accuracy of Girvan and Newmans Edge-betweeness algorithm.

However, the high computation cost of using these clustering algorithms becomes a major issue when the PPI network has thousands of nodes and millions of edges. In fact, the time complexities of these clustering algorithms for PPI networks are more than quadratic in terms of the number of nodes.

In this paper, we aim to develop a novel parallel implementation of Girvan and Newmans edge-betweenness algorithm using MapReduce to address the high com-putation cost issue. Although the edge-betweenness algorithm provides good accuracy for clustering small or medium size PPI networks, its high computation cost has become an obstacle to applying it for clustering large size PPI networks [6]. A parallel version of betweenness algorithm using the MapReduce distributed programming framework is proposed to handle large-scale PPI networks.

The rest of this paper is organized as follows: Section 2 describes background on notations and technologies used in this paper. In Section 3 we propose our MapReduce-based parallel algorithm. The experimental results are presented and discussed in Section 4. We conclude in Section 5.

2 Background and Related Work

2.1 Girvan and Newmans Edge-Betweenness Algorithm

Edge-betweenness measures the centrality of an edge within a graph. The betweenness is formulated as follows: Given a graph $G(V, E)$, where V is the set of n nodes and E is the set of m edges. Let $\sigma_{s,t}$ denote the total number of shortest

paths from node s to node t and $\sigma_{s,t}(e)$ is the number of those paths that pass through e where $e \in E$. $\delta_{s,t}(e)$ represents the ratio of the total number of shortest paths between s and t that pass through e to the $\sigma_{s,t}(e)$, where $\delta_{s,t}(e) = \frac{\sigma_{s,t}(e)}{\sigma_{s,t}}$. Thus, the betweenness of an edge e is defined as $BC(e) = \sum_{s \neq t \in V} \delta_{s,t}(e)$.

An edge with high betweenness indicates that a large number of the shortest paths between two nodes pass through it. If the edge is removed, the graph is split into two subgraphs. Similarly, by removing the edges with the highest betweenness in descending order, we can separate the PPI graph into several subgraphs. Such subgraph contains interconnected proteins that have strongly functional relationships.

Girvan and Newmans edge-betweenness algorithm first calculates the betweenness for all edges in a graph by using breadth-first searching (BFS) method. Then it removes the edge with the highest betweenness. The algorithm repeats the calculation and removal steps until no edges remain. The total time complexity of Girvan and New-mans edge-betweenness algorithm is $O(nm^2)$, where n is the number of nodes and m is the number of edges.

2.2 PPI Modularity Evaluation

There are m collections of subgraphs produced by Girvan and Newmans edge-betweenness algorithm. In order to determine which one is the best collection in terms of functionally related proteins in a cluster, the modularity evaluation for each collection is defined as $Q = \sum_{i=1}^{k} (f_{ii} - (\sum_{j=1}^{k} f_{ij})^2)$, where k is the number of subgraphs in current collection, f_{ii} represents the fraction of all edges in the network that connect nodes in the same subgraph while f_{ij} represents the fraction of all edges in the network that connect the nodes in subgraph i to the nodes in subgraph j.

If Q approaches 1, it indicates that the collection has strong clustering structure. The higher the value of Q is, the stronger the clustering structure is. Therefore, the collection with the highest value of Q is selected as the final clustering structure in PPI network. The time complexity of calculating all the values of Q is $O(nmk^2)$.

2.3 Parallel Implementations for Computing Betweenness

Since Girvan and Newmans edge-betweenness algorithm suffers from high computa-tional cost, it is impractical for processing large-scale PPI networks. Few researches studied the parallel implementation of betweenness algorithm to handle graphs with more than hundred thousand edges. These parallel algorithms were mainly designed for the shared-memory architecture. For example, Bader and Madduri [7] implemented the first parallel algorithm for computing the betweenness based on SNA software package on the IBM p5 570 with 16-processors and 20GB-memory, and the MTA-2 with 40-processor and 160GB-memory. The algorithms time complexity and space complexity is $O(\frac{nm+n^2 \log n}{p})$

and $O(n+m)$ respectively, where p is the number of processors. Madduri et al. [8] proposed a parallel implementation based on the massively multithreaded Cray XMT system with the Threadstorm processor and 16GB memory. It also takes $O(\frac{nm+n^2 \log n}{p})$ time complexity and $O(n+m)$ space complexity to calculate the betweenness. G. Tan et al. [9] extended their work and improved the algorithm by reducing the time complexity to $O(\frac{nm}{p})$. However, usually these massive computer servers are very expensive.

2.4 MapReduce and Hadoop

MapReduce is a distributed programming model for parallel processing large-scale datasets and the computing. Compared with shared-memory architecture, MapReduce has lower economical cost and better scalability in terms of problem sizes and re-sources.

A MapReduce-based algorithm is different from the parallel algorithms based on shared-memory architecture, because MapReduce is based on message-passing architecture. The MapReduce model mainly contains two data processing phases: Map and Reduce. During the Map phase, the input data is split into smaller data segments and distributed to multiple processing units, such as virtual machines in Cloud, for parallel processing by using a map function. The intermediate output produced by the map function is a collection of key-value pair tuples. During the Reduce phase, the intermediate outputs from each map function are transferred to the machines that execute a reduce function. The key-value formed data are sorted by using the keys and aggregated by using the reduce function.

Hadoop is an open source version of Googles MapReduce and Google File System. Hadoop includes a master node and multiple worker nodes. The master node consists of a JobTracker and a NameNode. The JobTracker manages jobs scheduling and assigns the jobs to the TaskTrackers on the worker nodes, which is responsible for executing the map function and the reduce function. The NameNode is responsible for storage and management of file metadata and file distribution across several DataNodes on worker nodes, which store the data contents of these files.

3 MapReduce-Based Clustering Algorithm Implementation

In this section, we propose our MapReduce-based parallel algorithm, called *Clustering-MR*. As being seen in Fig.1, the *Clustering-MR* contains the following four steps.

Step1: An algorithm, called *forward-MR*, executes n tasks in parallel by using MapReduce. Each task has been given a node v, which is set as a root node. The object of this algorithm is to find the shortest paths by using BFS and calculate their distances from other nodes to the root node.

Step2: An algorithm, called *backward-MR*, executes m tasks in parallel by using MapReduce. Each task has been give an edge e. The object of this algorithm is to calculate the edge-betweenness $BC(e)$.

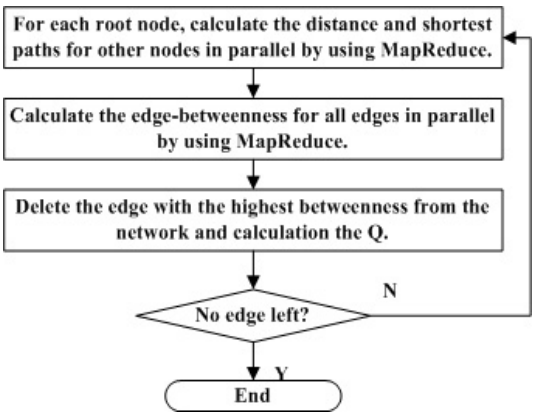

Fig. 1. The workflow of the *Clustering-MR* algorithm

Step3: The algorithm removes the edge with the highest betweenness and calculates Q value.

Step4: The algorithm repeats previous steps till no edge remains.

3.1 Input File Format for MapReduce

We use an adjacency list to represent a PPI network, which is considered as the MapReduce input file. There are totally $n \times n$ lines in the input file. The format of a line in the input file is defined as:

<*NodeId*> <*Root*> <*Neighbors*> | <*Distance*> | <*Color*> | <*Path*>

where:

. *NodeId* is the ID of a node;
. *Neighbors* is a list of nodes that are the neighbors of this node;
. *Root* is the ID of the root node;
. *Path* is the shortest path;
. *Distance* is shortest-path distance from the node to the root node;
. *Color* is the status of this node.

There are three states for each node: (1) WHITE: the node is unreached; (2) GRAY: the node is reached and going to be handled; and (3)BLACK: the node is reached and has been handled. Initially, the *Distance* for each node is set as MAX and its *color* is set as WHITE. If *NodeId* equals to *Root*, its *Distance* and *Color* are set to 0 and GRAY respectively.

3.2 Forward-MR

In *forward-MR*, the map function reads a line from the input file. If the *Color* is GRAY, it is changed to BLACK. For each node in *Neighbors*, if the *Color* is not

BLACK, the map function generates a set of *key-value* pairs as its output, where the keys are neighbors *NodeIds* and *Root*, and the values contains *Distance*+1, GRAY and *NodeId*.

Based on the map functions outputs, the reducer selects the minimum distance among the values , updates the status of this node, and records the path, then produces the output consisting of *NodeId*, *Root*, *Neighbors*, *Distance*, *Color* and *Path*, where *Distance* is the minimum distance, *Color* is the latest state and *Path* is the shortest path. If the node is not reached, then the *Distance* remains MAX and the *Color* remains WHITE. The pseudocode of the map function for *forward-MR* is shown in Algorithm 1 and reduce function is shown in Algorithm 2.

Fig.2 shows an example of the process of the *forward-MR* changing the nodes color in a network, where node 4 is the root node. Firstly, for node 2, 3 and 6, which are the *Neighbors* of the node 4, the map function emits the *key-value* pairs as <2 4 4 1 GRAY 4>, <3 4 1,4 1 GRAY 4> and <6 4 4,5,7 1 GRAY 4>. For node 4, the *key-value* pair is <4 4 2,3,6 0 BLACK >. Then the reduce function updates the *Color* of 2,3 and 6 as GRAY, the *Distance* and *Path* as 1 and 4, and the color of 4 as BLACK. Secondly, for node 1, 5 and 7 which are the *Neighbors* of the node 3 and 6, the map function emits the *key-value* pairs as <1 4 3 2 GRAY 4,3>,<5 4 6,7 2 GRAY 4,6> and <5 4 6,7 2 GRAY 4,6>. For node 2, 3 and 6, the *key-value* pairs are <2 4 4 1 BLACK 4>, <3 4 1,4 1 BLACK 4> and <6 4 4,5,7 1 BLACK 4>. Then the reduce function updates the *Color* of 1,5 and 7 as GRAY, the *Distance* and *Path* as 2 and 4,3, 4,6 and 4,6 separately, and the *Color* of node 2,3 and 6 as BLACK. Since the *Color* of node 4 is BLACK, the status of it is not changed. Thirdly, for node 1, 5 and 7, the *Color* of each neighbor *NodeId* is BLACK, the map function just emits the *key-value* pairs as <1 4 3 2 BLACK 4,3>,<5 4 6,7 2 BLACK 4,6> and <5 4 6,7 2 BLACK 4,6>. Then the reduce function updates the *Color* of node 1,5 and 7 as BLACK. At last, as each node has reached, the *forward-MR* finishes.

Algorithm 1. Mapper: Forwardmapper

```
Input: Each line contains NodeID, Root, Distance, Path, etc.
Output:<key, value>pair, where key contains nodeId and root while value contains node's info
1.if (line.Color is GRAY)
2.foreach node in neighbors do
3. if(the color of node is not BLACK)
4. node.distance = line.Distance + 1
5. node.color = GRAY
6. node.path = line.Path.add(line.NodeId)
7. output.write(<node,line.Root>,node.value)
8. endif
9.endfor
10.line.Color = BLACK
11.output.write(<line.NodeId, line.Root>,line.value)
```

3.3 Backward-MR

The final output file of *forward-MR* is used as the input file for *backward-MR*. The map function of *backward-MR* reads a line from the input file, and generates the *key-value* pairs, where the keys are the edges within the *Path*, and every value is set to 1. The reduce function calculates the edge-betweenness for each edge. Only one shortest path between two nodes is recorded in order to make computation tractable [10],thus *Clustering-MR* is an approximate algorithm.

4 Evaluation

In this section, we evaluate the performance of our *Clustering-MR* algorithm with different datasets and different numbers of worker nodes (4, 8, 12, 16 and 20). The theoretical time complexity of *Clustering-MR* algorithm is $\frac{O(nm^2)}{p}$, where n is the number of nodes, m is the number of edges and p is the number of processors. In addition, we compare *Clustering-MR* with a sequential algorithm (*Clustering-SEQ*) which executes on only one processor with single thread.

Algorithm 2. Reducer: ForwardReducer

Input: <key, list(value)>, where key contains nodeId and root while list contains the node's info.
Output:<key, value>pair, where key contains nodeId and root while value contains node's info.

```
1. currentNode = key.nodeId
2. foreach value in list(value) do
3. if(value.Nighbors != null)
4. currentNode.Neighbors = value.Neighbors
5. endif
6. if(value.distance <= currentNode.Distance)
7. currentNode.Distance = value.distance
8. currentNode.Path = value.path
9. endif
10. if(value.color != WHITE)
11. currentNode.Color = value.color
12. endif
13. endfor
14.output.write(key,currentvalue)
```

4.1 Experiment Setup

The *Clustering-MR* algorithm was tested on a Hadoop system, which consists of a single Master Node and 20 identical Worker Nodes. The Master Node contains four 64-bit Intel dual-core 3.3GHz processor, along with 4GB memory, and 500GB disk Each Worker Node contains a single 32 bit Intel 2.8GHz processor, 1GB memory and 164.7GB disk. All the nodes ran on Ubuntu 10.04 and had Hadoop v0.20.205.0 de-ployed. The network bandwidth was 12MB/s.

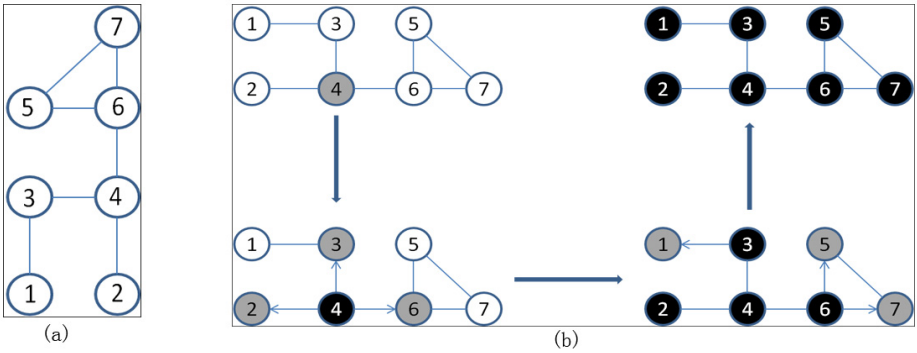

Fig. 2. (a) An example of a toy network. (b) The *forward-MR* algorithm execution process

4.2 Datasets

We evaluated the *Clustering-MR* algorithm on these seven datasets containing PPI graphs, which were downloaded from DIP database [11] on 18th, May, 2012. Table 1 shows the size of each dataset including the original size and the MapReduce input size.

Table 1. Experimental Datasets

Species	**Mark**	**Proteins**	**Interactions**	**Size(KB) (original)**	**Size(KB) (MapReduce Input)**
D.melanogaster	DM	7,439	22,632	227	2,637,954
S.cerevisiae	SC	2,993	7,029	70	365,973
C.elegans	CE	2,629	3,970	38	225,598
E.coli	EC	1,355	5,476	48	92,378
H.sapiens	HS	941	1,160	10	24,389
H.pylo	HP	7,01	1,358	12	15,966
M.musculus	MM	314	267	3	2,356

4.3 Experimental Results

Fig.3(a) shows the overall execution time of the Clustering-MR algorithm for processing each dataset in the Hadoop system using different worker nodes: 4, 8, 12, 16 or 20. For the DM dataset, the execution time of the algorithm with 4 worker nodes was almost 3 times longer than the runtime of the algorithm with 20 worker nodes. Similarly, for SC and CE dataset, the execution time of the algorithm with 4 worker nodes was almost 2 times longer than the runtime of the algorithm with 12 worker nodes. However, the performance didnt improve anymore when the number of worker nodes was larger than 12. For EC, HS, HP, and MM datasets, although the execution time decreased as the number of

machines increases, it didnt have big improvement. These results indicate that our *Clustering-MR* algorithm with Hadoop indeed can improve performance for large size datasets. However, the improvement is not obvious for small data sizes and cannot remain linear as the number of work nodes reaches a certain threshold. This is because the system would spend more time on the management and data transferring among these machines.

Fig.3(b) shows the relationship between the number of nodes (proteins) in PPI graphs and the runtime of the *Clustering-MR* algorithm in a Hadoop system using the different number of work nodes: 4, 8, 12, 16 and 20. Similarly, Fig.3(c) shows the relationship between the number of edges (interactions between proteins) in PPI graphs and the runtime of the *Clustering-MR* algorithm. In the case of using 4 worker nodes, the execution time of the algorithm increased dramatically when the number of proteins was larger than 3000 or the number of interactions was larger than 6000. But, in the case that more worker nodes were used in the Hadoop system, the runtime slightly increased. For small size of proteins and interactions, the number of machines used in the system did not affect much the execution time of the *Clustering-MR* algorithm. But for larger size of proteins and interactions, the impact of the number of machines became big.

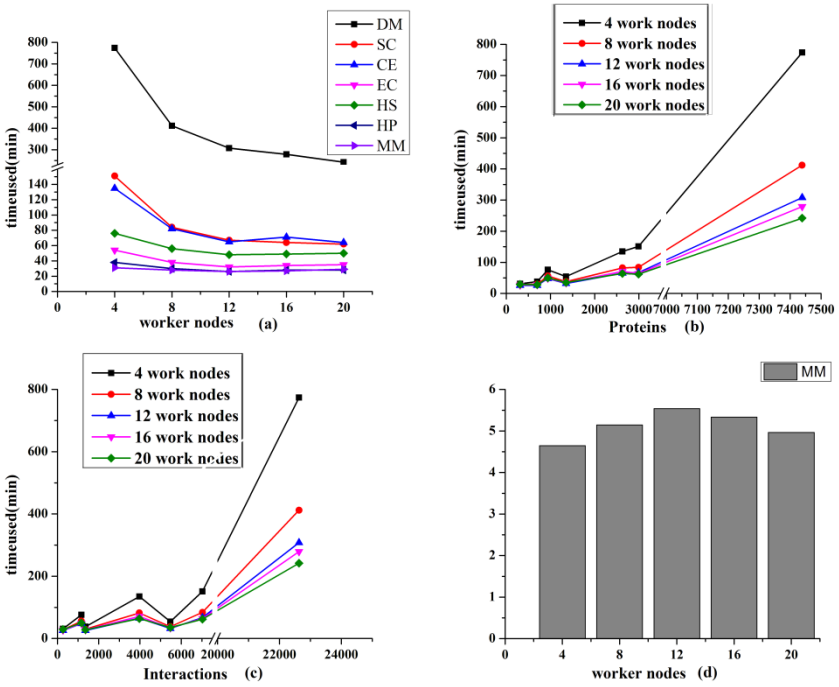

Fig. 3. The experiment results. In (a),(b),(c), the y-axis is the running time of computation, and the x-axis is the number of worker nodes, proteins and interactions respectively. In (d), the y-axis is the speedup (serial/parallel) and the x-axis is the number of worker nodes.

Fig.3(d) shows the ratio of the running time of the *Clustering-SEQ* algorithm to the *Clustering-MR* algorithm in a Hadoop system with using different worker nodes: 4, 8, 12, 16 and 20, when both algorithms are working on MM dataset. The speedup increases as the number of worker nodes increases from 4 to 12. Interestingly, the speedup decreases as the number of worker nodes increases from 12 to 20. This is because the MM dataset contain relative small PPI networks and the *Clustering-SEQ* algorithm can perform reasonably good on the small PPI networks. Furthermore, as the number of worker nodes increases, task scheduling, data transferring, large number of hard disk and memory switching took up a large portion of total running time. There-fore, given a dataset size, determining the optimal number of Hadoop worker nodes can be an interesting research work.

5 Conclusions

In this paper, we proposed a new parallel processing algorithm, called the *Clustering-MR*, applying the MapReduce model to parallelize the Girvan and Newmans edge-betweenness for large PPI networks. This new algorithm overcomes the issues of high computation cost caused by the sequential processing. Compared with the shared-memory based parallel algorithms, our algorithm is cost-effective and easy to scale. We implemented *Clustering-MR* on a real distributed system using Hadoop. Our experimental results show that: (1) *Clustering-MR* gets better performance on runtime than *Clustering-SEQ*. (2) *Clustering-MR* can efficiently handle larger-scale PPI networks with more than 1000 proteins or 5000 interactions, which would be very difficult (if not impossible) for *Clustering-SEQ*.

We observed that the experimental results in practice do not totally agree with the theoretical analysis, especially when the number of nodes or edges becomes lager and lager. The reason is that the space complexity of *Clustering-MR* algorithm that equals to $O(nm^2)$ leads to the increment of data transmission between machines which costs a large amount of time to complete. To reduce the space complexity is one of the key points in our following work. We will improve the *Clustering-MR* algorithm by changing the input data format and optimizing the Reduce step.

In our future work, we will also compare *Clustering-MR* to the MPI(Message Passing Interface)-based clustering algorithm for large PPI networks. Both of them are message-passing based algorithms. Besides, we will evaluate our *Clustering-MR* algorithm in other applications, such as social networks and linked web.

Acknowledgements. This work was partially supported by National Natural Science Foundation of China (grant no. 61003240) and Gansu Provincial Science & Technology Department (grant no. 1007RJYA010). We would also like to thank DIGICOM JAPAN Co.Ltd (http://www.digicomnet.co.jp) to provide us the Hadoop system for our experiments.

References

1. Maslov, S., Sneppen, K.: Specificity and stability in topology of protein networks. Science 296(5569), 910–913 (2002)
2. Baraba'si, A., Oltvai, Z.N.: Network Biology: Understanding the Cell's Functional Organization. Nature Reviews Genetics 5, 101–113 (2004)
3. Satuluri, V., Parthasarathy, S.: Scalable Graph Clustering Using Stochastic Flows: Applications to Community Discovery. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2009, Paris, France, pp. 737–745 (2009)
4. Hwang, W., Cho, Y., Zhang, A., Ramanathan, M.: CASCADE: a novel quasi all paths-based network analysis algorithm for clustering biological interactions. BMC Bioinformatics 9(64) (2008)
5. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. PNAS 99(12), 7821–7826 (2002)
6. Dunn, R., Dudbridge, F., Sanderson, C.M.: The Use of Edge-Betweenness Clustering to Investigate Biological Function in Protein Interaction Networks. BMC Bioinformatics 6(39) (2005)
7. Bader, D.A., Madduri, K.: Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks. In: International Conference on Parallel Processing (ICPP 2006), pp. 539–550 (2006)
8. Madduri, K., Ediger, D., Jiang, K., Bader, D.A., Chavarria-Miranda, D.: A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets. In: IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009), pp. 1–8 (2009)
9. Tan, G., Tu, D., Sun, N.: A Parallel Algorithm for Computing Betweenness Centrality. In: International Conference on Parallel Processing (ICPP 2009), pp. 340–347 (2009)
10. Maier, M., Rattigan, M., Jensen, D.: Indexing network structure with shortest-path tree. ACM Transactions on Knowledge Discovery from Data 5(3) (2011)
11. DIP Database, `http://dip.doe-mbi.ucla.edu/`