Accelerating GN Algorithm on Many-core Processors to Find Community Structure in Complex Networks

YANG, Liwen

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
8
(開始ページ / Start Page)
19
(終了ページ / End Page)
22
(発行年 / Year)
2013-03
(URL)
https://doi.org/10.15002/00009516

Accelerating GN Algorithm on Many-core Processors to Find Community Structure in Complex Networks

Liwen Yang
Graduate School of Computer and Information sciences
Hosei University
E-mail: liwen.yang.7x@stu.hosei.ac.jp

***Abstract*—Complex network has become an important field in science research recently and it is proved that many networks possess strong community structure. In order to exploit and utilize networks, we need to detect community structure. Here we introduce a kind of classical community detecting algorithm, Givern-Newman (GN) algorithm and propose two methods to parallel GN algorithm. These two proposed methods are based on calculating edge betweenness in parallel. The first method, calculating full betweenness of each edge for different source vertices in parallel, was implemented on the many-core processor. GN algorithm is an iterative community detection algorithm based on removing edge repeatedly. Our two methods can reduce the running time of per iteration. As a result, the method using coarse-grained parallelism for 4-core processor is 3 times faster than the sequential program of GN algorithm.**

Keywords*— *GN algorithm; community detecting; edge betweenness; many-core processor

I. INTRODUCTION

The study of community structure in networks has a long history. It is closely related to the ideas of graph partitioning in graph theory and hierarchical clustering in sociology [1, 2]. Many networks possess strong community structure. For example, social networks, biochemical networks, food webs and information networks. It can help us to understand how network function and topology affect each other. Network is composed with many kinds of vertices. There are more connections among the vertices with the same properties, and there are few connections among the vertices with different properties. The sub-graph has the vertices with the similar properties and the edges among the vertices is called cluster (i.e. community). Community detecting algorithm is used to find the potential community in the network. So far, many community detecting algorithms have been around. The most representative community detecting algorithms are Graph Partitioning method [9], W-H algorithm [5], Hierarchical Clustering method [6]. Among them, GN algorithm [3] belongs to a divided Hierarchical Clustering method. It needs to recalculate full betweenness (refer Section II part C) for all remaining edges after removing an edge. This process takes significant time in GN algorithm.

At present, with the improvement of the hardware's performance, many-core processor gradually becomes a mainstream. But previous community algorithms are almost sequential. In addition, with the increased information, we need a faster community detecting algorithm. However, parallel community detecting algorithm can get the faster speed on many-core processors.

In this paper, we propose that speed GN algorithm by parallel processing on many-core processors. That is to parallelize GN algorithm. The concrete methods are in Section III.

The paper is organized as follows. The instruction of conventional GN algorithm is in Section II. Section III presents the proposed methods. Experiments and conclusions are discussed in Section IV and V respectively.

II. CONVENTIONAL GN ALGORITHM

A. GN Algorithm Outline

GN algorithm is the classic algorithm in the area of community detecting. The basic thought of GN algorithm is calculating full betweenness of each edge continuously and progressively removing edge which has the highest edge betweenness. Fig. 1 shows an example of a small network with communities. In Fig. 1, the small black circles (i.e. vertices in network) and the lines (including the black and green ones, i.e. edges in network) represent the individuals with some relationship and their relationship respectively. There are three communities, denoted by the red dashed circles, which have dense internal links (black lines) but between which there are only a lower density of external links (green lines). The green lines will be removed when uses GN algorithm to process this graph. We can get the potential communities in this graph.

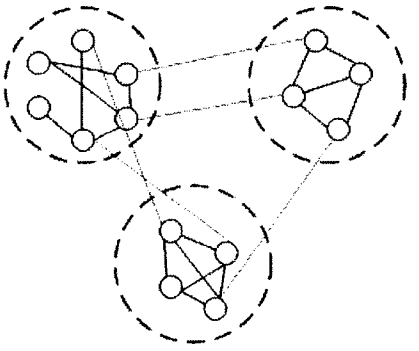

Figure 1. A small network with communities

Supervisor: Prof. Yuji Sato

B. Modularity

In order to know how many communities networks can be divided into, a quality measure of a particular division of a network, modularity is proposed by Newman and Girvan [4]:

$$Q = \sum_{i=1}^{k} (e_{ii} - a_i^2) \quad (1)$$

Where consider a particular division of a network into k communities and define a $k \times k$ symmetric matrix e whose element e_{ij} represents the fraction of all edges in the network that link vertices in community i to vertices in community j. Therefore, e_{ii} represents the fraction of edges in the network that connect vertices in the same community. $a_i = \sum_j e_{ij}$ refers to the fraction of edges that connect to vertices in community i. Generally the bigger this value of Q is, the better the result is. Fig. 2 shows a dendrogram of a network with 12 vertices. This dendrogram represents an entire nested of possible community divisions for the network and it is divided into 4 communities. The vertices in the same color are in the same communities. At the right of the figure we also show the modularity calculated by Eq. (1). We can use the value of the modularity (i.e. Q) to judge where we should cut the dendrogram to get a significant division. Each division (there are different number of communities in each division) corresponds to a value of modularity. The peak of the modularity (red dotted line) corresponds to perfect community division.

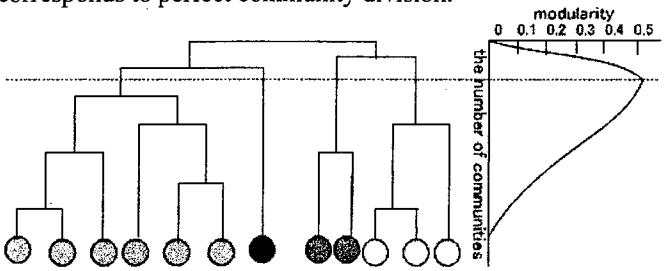

Figure 2. Modularity and dendrogram for a community-structure graph.

C. Edge Betweenness Definition and Calculation

Betweenness of an edge is the number of the shortest paths around this edge from a source vertex s. Do repeatedly the process for all possible source vertices s and we can get full betweenness of an edge through summing the betweenness.

Suppose a graph with m edges and n vertices. Breadth-first search for this graph can find the shortest paths from a source vertex s to all others in time $O(m)$. Calculate the number of the shortest paths along each vertex from source vertex s. Weight w_i represents the number of shortest paths along vertex i from source vertex s. Suppose vertex i connecting with s, and j is farther than i from the source vertex s, and then the fraction of the shortest path from j through i to s is given by w_i/w_j. Thus, to calculate betweenness starting at s, we need do the following steps [4]:

1. Find each "leaf" vertex f, there are no paths from s to other vertices go through f.
2. Suppose vertex i adjoins f, a weight of w_i/w_f is assigned to the edge from f to i.
3. Starting with the edges that are farthest from the source vertex s and work up towards s. To the edge from vertex i to j, j is farther from s than i, assign a weight that is 1 plus the sum of the weights on the neighboring edges (i.e., each of them shares a common vertex with this edge) immediately below it, all multiplied by w_i/w_j.
4. Repeat from step 3 until vertex s is reached.

Now repeat the above process for all n source vertex s, and sum the edges of weights (betweenness) each time, we can get the full betweenness for all edge in time $O(mn)$.

III. PROPOSED PARALLEL METHODS

According to the basic thought of GN algorithm, the process of calculating and removing edges is repeated m times. Here we define m is the number of edges in graph. From aforementioned, the process of calculating betweenness is the main part of GN algorithm. Therefore, our method is decreasing the running time of calculating betweenness. Here we propose two parallel ways: Coarse-grained parallelism and Fine-grained parallelism.

A. Coarse-grained Parallelism

In coarse-grained parallelism, we regard the process of calculating betweenness for all edges from each source vertex as a task. Copy the graph data for each task and it can avoid data dependency and access conflict. All tasks on different cores run in parallel at the same time. Here suppose the number of tasks is equal to the number of cores. Synchronize only when all tasks end, and then start another iteration process. All tasks can execute in parallel during each iteration process. Therefore, speed up the program of GN algorithm successfully. For multi-thread technology, as Fig. 3 shows, the circles in the rectangle represent any vertices in a graph including source vertices. Any vertex in a graph has a chance to be a source vertex. Therefore, the number of vertices is equivalent to the number of source vertices. Allocate each task to each thread. *Ti* represents that thread i performs a task for source vertex i. And this graph is an undirected and unweighted graph. The dotted line with arrow only represents the direction of search from source vertex S_i to target vertex in the shortest path tree.

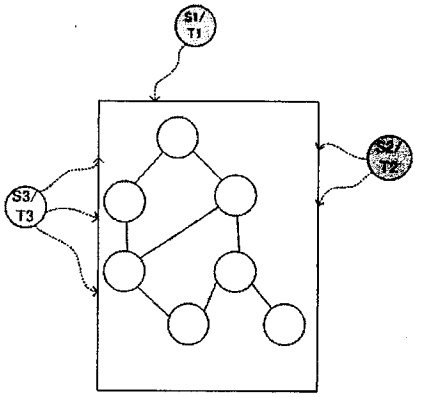

Figure 3. Graph search from different source vertices

$$\delta(e) = \sum_{s \in V} \delta_s(e) \quad (2)$$

In the formula above, s and e represent a source vertex and an edge in a graph. V represents all the vertices. $\delta_s(e)$ refers to betweenness of edge e for a source vertex s. $\delta(e)$ represents the sum of betweenness for different source vertices (i.e. full betweenness). All values of $\delta_s(e)$ can be calculated in parallel.

B. Fine-grained Parallelism

In this part, we introduce our method about Fine-grained parallelism by Fig. 4. In Fig. 4, we see that the search process starting from source vertex s is similar to p-breadth first search. Here p refers to the number of processors. The parallelization is performed in a task and this task includes two procedures. One is calculating the number of the shortest paths from s to vertex i using breadth-first search. The other one is a backtracking course. That is calculating betweenness of edges from leaf vertex to s. We can search all the vertices with the same adjacent vertex in parallel for starting from one source vertex. More specifically, we can operate in parallel for each of them when travels the adjacent vertices of a vertex. Apparently, the degree of parallelism depends on the degree of each vertex. In the sparse graph, the degree of vertices is very small. Therefore, this method can get a better result in a dense graph than a sparse graph.

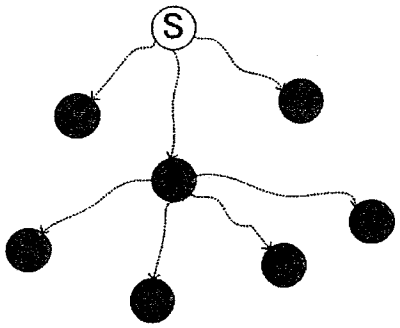

Figure 4. Graph search from one source vertex

IV. EXPERIMENTS

A. Execution Platform and Test Data

We perform all experiments on a PC with the following properties: Intel(R) Core (TM) i7 960 3.20GHz CPU, 12 GB RAM, 1TB Hard disk drive. The test datasets we used are downloaded from a website about network data [7]. They are the undirected and unweighted graphs and from the realistic data. Here we introduce Dolphins network [4, 8] in detail. It was generated by Lusseau et al. for seven years of observation of 62 dolphins living off Doubtful Sound. The network has 62 vertices (representing the dolphins) and 159 edges (representing a significant frequent association. It is divided into two large communities, and the larger of the two is also divided into four smaller sub-communities. The Dolphins network has five communities in total [4].

B. Experimental Methods

There are two experimental methods applied for each network. For comparison with our parallel method, we also implement the sequential program of GN algorithm. And then design and implement the parallel program of GN algorithm on multi-core processor by Java multi-thread technology.

1) Sequential program of GN algorithm

In this part, implement a sequential program in Java of GN algorithm. Problem Analysis Diagram of GN algorithm is shown in Fig. 5. The procedure of GN algorithm [3] with modularity for detecting community is simply stated as follows:

1. Calculate the full betweenness (full betweenness is the accumulation of betweenness) for all edges in the network.
2. Remove the edge with the highest full betweenness. We should compute the value of modularity while gets a new community and record the network structure in this time.
3. Recalculate full betweenness for all remaining edges.
4. Repeat from step 2 until no edges remain. And then choose the division situation corresponding to the maximum value of modularity.

Test these six datasets using the sequential program of GN algorithm and record the experimental results in Tab. 2.

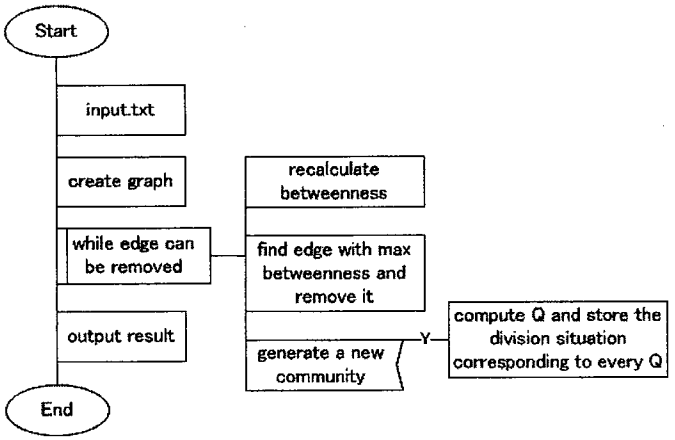

Figure 5. Problem Analysis Diagram of GN algorithm

2) Parallel program of GN algorithm on multi-core processor

In this experiment, we adopt the task-level parallelism reach the goal of accelerating GN algorithm. Since this kind of coarse-grained parallelism needs relatively small amount of communication. It is better to simplify programming and can reduce the running time of program. Specifically, search in a graph by breadth-first search algorithm from a source vertex and calculate the distance from this source vertex to any other vertices. And then calculate the number of the shortest paths along each vertex. According to the preceding results, we can obtain the number of the shortest paths along each edge (i.e. betweenness). We use synchronization mechanism to avoid read-write conflict when calculates full betweenness. (full betweenness is the accumulation of betweenness). In other words, multiple threads can execute simultaneously, and then accumulate each thread's results using synchronization mechanism when all the threads end in each iteration process. Because of assigning a copy of the graph data for each thread, each thread executes on its exclusive data. Therefore, the value of modularity is not changed. At the same time, it ensures the correctness of parallel program.

C. Experimental Results

The results of our parallel GN algorithm are the same as the sequential GN algorithm. The output of the algorithm for the Dolphins network [4, 8] is shown in Fig. 6. The characteristics of network and the experimental results are shown in Tab. 1 and Tab 2. Q presents the modularity of the network (graph). Network scale is the number of vertices plus edges.

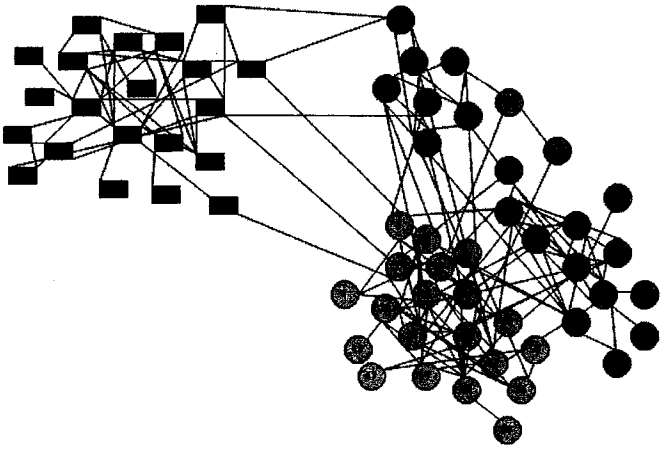

Figure 6. Dolphins network

TABLE I. NETWORK CHARACTERISTICS

Network	Vertex number	Edge number	Network scale
Karate	34	78	112
Dolphins	62	159	221
Polbooks	105	441	546
Football	115	616	731
Jazz	198	2742	2940
Power	4941	6594	11535

TABLE II. EXPERIMENTAL RESULTS COMPARISON

Network	Q	1 Thread's run time(s)	4 Thread's run time(s)	Speedup ratio
Karate	0.538	0.132	0.101	1.31
Dolphins	0.610	0.450	0.302	1.49
Polbooks	0.569	4.507	1.996	2.26
Football	0.662	8.813	3.608	2.44
Jazz	0.318	348.692	115.809	3.01
Power	0.935	23883	12704	1.88

D. Discussion

In experiments, the tendency of speedup ratio with network scale is shown in Fig. 7. We can see that speedup ratio of sequential and parallel program is up to 3 on 4-core processor.

In parallel program, build a copy of graph data for every thread. So it will not change other threads' graph data when every thread is computing betweenness of every edge for one source vertex. We use synchronization mechanism when compute full betweenness of edges. Consequently, the value of modularity is not changed. Through observing the experimental results in Tab. 2, we can know parallel program is faster than the sequential and ensure the correctness of experimental results. Fig. 7 shows that Speedup ratio does not increase immeasurably with the increased network scale. There are three reasons: one is that the number of threads should match the number of processor cores. Another one is that the number of program's tasks should match the number of threads. The last one is that the computation of each task tends to large and equivalent is very important. When these three conditions have been met, the speedup ratio will reach the maximum. Therefore, we can get the better result when execute parallel GN algorithm by coarse-grained parallelism to process the large network on many-core processors. For fine-grained parallelism, it will also apparently raise speed without distributed storage.

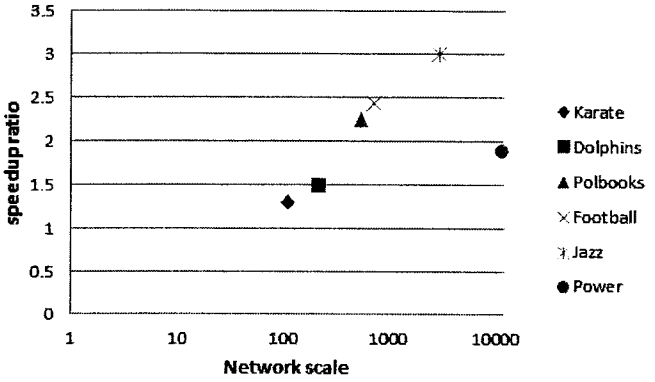

Figure 7. Tendency of speedup ratio with network scale

V. CONCLUSIONS

In this thesis, we have proposed two methods to parallelize GN algorithm and showed that our methods can reduce the execution time to find community structure in complex networks. Specially, we implemented our parallel GN algorithm by coarse-grained parallelism on Intel Core i7, and evaluated our methods using real networks, such as Dolphins network. The evaluation results showed our coarse-grained parallel method can yield the best case performance speed-up 3 times on 4-core processor.

REFERENCES

[1] M. R. Garey and D. S. Johnson, Computers and In-tractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, San Francisco, 1979.

[2] J. Scott, Social Network Analysis. A handbook. Sage Publications, London 2nd edition, 2000.

[3] M.E.J. Newman and M.Givan, Community structure in social and biological networks. Proc. of International Acdemic of Science, 7821-7826, 2002.

[4] M.E.J. Newman and M.Givan, Finding and Evaluation Community Structure in Network. Physical Review E, 69, 2003.8.11.

[5] Wu F, Huberman B A. Finding communities in linear time: A Physics approach [J] Euro. Phys. JB, 38:331-338, 2003.

[6] Scott, J., Social Network Analysis: A Handbook, Sage Publication, London, 2nd edition, 2000.

[7] http://www-personal.umich.edu/~mejn/netdata/.

[8] D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson, The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. Behavioral Ecology and Sociobiology 54, pp 396-405, 2003.

[9] George Karypis, Vipin Kumar, A Coarse-Gramin Parallel Formulation of Multilevel k-way Graph Parationing Algorithm. Parallel processing for scientific computing, SIAM, 1997.