A distributed parallel community detection heuristics for large-scale real graphs (Unrefereed Workshop Manuscript)

Guanglong Chi[1,†1,a)] Ken Wakita[2,†1,b)] Hitoshi Sato[3,†1,c)]

Abstract: In the era of "Big Data", the need for fast, scalable, and high-precision technique for analysis of huge social networks is growing. The paper proposes a distributed implementation scheme of a community structure identification technique known as the Louvain method [5]. Our technique copes with the growing size of the social network by partitioning and distributing the whole social network. It also employs a few approximation techniques and collective communication to increase parallelisms among the computing nodes. The performance is evaluated using various social network data as well as synthesized data.

Keywords: Community detection, Approximation computation, Collective communication, Edge Partition

1. Introduction

To the explosively increasing data, the need for analyzing these data in fast, scalable, and heigh-precision way becomes critical. Community detection is one of the most popular analysis pattern. Among the community detection approaches, there are some fast and scalable methods. Louvain method [5] is one of the fastest and scalable method that can process the graph data up to 100M nodes and 1B edges.

However, even the Louvain method has limitation in scalability, for instance, the Louvain method should need about 377GB memory (It is simulated from the data structure of Louvain method's C++ version implementation) to process a Facebook graph data made up of 149M nodes and 31B edges [6], [7], which obviously can not fit a single computer memory.

In order to process such a large-scale graph, a distributed version of community detection method is necessary. To our knowledge there is only one approach that proposed by Wickramaarachchi et al. [21]. In that work, they adopted graph node partition techineque as a preprocess to decrease the existence probability of communities that span several computing nodes. Through this way, they made the information need to exchange among computing nodes less important and also controlled the number of it to minimum. As a result, they got a scalable method without any noticeable loss in modularity (See detail in the next section.).

In this work, we mainly focus on the memory limitation. We hope to provide a distributed community detection method that could deal large-scale data which can not fit the memory of a single computing node. In addition, we hope to make the most of each computing node resource without losing modularity.

According to the previous work, we also gave a simple distributed Louvain method that adopt node

[1] Department of Human System Science
[2] Department of Mathematical and Computing Sciences
[3] Global Scientific Information and Computing Center
[†1] Presently with Tokyo Institute of Technology, Japan
[a)] chi.g.aa@m.titech.ac.jp
[b)] wakita@is.titech.ac.jp
[c)] hitoshi.sato@gsic.titech.ac.jp

partition as a preprocess in order to minimum communication cost among computing nodes.

However, from the previous work and our simple distributed Louvain method, we find there exists load imbalance problem that caused by graph node partition preprocess. From our preliminary experiment, our dataset showed 11 ∼ 180 times differences in edge size between the max and the minimum one.

The imbalance of edges should lead to the imbalance of memory usage per computing node, and it would influence the scalability. Because it is obvious that the maximum memory usage among the computing nodes should be the bottleneck that limit the scalability of the method. In addition, the imbalance of edges should also cause the difference in computation time per computing node since the computation cost of each computing node should be $O(m')$, when m' indicates the edge size per computing node.

In order to solve this problem, we propose to adopt graph edge partition as a preprocess instead of graph node partition, and evaluate the efficiency in several way.

The main contribution of this paper is that by using edge partition as a preprocess, it takes the balance both in memory usage and computation time that is related to the scalability and the total computation time.

The rest structure of the paper is as follows: Section 2 explains the detail of the original Louvain method and our simple distributed Louvain method, while at the end it represents the problem of the previous work and our simple distributed Louvain detailedly; section 3 gives related works on distributed Louvain and load balancing in distributed graph processing systems; section 4 describes the detail of our proposal; section 5 shows the result of several simple experiments; and section 6 concludes our work while gives some future work.

2. Louvain Method

The Louvain method for community detection is a greedy modularity maximization approach [5]. Modularity [12] is a metric for evaluating the quality of the community detection result and is defined as:

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

where C means the set of all detected communities. $e_{i,j}$ represents the fraction of the edges between community i and community j. $a_i = \sum_j e_{i,j}$ represents the fraction of the edges that connect community i to others.

Since in a graph, an edge falls between two nodes without regarding for the communities that the two nodes of the edge should belong to, we would have $e_{i,j} = a_i a_j$. Then, we can know that a_i^2 is the expected value of $e_{i,i}$. If the detected community is close to the real one, $e_{i,i}$ will greater than it's expected value, a_i^2, otherwise it will less than a_i^2. The closer the detected community to the real one, the greater modularity value will be given to the community, as a result the modularity of the whole graph will become greater.

Louvain method detects communities by inserting each node to the best community that make the modularity of the whole graph maximum. Each node will iteratively move from current community to the best community leading the increase of modularity until there is no nodes' movement.

2.1 Sequential Louvain method

In this section, we will go into detail about Louvain method and in order to distinguish it from distributed Louvain method, we call it sequential Louvain method.

Sequential Louvain method mainly consists of two phases - computation phase and contraction phase, and it runs these two phases iteratively. In the computation phase, it considers each node as a community at initial time. Then, for each node i, the method would compute Δmodularity between node i and its neighbor communities. Δmodularity means the increment of modularity that would take place under the assumption of node i's movement from current community to neighbor community. Finally, node i moves to the community with the largest Δmodularity. If only minus Δmodularity achieved, the node stays in its current community. The computation phase will be repeated several times until there is no nodes' movement, and then it enters the contraction phase. We call one computation phase as one *pass*, while call the whole process from the start of computation until before it enters the contraction phase as one *level*.

In the contraction phase, the method rebuilds the graph by contracting each detected community into one new graph node. An edge exists between two new graph nodes when there are inter-community edges between two corresponding detected communities, and the weight of this edge is calculated by summing up the weight of the inter-community edges. Each new graph node holds self weight achieved from the total weight of intra-community edges. Once the contraction phase is completed, the method would continually do computation and contraction phases with the new graph. Like this, the method would switchover between computation phase and contraction phase repeatedly until no modularity improvement take place over the graph.

Although the sequential Louvain method makes better performance both in modularity and speed than others, there is only one approach about distributed Louvain method as we know. In the next section we give a simple distributed Louvain method, and point out some problems existing both in our simple distributed Louvain method and the previous studies.

2.2 Distributed Louvain method

Simple distributed Louvain method In order to detect communities in distributed way, we need to carry out two basic steps, graph data divide and community detection. In the graph data divide step, we need to divide a large-scale graph into several subgraphs with each subgraph size is smaller than computer's memory size. In the community detection step, we read each subgraph per computing node, then detect communities included in each subgraph by running the Sequential Louvain method.

When dividing a large-scale graph into subgraphs, some edges that two ends of which fall in two different computing nodes should be cut (We call these edges *cross edges*. See figure 1.). In order to compute the Δmodularity value of the cross edge's two ends node-neighbor communities, it needs to exchange information through the cross edge by communication. However, if we assume a graph data has m edges, and one want to divide the graph into p subgraphs, then one need to communicate about $\frac{m}{p}$ nodes' information separately per computation step. It is obviously consume too much time while community detection

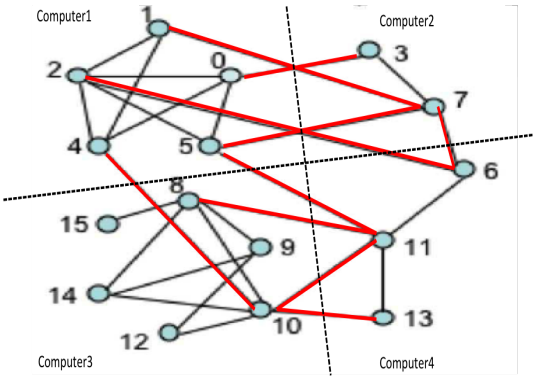

Fig. 1 Suppose that a graph is divided into four subgraphs. Cross edges (the red edges) should be occur between computing nodes.

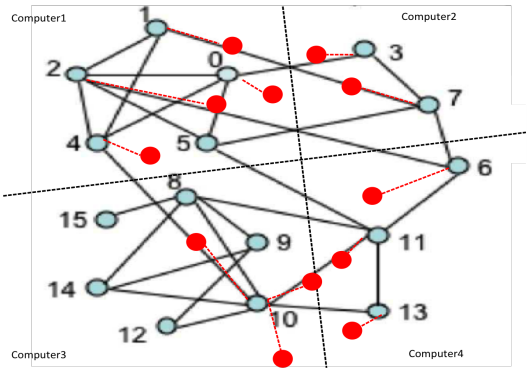

Fig. 2 Store the remote graph node's information locally. For example, store information of nodes {3, 7, 10 ,11} in computing node 1 when the graph is divided as Figure 1, as the edges {(0, 3), (1,7), (2, 6), (4, 10), (5,7)} become cross edges.

to each subgraph can run in $O(\frac{m}{p})$ complexity.

In view of the problem mentioned above, we propose a simple distributed Louvain method (*SDLouvain*). In the method, for reducing communication cost we store the remote node's information locally (We call these nodes as *border nodes*). For example, store information of node 7 in computing node 1, since node 7 is belong to computing node 2 and (1, 7) is a cross edge. Then using these locally stored border nodes' information approximately compute Δmodularity related to cross edges. We call it approximation computation, because the locally stored information is not updated as soon as the remote information updated.

Instead of exchanging information at each time the remote information updated, the method just

communicate information after a certain number of Δmodularity computation have been done. As shown in the computation phase's pseudocode of this method (Algorithm 1), the communication should be needed when aggregate neighbor communities information (line 3) if there are cross edges. But in this method, it just communicates all graph nodes are computed one time. In this way, it reduces communicate time efficiently, since it can exchange a set of remote nodes' information in collective way rather than do it separately. Additionally, it alleviates the modularity from becoming bad.

In communication part of computation phase (line 13), the method mainly exchange the information of neighbor communities that belong to remote computing node. However, an additional communication should be needed when a community spans several computing nodes. In this case, before exchanging the information of remote community, it is need to

- gather the information of the same commmunities into one computing node,
- aggregate the information,
- scatter the updated information to each computing node that the communitiy belongs to.

In consideration of communication cost, this additional communication steps (*aggregation step* in communication part) are also not done each time the community information updated, but just be done at the first of the communication part. The computing node that would gather and aggregate the information of communities should be decided by the communities' id. When a graph data has n nodes and is dealed by P computing nodes, the information of community i, $i \in [p_i * \frac{n}{|p|}, (p_i + 1) * \frac{n}{|p|}), p_i \in P$ should be gathered and aggregated in computing node p_i.

Discussion In the above method, it mainly focused on the computation phase of the community detection step, and through approximation technique and collective communication, it efficiently reduced the number of communicate times. However, as we mentioned, the communication volume of each computing node is up to $O(\frac{m}{p})$ in one computation phase, which is the same as computation phase itself. It is obvious that the communication cost should become a bottleneck of the whole community detection time. In order to reduce the communication volume, graph partition methods are considerable, since they can help to divide a graph data into subgraphs with minimum cross edges between them.

Algorithm 1 *Computation Phase*

```
Require: 𝔾 = {𝕍, 𝔼}
 1: do
 2:     for ∀v ∈ 𝕍 do
 3:         ℂ′ ⇐getNeighborCommunity(v);
 4:         for C_i ∈ ℂ′ do
 5:             ΔMod ⇐getDeltaMod(v, C_i);
 6:             if ΔMod > ΔMod_max then
 7:                 ΔMod_max ⇐ ΔMod;
 8:                 bestComm ⇐ C_i;
 9:             end if
10:         end for
11:         insert(v,bestComm);
12:     end for
13:     communicateRemoteInfo(𝕍′ ∈ V);
14: while modularityImprovement
```

Wickramaarachchi et al.[[21]] did some studies by using a node graph partition method PMETIS[[14]], though their purpose is different somewhat from this paper. They supposed that the probability of cross communities' existence should become lower if one divide graph data by PMETIS. Since there are little cross communities, each computing nodes can detect local communities successfully without exchange information. Their results showed that their program could achieve almost the same modularity value as sequential Louvain method, even though it ignored communications between computing nodes.

From the previous work, we can know that PMETIS is efficient enough to reduce the probability of cross communities' existence, but in order to verify the efficiency of node graph partition method (ex.PMETIS) more, we did another preliminary experiment. We implemented the *SDLouvain* method with dividing graph data by PMETIS in graph divide step, and compared the size of subgraphs.

As a result, we find there are big differences in edge size among achieved subgraphs. For example, with our dataset (See section 5.3 for details), it showed 11 ~ 180 times differences in edge size between the max and the minimum one, which should lead to 2 ~ 7 times differences in memory when dividing the graph into 8 subgraphs. (The difference of memory is

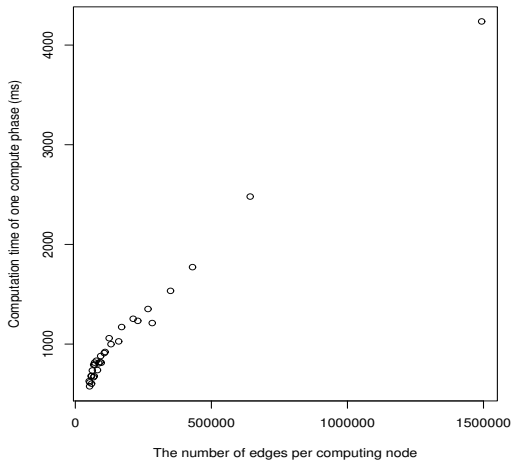

Fig. 3 This figure shows the relationship between the number of edges and computation time. From it, we can see a linear correlation. It uses YouTube graph data (#Nodes=1,134,892, #Links=5,995,248), which shows friendship among the YoutTube users.

simulated from the data structure used in the sequential Louvain method implementation.) It means that PMETIS causes memory imbalance, while it is undesirable since the max memory consumption of computing nodes should be a bottleneck.

On the other hand, because the Louvain method needs to iterate each edges in computation phase, the difference in edges should also lead to the difference in computation time. In this case, the computing node that completed computation phase quicker have to wait others, since the SDLouvain needs to exchange information at the end of computation phase (See algorithm 1, figure 3).

From the preliminary experiment result, we can sum up that there exists edge imbalance problem among subgraphs, which should lead to the imbalance of compute times and memory consumption. In this work, we mainly focus on this problem.

3. Related work

Distributed Louvain As we mentioned in section 2.2, there is only one approach about distributed Louvain method proposed by Wickramaarachchi et al [21]. The work succeed in reducing the number of communities that span several computing nodes when dividing the graph data by taking advantage of the graph partitioning method. To the divided subgraphs with little cross communities, it runs the sequential Louvain method without any communication. It just detects communities in distributed way for the first level, because most of the time is spent in the first level.

After the first level, it contracts the subgraph of each computing node into new subgraph separately and send the new subgraph to master computing node for merging them and running in sequential way after the first level. Computing in distributed way just for the first level is also one of the reasons that the method can complete the detection without any noticeable loss in final modularity, since the levels after the first one can play a part in replenishing the modularity value that lost in the first level.

In this work, we also apply the same strategy because of the significant reduction of graph size after the first level (in our dataset the graph after the first level reduced to 10% $\sim$ 50% in node size and 5% $\sim$ 24% in edge size), which means the contracted new graph may be smaller than the computer memory, even though the original graph exceeds the memory.

Load balancing in distributed graph processing Dividing a large-scale graph into k subgraphs is the first step for distributed graph processing. In general, for minimizes the total solution time, some strategies need to be adopted in this step. The goal of these strategies should be balance the divided subgraph size and minimize the number of cross edges between them.

Many distributed graph processing systems utilize graph partition method to pursue the above goal. Giraph [1] uses hash- or range-based graph partitioning methods, while GraphLab [18] and GoldenOrb [2] use multilevel k-way graph partition method [15].

In addition, approaches about distributed graph partition are also proposed widely. ParMETIS [16] and PT-SCOTCH [9] provide a distributed multilevel graph partition method by using multilevel schemes, while the approach of Kirmani et al. [17] divide the graph data in distributed way using geometric mesh partitioning schemes [11].

Most of the graph partition methods provided until now equipartition the graph nodes, while there are also some needs on dividing a graph into subgraphs

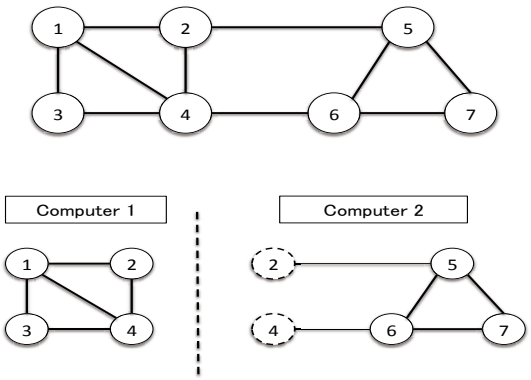

Fig. 4 Assuming that there is a graph with 7 nodes and 10 edges. Edge partition divides this graph into two subgraphs with 5 edges per subgraph. In this case, we call node 2 ,4 in computing node 1 as *master nodes*, while those in computing node 2 as *replicas*.

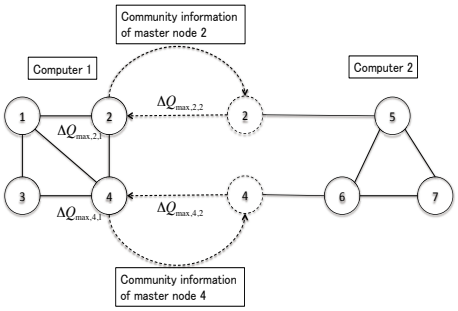

Fig. 5 In ICLouvain method, the communication

with equal edges in practical computation, such as Louvain method.

Moreover, it is observed by Gonzalez et al. [13] that some existing graph node partition method should perform poorly when a graph data has scale-free property. Motivated by this problem, they provided a graph edge partition method (PowerGraph, PG), while Bourse et al. [8] extended PG that can achieve subgraphs balanced both in nodes and edges (the least incremental cost assignment, IC).

In this work we want to adopt the work of Bourse et al. because the problem we find in section 2.2 are mainly related to the subgraph edges' imbalance.

4. Proposal

Preprocess In this work, we propose to use balanced edge partition method as preprocess for diving a large-scale graph into k subgraphs, since, as we see in the above, it can achieve balance both in nodes and edges, which is benefit to improve total time and balance memory consumption.

Edge partition divides graph edges into equal size, therefore each edge should be counted only one time and belong to a unique computing node. It leads to multiple existence of one node among computing nodes (Such as node 2, 4 in figure 4 exist both in computing node 1 and 2). In that case, we call one of the nodes as *master node*, while others as *replicas*. Thus, in figure 4, node 2 has one master node and one replica.

Communications should occur among master node and it's replicas, while it occurs between two ends of a cross edge in node partition. In the work of Bourse et al. [8], it is proved that the communication cost of node partition should be greater than edge partition when assigning nodes or edges one of k partitions independently and uniformly at random. However, when it comes to use PMETIS and IC, we find that IC shows greater communication cost than PMETIS (up to 4.5 times in our dataset) when the graph has low scaling exponent (≈ 1.4. See detail in section 5.3). In contrast, with the graph holding high scaling exponent value (1.7~2.4), PMETIS shows greater communication cost that is up to 9 times in our dataset.

Therefore, we consider that IC is fit for the graph divide step, since edge load imbalance mainly occurs in the scale-free graphs.

Computation phase The mechanism of computation phase should be different from SDLouvain, since the information that stored in each computing node is different. For instance, in the figure 4, when the graph is divided into two subgraphs, computing node 1 only holds parts of graph node 2's neighbor information, since node 2's neighbor node 5 belongs to computing node 2. Thus, when the compute phase comes to node 2, it can't compute Δmodularity between node 2 and the community that node 5 belongs to. It means node 2 can't move to the community with max Δmodularity value. In order to alleviate this problem, we consider to do as follows:

- Store Δmodularity value of each master node and it's replicas per computing node. In figure 5, computing node 1 stores max Δmodularity of master node 2 and 4 ($\Delta Q_{max,2,1}$, $\Delta Q_{max,4,1}$), while the max Δmodularity values of replica 2 and 4

($\Delta Q_{max,2,2}$, $\Delta Q_{max,4,2}$) are stored in computing node 2.

- Gather the stored values in one computing node that the corresponding graph nodes' master nodes belong to. Since the master node 2 and 4 are in computing node 1, computing node 1 will gather the values ($\Delta Q_{max,2,1}$, $\Delta Q_{max,2,2}$, $\Delta Q_{max,4,1}$, $\Delta Q_{max,4,2}$).
- Each master node selects the max of the max Δmodularity values that gathered from each computing node and updates their information according to it. In figure 5, master node 2 selects a max value form the set $\{\Delta Q_{max,2,1}, \Delta Q_{max,2,2}\}$, and updates the community information of it.
- Scatter the updated master nodes' community information to each replicas. In figure 5, computing node 1 scatters updated community information of master node 2 and 4 to each replicas.

Above four steps should be done per one pass computation completed, and then it enters into the next pass computation or contraction phase.

5. Experiments

In this section, we show the result of preliminary experiment and compare PMETISSDLouvain (divide graph nodes by PMETIS) and ICLouvain on computation time and memory consumption per computing node to evaluate the efficiency of edge partition.

All of the programs are implemented by X10 [3], a parallel distributed programming language provided by IBM. The experiment would be done on Tokyotech Supercomputer and UBiquitously Accessible Mass-storage Environment (TSUBAME) cluster. We used two *THIN* computing nodes of TSUBAME, while there are heterogeneous computing nodes on it. Each *THIN* computing node consists of 54GB RAM and two six-core Intel Xeon X5670 2.93GHZ processors. Each program was executed 5 times and we show the average of all results in this work. The experiments of all rounds presented stable results per graph data.

Twelve real graph data were used in the experiments. All of the datasets used here are downloaded from Stanford University's Dataset Collection homepage. (`https://snap.stanford.edu/data/`)

Table 1 Real Graph Datasets

	#Nodes	#Edges	Description
Wiki-Talk	2,394,385	9,319,130	Web graphs
Pockec	1,632,803	44,603,928	Social
YouTube	1,134,892	5,995,248	Social
Web-Google	875,713	10,210,078	Web graphs
Amazon	334,864	1,851,748	Co-purchasing
DBLP	317,080	2,099,732	Co-authorship
Web-NotreDame	325,729	2,966,813	Web graphs
Loc-gowalla	196,591	3,801,308	Social
BrightKite	58,228	428,156	Social
Email-Enron	36,692	367,662	Communication
ca-AstroPh	18,772	396,100	Co-authorship
Wiki-Vote	7,115	207,378	Adminship Election

5.1 Efficiency of edge partition

We observed the load balance and communication cost of IC in preliminary experiment. Here, the communication cost indicates the number of graph nodes that need to exchange information.

Edge / Node balance First of all, we compared the subgraphs' edge size in three cases (Random, METIS, IC partition). In table 2, we present the ratio between maximum and minimum edge size of subgraphs, when dividing a graph into 8 subgraphs. It shows that the METIS is not efficiency in edge balance, which is sometimes even worse than random partition. However, when using IC, the ratio becomes nearly to 1.

Table 2 $|\text{Edge Size}|_{max}/|\text{Edge Size}|_{in}$ in subgraphs

	#Random	#METIS
Wiki-Talk	15.3	5.5
Pockec	21	1.2
YouTube	10.9	10.9
Web-Google	1.2	2.1
Amazon	1.0	1.2
DBLP	3.2	1.6
Web-NotreDame	3.2	3.4
Loc-gowalla	10.6	15.5
BrightKite	16.4	11.5
Email-Enron	17.9	17.8
ca-AstroPh	1.1	9.5
Wiki-Vote	3.3	23.8

Communication cost Since it is obviously considerable that edge imbalance of subgraphs might occur when there are hubs in the graph data, we fitted

power-law distribution to all of the data according to the method of Clauset, Shalizi and Newman [10]. The result shows the power law is a plausible hypothesis for the data, and with the result in mind, we compared the communication cost.

Table 3 Communication cost per graph partition method.

	α	Random	METIS	IC
Wiki-Talk	2	2,191,822	1,270,747	147,230
Pockec	2.4	6,244,429	3,132,701	3,482,163
YouTube	1.7	1,594,982	535,394	401,805
Web-Google	1.4	3,154,762	107,640	620,131
Amazon	1.3	1,040,006	57,789	235,225
DBLP	1.4	661,924	141,304	215,119
Web-NotreDame	1.7	386,576	39,514	136,349
Loc-gowalla	1.4	412,292	172,965	172,511
BrightKite	1.8	107,654	57,354	40,344
Email-Enron	1.7	50,535	31,086	24,582
ca-AstroPh	1.9	85,423	29,494	28,995
Wiki-Vote	2.1	16,827	15,363	9,296

Table table 3 shows the communication cost when the graph is divided into 8 subgraphs. In the table, α in table means the scaling exponent of power law distribution. From it, we find that IC makes the same or less communication cost when the data has high scaling exponent. It means that if we adopt IC as preprocess, the communication time should also be benefited.

5.2 Memory consumption

In order to observe memory consumption and time consumption of each computing node, we ran the PMETISLouvain and ICLouvain method with the top two data from table 1, since they have highest scaling exponent values and largest scale. We used 16 processers since the data Pockec has no imbalance in 8 partitions.

Both in PMETISLouvain and ICLouvain, the memory consumed through the whole community detection can divide into three parts. The first part is used to store graph information such as degrees, edges and the weights of edges. The second part is used to store border / replica nodes that need to exchange information, while the last part consumes memory as communication buffers. The memory usage of the first two parts would be constant, since the subgraphs read by each computing node would never change and also the number of cross edges or replicas would never change. However, the third part would use memory dynamically. Because in the aggregation step of communication part, the number of communities fall in

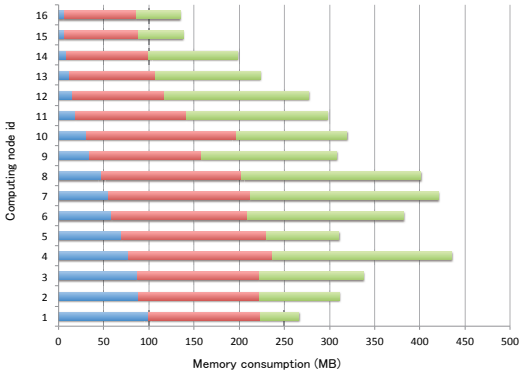

Fig. 6 Memory consumption of Pokec preprocessed by METIS.

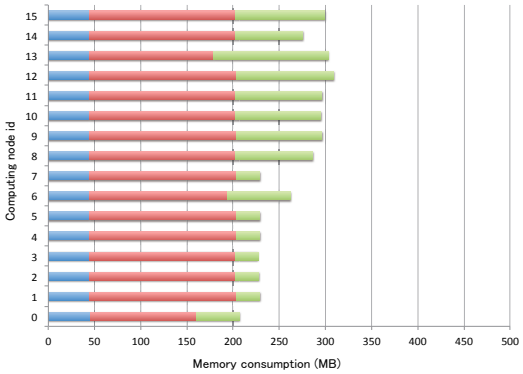

Fig. 7 Memory consumption of Pokec preprocessed by IC.

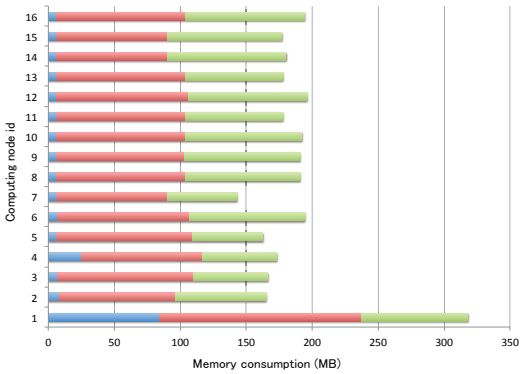

Fig. 8 Memory consumption of Wiki-Talk preprocessed by METIS.

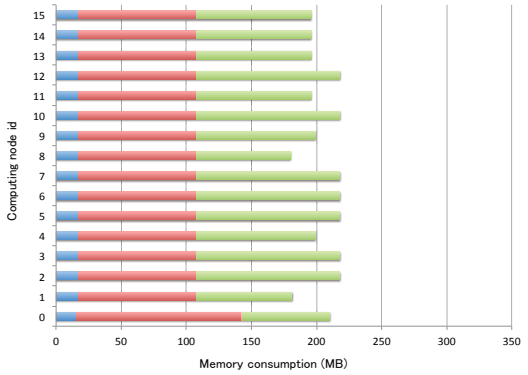

Fig. 9 Memory consumption of Wiki-Talk preprocessed by IC.

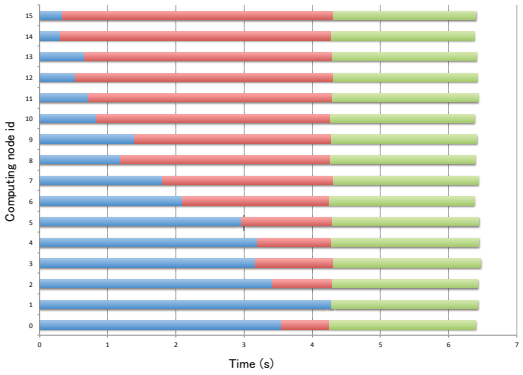

Fig. 10 Time consumption of the first pass in PMETISLouvain with Pokec.

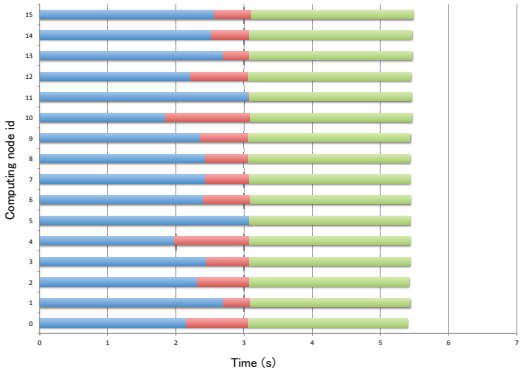

Fig. 11 Time consumption of the first pass in ICLouvain with Pokec.

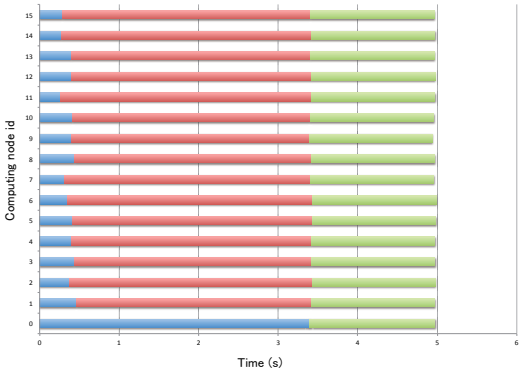

Fig. 12 Time consumption of the first pass in PMETISLouvain with Wiki-Talk.

the range $[p_i * \frac{n}{|P|}, (p_i + 1) * \frac{n}{|P|})$ should dynamically change and imbalance.

According to the above, we represented the memory consumption in stacked bar chat. Figure 8 and 9 show the result of Pokec. The three parts in one bar means the memory usage of graph data initialization, borders / replicas and communication from left to right. From the figures, we can see that the edge size of subgraphs get balanced along with the node size, and the maximum memory consumption per computing node is controlled to 71% of the one used in PMETISLouvain. In addition, the number of borders / replicas is also get balanced, which means the wait time of exchanging borders / replicas information part should also be controlled to the minimum.

However, the imbalance of memory consumption in communication part can be seen both in PMETISLouvain and ICLouvain. It should be our future work since this imbalance is the bottleneck at the current stage.

Almost the same result can be seen from the result of Wiki-Talk (Figure 8 and 9).

5.3 Comparison of Time

In this part, we observe the time structure of one pass in order to know whether the computation time imbalance is alleviated by edge partition. Communication time is also be compared between PMETISLouvain and ICLouvain.

Time balance Figure 10, 11, 12, 13 show the time consumption in the first pass per computing node in PMETISLouvain and ICLouvain.

The time of one pass consists of two parts, computation part and communication part, and between the two part, some wait time occur due to all computing nodes need to synchronize information after computation part, while the computation time is different per

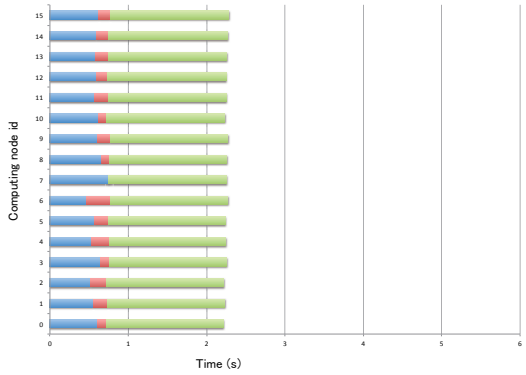

Fig. 13 Time consumption of the first pass in ICLouvain with Wiki-Talk.

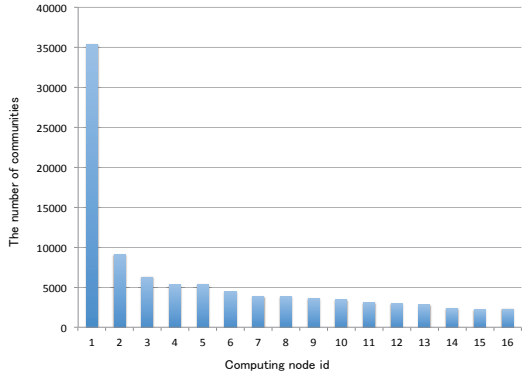

Fig. 14 The distribution of communities in the first aggregation step with Wiki-Talk.

computing node. The tree parts in the figure indicates computation time, wait time and communication time in order from left to right.

In the figure, the total time of each computing node is equal because the communication part would run while doing barrier synchronization in each communication step.

The figures show that the computation time per computing node is balanced well, and the total time conserved 16% in Pokec, 55% in Wiki-Talk.

However, from the total time structure, we can see that the proportion of communication time is up to 43 % in Pokec and 66% in Wiki-Talk.

Additively, when we observe the detail of communication part, the aggregation step consumes 51% of communication time in Pokec, while it up to 82% in Wiki-Talk. The balance of aggregation step also shows imbalance (See figure 14, 15), though would be different by each data. Thus, balancing the number

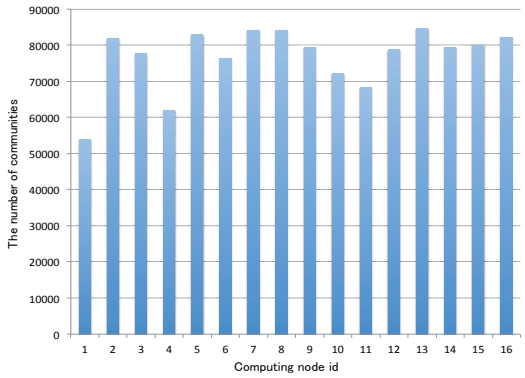

Fig. 15 The distribution of communities in the first aggregation step with Pokec.

of communities per computing node in aggregation step should cut down the communication time. As the memory imbalance of aggregation step, this is one of our future work.

6. Conclusions and future work

6.1 Conclusion

There are several approaches [4], [19], [20] about parallel Louvain method that mainly focus on computation time without noticeable loss in modularity. In contrast, there is only one approach [21] on distributed Louvain method. In that work, they proposed to adopt node partition technique as a prepocess to reduce the communication cost among computing nodes.

In this work, we forcused on providing a distributed community detection method that could deal large-scale data which can not fit the memory of single computing node. It is obvious that the maximum memory usage among the computing nodes should be the bottleneck that limit the scalability of the method. Thus, in this case, node partition is not suitable because node parititon only take the graph node balance and it would lead to edge imbalance if there are hubs in the graph data.

In our work, we adopted edge partition as preprocess, which can take balance both in nodes and edges. Furthermore, the balance of computation time per computing node is also considerable due to the Louvain method is a iterative algorithm that the computation of each iteration per computing node should be done in $O(m')$. Here, m' means edge size of the sub-

graph that falls into each comuting node. Also, some approximation techniques and collective communication are used to reduce the communication cost.

The results of our experiment show some efficiency of edge partition method in memory consumption, computation time and communication time. However, it is still far away from our goal. We just proposed the basic distributed model at the current stage, and kinds of future work are considerable.

6.2 Future work

First of all, we hope to know what kinds of data are suitable to process by ICLouvain. Though we gave roughly tendency in table 3, but it is not correct in many data. Such as, Web-NotreDame data has more communication cost in IC, while it has high scaling exponent. Amazon data never shows imbalance, while Web-Google and DBLP with the near scaling exponent show imbalance in data partition. There are also some data (e.g. Pockec) show imbalance over a fixed number of processor number.

In addition, some sensible polilcy is needed to balance the memory consumption of aggregation step. We consider to decide the range of community id that one computing need to process as $[p_i * \frac{|C|}{|p|}, (p_i+1) * \frac{|C|}{|P|})$ instead of $[p_i * \frac{n}{|p|}, (p_i+1) * \frac{n}{|P|})$, when C indicates the set of communities over the whole graph data. In order to know C size just one reduction communication is needed.

Some strategies such as only exchange changed community information or just gather and aggregate cross communities information are also need to be done.

The interval of communications is also worth to consider, since if it can take larger interval than one pass, the shortening of the total time is available.

Finally, the study about the difference in the number of detected communities and its structures is also necessary for evaluating the detection result in more detail.

References

[1] Giraph: `http://giraph.apache.org/`.
[2] Goldenorb: `http://goldenorbos.org/`.
[3] X10: `http://x10-lang.org/`.
[4] Bhowmick, S. and Srinivasan, S.: A template for parallelizing the louvain method for modularity maximization, *Dynamics On and Of Complex Networks, Volume 2*, Springer, pp. 111–124 (2013).
[5] Blondel, V. D., Guillaume, J.-L., Lambiotte, R. and Lefebvre, E.: Fast unfolding of communities in large networks, *Journal of Statistical Mechanics: Theory and Experiment*, Vol. 2008, No. 10, p. P10008 (2008).
[6] Boldi, P., Rosa, M., Santini, M. and Vigna, S.: Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks, *Proceedings of the 20th international conference on World Wide Web*, ACM Press (2011).
[7] Boldi, P. and Vigna, S.: The WebGraph Framework I: Compression Techniques, *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, Manhattan, USA, ACM Press, pp. 595–601 (2004).
[8] Bourse, F., Lelarge, M. and Vojnovic, M.: Balanced graph edge partition, *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, pp. 1456–1465 (2014).
[9] Chevalier, C. and Pellegrini, F.: PT-Scotch: A tool for efficient parallel graph ordering, *Parallel computing*, Vol. 34, No. 6, pp. 318–331 (2008).
[10] Clauset, A., Shalizi, C. R. and Newman, M. E.: Power-law distributions in empirical data, *SIAM review*, Vol. 51, No. 4, pp. 661–703 (2009).
[11] Gilbert, J. R., Miller, G. L. and Teng, S.-H.: Geometric mesh partitioning: Implementation and experiments, *SIAM Journal on Scientific Computing*, Vol. 19, No. 6, pp. 2091–2110 (1998).
[12] Girvan, M. and Newman, M. E.: Community structure in social and biological networks, *Proceedings of the national academy of sciences*, Vol. 99, No. 12, pp. 7821–7826 (2002).
[13] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D. and Guestrin, C.: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs., *OSDI*, Vol. 12, No. 1, p. 2 (2012).
[14] Karypis, G. and Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM Journal on scientific Computing*, Vol. 20, No. 1, pp. 359–392 (1998).
[15] Karypis, G. and Kumar, V.: Parallel multilevel series k-way partitioning scheme for irregular graphs, *Siam Review*, Vol. 41, No. 2, pp. 278–300 (1999).
[16] Karypis, G., Schloegel, K. and Kumar, V.: Parmetis: Parallel graph partitioning and sparse matrix ordering library, *Version 1.0, Dept. of Computer Science, University of Minnesota* (1997).
[17] Kirmani, S. and Raghavan, P.: Scalable parallel graph partitioning, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ACM, p. 51 (2013).
[18] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A. and Hellerstein, J. M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud, *Proceedings of the VLDB Endowment*, Vol. 5, No. 8, pp. 716–727 (2012).
[19] Lu, H., Halappanavar, M. and Kalyanaraman, A.: Parallel heuristics for scalable community detection, *Parallel Computing* (2015).
[20] Staudt, C. L. and Meyerhenke, H.: Engineering high-performance community detection heuristics for massive graphs, *Parallel Processing (ICPP), 2013 42nd International Conference on*, IEEE, pp. 180–189 (2013).
[21] Wickramaarachchi, C., Frincu, M., Small, P. and Prasanna, V. K.: Fast parallel algorithm for unfolding of communities in large graphs, *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, IEEE, pp. 1–6 (2014).

Errata

- **Author affiliations**
  The affiliations of all the authors should be:
  “Tokyo Institute of Technology and JST, CREST.”

- **Acknowledgment**
  The following acknowledgement should be added at the end of the paper:
  “This work was partially supported by the JST-CREST Project.”