Parallel Community Detection on Large Networks with Propinquity Dynamics*

Yuzhou Zhang†
yuzhou.zh@gmail.com

Jianyong Wang‡
jianyong@tsinghua.edu.cn

Yi Wang§
wyi@google.com

Lizhu Zhou♯
dcszlz@tsinghua.edu.cn

†‡♯Department of Computer Science and Technology
Tsinghua University
Beijing 100084, China

§Google Beijing Research
Beijing 100084, China

ABSTRACT

Graphs or networks can be used to model complex systems. Detecting community structures from large network data is a classic and challenging task. In this paper, we propose a novel community detection algorithm, which utilizes a dynamic process by contradicting the network topology and the topology-based propinquity, where the propinquity is a measure of the probability for a pair of nodes involved in a coherent community structure. Through several rounds of mutual reinforcement between topology and propinquity, the community structures are expected to naturally emerge. The overlapping vertices shared between communities can also be easily identified by an additional simple postprocessing. To achieve better efficiency, the propinquity is incrementally calculated. We implement the algorithm on a vertex-oriented bulk synchronous parallel(BSP) model so that the mining load can be distributed on thousands of machines. We obtained interesting experimental results on several real network data.

Code and Data Sets: The source code(without Google proprietary parts) and data sets for the algorithm are available at: http://dbgroup.cs.tsinghua.edu.cn/zhangyz/kdd09/

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Data mining*

General Terms

Algorithms, Performance

*This work was supported in part by National Natural Science Foundation of China under grant No. 60833003 and 60873171, 973 Program under Grant No. 2006CB303103, a research award from Google, Inc., Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (abbr. TNList), and the Program of State Education Ministry of China for New Century Excellent Talents in University under Grant No. NCET-07-0491.

KDD'09, June 28–July 1, 2009, Paris, France.

1. INTRODUCTION

Graphs(or Networks, preferred by physicists) can be used to model a variety of complex systems, such as social communication network, biological interaction network, paper citation and coauthor network, and web linkage graph. A distinguishing property of real-life graphs other than random graphs is the ubiquitousness of community structures, which are intuitively highly intra connected subgraphs with relatively sparse connections to the leaving parts. The interpretation of community structures varies with the application domain. For example, community structures in social network may imply a group of people who share common interests or simply live near, depending on the context. Web pages included within a community structure discovered from web linkage graph deal with related topics. Community structures in biological network help identify functional module[14]. Emerging Internet services have brought abundant traditionally unattainable graph data online. For instance, the social network service(SNS, e.g. Orkut, MySpace, and Facebook) saves the sociologist from questionnaire survey, and generates large online social network. The hyperlinks between Wikipedia pages form an encyclopedia network with millions of articles as vertices. Detecting and interpreting community structures from various newly available large online graph data is of great significance and challenge.

Although the community detection problem can be intuitively and simply described as discovering coherent parts from network, there is no widely accepted formalized definition. The most strict community structure definition is by clique, which means the fully connected subgraph. Certainly the clique is too strict to be realistic. There are several relaxed clique definitions, like quasi-clique, in which each vertex is connected with at least a minimum proportion of other vertices[20]. Some other community definitions require that the inner community edges exceed the inter community edges[17]. Regardless of the community overlapping, in its simplest form the community detection means dividing the graph into disjoint sets. Therefore some researchers do not directly define what is a community structure but define a quality function to quantitatively evaluate a graph division scheme. The community detection problem is then transformed to optimizing the quality function with proper graph division[11, 9, 5, 12]. The mostly referred quality function is the modularity [13]. Other algorithms even have

no explicit definition or objective function, but can naturally generate community division scheme through reasonable heuristics[18, 1]. The algorithm proposed in this paper belongs to this class.

Other aspects deserving attention is community overlap and hierarchy. In some real networks, it is not easy to clearly divide the graph into disjoint sets. There are swing vertices lying at the boundary between communities, causing the difficulty to decide which community they belong to. A considerate algorithm should be able to identify the community overlaps[14]. Our algorithm can meet this requirement through simple postprocessing. Another natural phenomena related to the community structure is their hierarchical organization, which means a community obtained under coarse granularity can be further divided into more compact ones.

There have been many interesting community detection algorithms, why do we propose another? The primary motivation of this work is we need a more efficient algorithm to discover community information from web scale graph data. Few existing algorithms can finish within $O(|V|^2)$ on sparse graph, which is unacceptable on very large and even dense graph. The complexity of our algorithm on sparse graph is $O(k \cdot |V|)$, where k is the iteration count. Another difference from the algorithms proposed mainly by physicists is that we emphasize on the scalability aspects without loss of community quality. The basic idea of our algorithm is that, the community structure can naturally emerge by a self-organizing dynamic process, which is easily comprehended. Incremental techniques are adopted for further efficiency improvement. We implement the algorithm on special vertex-oriented parallel model, which helps distribute the computation to thousands of machines. The experimental results indicate that our algorithm is competitive with respect to both community quality and mining efficiency.

2. RELATED WORK

A historical problem related to community detection is graph partitioning[10], the problem with which is the number of communities must be specified before hand. To avoid trivial communities the approximate size of the expected community also has to be given as input parameter. Both are supposed to be parts of running results for a successful community detection algorithm. There is a class of algorithms by iteratively removing the edges that lie between communities[6]. The key is defining the edge centrality, which indicates the possibility for an edge lying between communities. Several centrality definitions were proposed: edge betweenness[8], current-flow betweenness[8], random-walk betweenness[8], information centrality[7], edge clustering coefficient[17]. The algorithm in [17] is the more efficient one in the three with time complexity of $O(|E|^4/|V|^2)$. Since Newman et al proposed the null model based modularity[13], there emerged a class of community detection algorithms with the target of maximizing modularity. The modularity optimization techniques proposed by the literature include greedy algorithm[11], simulated annealing[9], extremal optimization[5], spectral optimization[12]. [3] improves the efficiency of the greedy optimization in [11] by achieving the time complexity of $O(|V| \log^2 |V|)$, which is very efficient. But the problem with the greedy modularity optimization is its ineffectiveness on finding accurate community. The extremal and spectral optimization algorithm can both scale to $O(|V|^2 \log |V|)$ on sparse graphs. Another class of community detection algorithms exploit the spectral property of the Laplacian matrix[4] or normal matrix[2]. [18] maps the graph onto a q-Potts model with nearest-neighbors interaction. In [16], the authors utilized the heuristic that random walk mostly happens within the community. [1] takes use of the synchronization dynamics to reveal the hierarchical community structure.

3. COMMUNITY DETECTION WITH PROPINQUITY DYNAMICS

The necessity of a community detection algorithm is because of the belief that there are abundant community structures hidden in real networks. Now that the community structures are right there, why do we work so hard to mine them? Why not just let the communities claim themselves? In the real social network, the implicit interpersonal communities are progressively and spontaneously formed by the collaborative local decision of each individual. If this spontaneous process continues with more aggressive local criterion, the community structures are expected to emerge as natural results. Based on this intuition, in this section we first describe the dynamic process which we term as *propinquity dynamics* and its incremental form. Then we give one concrete definition of propinquity and show how to calculate it efficiently.

3.1 Propinquity Dynamics

In some clustering algorithms, it is a elementary component to define the similarity or distance between a pair of objects. With respect to the graph or network, we need a quantity to evaluate the probability that a pair of vertices are involved in a coherent community. We name this quantity as *propinquity*, which is a term borrowed from sociologist and refers to the physical or psychological proximity between people. Certainly there are a number of alternative heuristics to concretely define the propinquity. Here we simply assume we had one, denoted by $P_G(v_1, v_2)$, in which subscript G will be dropped if there is no ambiguity. Note that, the propinquity must be calculated purely from network topology and be sensitive to the topology changes.

For a graph containing disjoint densely connected components, one can easily identify the community structures. However for real networks, the topology is not that clear. An intuitive way for identifying the community structures may be by purposely increasing the graph contrast. In order to maximize the community contrast, we can utilize the contradiction between propinquity and topology. Specifically, the graph topology should be updated to keep consistent with the propinquity calculated from itself.

$$\begin{aligned}&\text{If } P(v_1, v_2) \le \alpha \text{ and } (v_1, v_2) \in E(G), \text{ then}\\ &E(G) \leftarrow E(G) - \{(v_1, v_2)\}.\\ &\text{If } P(v_1, v_2) \ge \beta \text{ and } (v_1, v_2) \notin E(G), \text{ then}\\ &E(G) \leftarrow E(G) + \{(v_1, v_2)\}. \end{aligned} \quad (1)$$

For conceptual clarity, this paper focuses on undirected non-weighted graph without multi edge and self loop. The graph is represented by $G(V, E)$, where V is the vertex set and E is edge set. From local perspective, updating topology according to propinquity means cutting existing edges or inserting new edges. The updating criteria is shown in Equation 1, where α is the minimum propinquity that can support the survival of an edge and is called the *cutting threshold*. β

is the *emerging threshold* that controls whether a new edge should be inserted in order to guarantee the consistency between local topology and propinquity.

From the global perspective, given a graph topology a set of propinquity values can be calculated. Let $\mathbb{P}(T)$ denote the overall propinquity set of topology T. According to the local topology updating criteria, we use $\mathbb{C}_{\alpha,\beta}(T,P)$ to denote the overall process that contrasting topology T and propinquity P. It returns a new topology, say P', which should be more consistent with propinquity than the previous one. Then the propinquity of the new topology can be further calculated by $\mathbb{P}(T')$. This process goes on iteratively until we have a satisfactory consistency between the topology and propinquity. The final topology should be a good material for the next steps of community detection. Let $T_0 \leftarrow G$, then the process can be represented by Equation 2. We term the iterative process in Equation 2 as *propinquity dynamics*. The termination condition of the propinquity dynamic process should be $|T_{n+1} - T_n| < \epsilon$, where ϵ is a nonnegative integer.

$$\begin{aligned} P_n &\leftarrow \quad \mathbb{P}(T_n) \\ T_{n+1} &\leftarrow \quad \mathbb{C}_{\alpha,\beta}(T_n, P_n), \quad n = 0, 1, 2 \ldots \end{aligned} \tag{2}$$

While the propinquity dynamics goes on, the topology difference, ΔT, between successive iterations will become relatively small compared with the whole graph. Similar case happens on the propinquity. In order to avoid the recalculation of all the propinquity values in each iteration, we can incrementally update the existing propinquity calculated in the previous iteration by comparing the current topology with the previous one. Then the propinquity dynamics in Equation 2 can be rewritten in its incremental form:

$$\begin{aligned} \Delta T_{n+1} &\leftarrow \quad \mathbb{C}^I_{\alpha,\beta}(T_n, P_n) \\ \Delta P_{n+1} &\leftarrow \quad \mathbb{P}^I(T_n, \Delta T_{n+1}) \\ T_{n+1} &\leftarrow \quad T_n + \Delta T_{n+1} \\ P_{n+1} &\leftarrow \quad P_n + \Delta P_{n+1} \quad , n = 0, 1, 2 \ldots \end{aligned} \tag{3}$$

Different from the non-incremental one, at the beginning of an incremental iteration besides the current topology T_n, we have the propinquity P_n corresponding to it. Their initial values come from $T_0 \leftarrow G$ and $P_0 \leftarrow \mathbb{P}(T_0)$. By contradicting them($\mathbb{C}^I_{\alpha,\beta}(T_n, P_n)$), the updated part of the new topology(ΔT_{n+1}) can be generated. Then the propinquity update(ΔP_{n+1}) can be calculated purely from old topology(T_n) and the incremental part(ΔT_{n+1}). This is the most computationally significant step therefore we will elaborate it in a separate section. The last two steps are easily comprehensive without explanation. Note that, we must assume $\Delta P \propto \Delta T$, which is the prerequisite of the efficiency of incremental propinquity update over the initial one, and depends on the concrete propinquity definition. Equation 3 is just a high level description. We will elaborate on the incremental propinquity update in Section 4, before that, we will first give a concrete propinquity definition in Section 3.3.

In both version of propinquity dynamics we assumed that the topology can converge, which means $\lim_{n\to\infty} \triangle T_n = \emptyset$. We have not yet proved the convergency of the dynamic propinquity and the necessary condition, however we did observe the convergency in all the real graph data we use.

3.2 Overlapping Community Extraction

Through the edge redistribution of the propinquity dynamics, the resulting topology is supposed to be clear with high density contrast. Then extracting community structures from it can be trivially done by discovering the connected components. Before that we have a chance to identify the community overlaps. Besides the topology, we have its corresponding propinquity as a byproduct, which can be reutilized to carry out a micro clustering on each vertex's neighbor vertices. The connected components extraction can still be done by breath-first search. The difference is that, the search cannot pass through the boundary between different micro clusters. Overly split micro clusters which are supposed to be in the same community will probably still be connected by micro clusters at other vertices. The robustness is based on the stable structure obtained from propinquity dynamics.

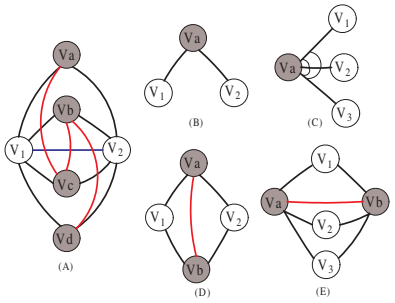

Figure 1: Coherent neighborhood propinquity and its calculation.

3.3 Coherent Neighborhood Propinquity

Now that we have the propinquity dynamics framework, it is time to specifically define the propinquity. Under the framework, the propinquity definition must possess several properties. First, the propinquity should reflect the probability that a pair of vertices are involved in a same community structure. Second, it can be efficiently calculated purely from topology. In addition, the definition should facilitate the incremental update. As mentioned in Section 1, the quasi-clique guarantees both density and balance of edge distribution, therefore can act as a hint for defining the community structure. In [20], a *gamma*-quasi-clique(where $\gamma \geq 0.5$) is said to be *coherent*. In [15], the author proved that the diameter of a coherent graph is no greater than 2. If we assume the resulting community structures are coherent, the topological distance for each pair of vertices in a community should be within 2. It suggests that the propinquity definition should consider only the local neighborhood within two hops. The propinquity between a pair of vertices with distance great than 2 should be zero. Then we can use the number of common neighbor vertices as part of the propinquity evaluation. Comparing back with the coherent graph, sharing large number of common neighbor does not guarantee the balance of the local neighborhood as a community. Therefore, the overall connectivity of the local neighborhood must be taken into consideration for propinquity evaluation. Let $N_G(v)$ represent the neighbor vertex set of vertex v in G. The subscript G will be omitted if there is no danger of confusion. Let $G[S]$ represent the induced subgraph of vertex set $S \subset V(G)$ with respect to G. Then the propinquity between $v_1, v_2 \in V(G)$ can be defined by Equation 4. The first sum term is trivially the number of direct connecting edges. The second term is the common neighbor vertices count. The last part is the number of edges connecting common neighboring vertices of v_1 and v_2. We name the edge

set in the third part the *conjugate* between v_1 and v_2.

$$P_G(v_1, v_2) \equiv |E(v_1, v_2)| + |N_G(v_1) \sqcap N_G(v_2)| + |E(G[N_G(v_1) \sqcap N_G(v_2)])| \quad (4)$$

This propinquity definition reflects the connectivity of the maximum coherent subgraph involving a vertex pair to a certain degree, therefore we term it as *coherent neighborhood propinquity*. Figure 1(A) shows an example of the coherent neighborhood between v_1 and v_2. The four shaded circles are common neighbors contributing 4 to the coherent neighborhood propinquity. The three edges in red are conjugates which contribute 3. Added up with the direct connection, the total propinquity between v_1 and v_2 is 8. The locality of the coherent neighborhood propinquity restricts the propinquity calculation and updates in a controllable scope.

3.4 Propinquity Calculation

A straightforward way to calculate all the neighborhood propinquity value in a graph is by the literal definition. That is, for each pair of vertices, intersecting their neighbor vertex set to get the common neighboring set, then counting the edges connecting the common neighbor vertices. The complexity of this naïve calculation is approximately $O((|V|+|E|)\cdot|E|)$, which is unacceptable for a large graph with millions of edges(e.g., Wikipedia linkage graph). Note that in the complexity analysis, we use $|E|/|V|$ to approximate each vertex degree. However in a real network the distribution of the vertex degree is highly skewed with a long tail in most of the cases, which makes the naïve computation even more inefficient. So we need a better one. In Figure 1(B), v_a is a common neighbor of v_1 and v_2, therefore can be gotten by set intersection according to propinquity definition. From the view point of v_a, it contributes a *propinquity unit* to v_1 and v_2. Actually, v_a contributes propinquity unit to each pair of its neighbor vertices, if $|N(v_a)| \geq 2$. As in Figure 1(C), v_a contributes propinquity to three pair of vertices. This type of propinquity components are produced intuitively by an angle in graph, therefore we name it as *angle propinquity*. The angle propinquity calculation can be formulated as: $\forall v \in V(G), \forall v_i, v_j \in N(v)(i \neq j), P(v_i, v_j) \leftarrow P(v_i, v_j) + 1$. In Figure 1(D), (v_a, v_b) is a conjugate of v_1 and v_2, therefore can be directly calculated according the propinquity definition. From the view point of conjugate edge (v_a, v_b), it contributes propinquity unit to v_1 and v_2. Actually, it contributes to each pair of common neighbors of v_a and v_b, e.g., the three pairs formed by v_1, v_2 and v_3 in the Figure 1(E). This type of propinquity components come from the conjugate edges, therefore we name it as *conjugate propinquity*. The conjugate propinquity calculation can be formulated as: $\forall (v_s, v_d) \in E(G), \forall v_i, v_j \in N(v_s) \sqcap N(v_d)(i \neq j), P(v_i, v_j) \leftarrow P(v_i, v_j) + 1$. The advantage of the later algorithm is that the computation can be done in a manner of for-each-vertex and for-each-edge, while the former naïve algorithm is in a for-each-vertex-pair manner. The overall complexity is therefore reduced to $O((|V| + |E|) \cdot |E|/|V|)$. For sparse networks, the complexity is simply $O(|V|)$.

4. INCREMENTAL PROPINQUITY UPDATE

In the previous section, we described the incremental propinquity dynamics framework from a global view. Now that we also have the concrete coherent neighborhood propinquity definition and its efficient calculation algorithm, in this section we will focus on how to incrementally update the propinquity in order to reflect the graph topology updates.

The topology update involves many edge deleting and inserting. Let us first consider the simplest case of single edge update. If a single edge is deleted from a graph, most of the propinquity values will remain unchanged, and we only need to update limited number of propinquity around the neighborhood. Assume the edge to be deleted is (v_1, v_2). The angle propinquity involving (v_1, v_2) as an angle edge should be reduced. That is, for each $v' \in N(v_1) - \{v_2\}$, $P(v_2, v')$ should be reduced by one propinquity unit. Symmetrically, for each $v' \in N(v_2) - \{v_1\}$, $P(v_1, v')$ should also be reduced by one unit. As for the conjugate propinquity update brought by single deleted edge, it is just an inverse process of its initial calculation. The single edge inserting case can be easily derived in a similar way. While all the deleting and inserting are taken into consideration as a whole, it is a little more complex than the single edge case to conduct efficient propinquity update.

Let $N_n(v)$ be the neighboring vertex set of v in topology T_n. The local neighboring topology update of v in the n-th iteration can be represented as Equation 5a, where $N_n^I(v)$ represents the set of vertices with which connection will be established, and $N_n^D(v)$ is the vertices from which vertex v will disconnect.

$$N_{n+1}(v) = N_n(v) + N_n^I(v) - N_n^D(v) \quad (5a)$$

$$N_n^R(v) = N_n(v) - N_n^D(v) \quad (5b)$$

Let $N_n^R(v)$ be the set of neighbors remaining unchanged after the $(n+1)$-th iteration, as shown in Equation 5b. We will use $N_n^R(v)$,$N_n^I(v)$ and $N_n^D(v)$ at each local vertex to integrally express $N_n(v)$ and $\Delta N_{n+1}(v)$. All the following formulations will be mapped to operations on the three set. The subscript n or target vertex v will be dropped for simplicity if the context allows.

In order to facilitate the description, we need to introduce several convenient symbolic shortcuts. Given two disjoint vertex sets, $S_1, S_2 (S_1 \sqcap S_2 = \emptyset)$, we use $S_1 \bowtie_+ S_2$ to represent the operation that, for each $v_i \in S_1$ and each $v_j \in S_2$, increase $P(v_i, v_j)$ by a unit propinquity. Accordingly we define the binary operator $\bowtie_-$ as the pairwise unit propinquity decreasing on the cartesian product of two operand vertex sets. In the case of self-join, two unary operators, $(\cdot)^{\bowtie_+}$ and $(\cdot)^{\bowtie_-}$, will be used. $(S)^{\bowtie_+}$ means, for each $v_i, v_j \in S (v_i \neq v_j)$, add a unit propinquity to $P(v_1, v_2)$. The $(S)^{\bowtie_-}$ can be easily comprehended.

$$\begin{aligned}
& (N_{n+1})^{\bowtie_+} - (N_n)^{\bowtie_+} \\
= \; & (N_n^R + N_n^I)^{\bowtie_+} - (N_n^R + N_n^D)^{\bowtie_+} && (6a) \\
= \; & (N_n^R)^{\bowtie_+} + (N_n^I)^{\bowtie_+} + N_n^R \bowtie_+ N_n^I \\
& -(N_n^R)^{\bowtie_+} - (N_n^D)^{\bowtie_+} - N_n^R \bowtie_+ N_n^D && (6b) \\
= \; & (N_n^I)^{\bowtie_+} + N_n^R \bowtie_+ N_n^I \\
& -(N_n^D)^{\bowtie_+} - N_n^R \bowtie_+ N_n^D && (6c) \\
= \; & (N_n^I)^{\bowtie_+} + N_n^R \bowtie_+ N_n^I \\
& +(N_n^D)^{\bowtie_-} + N_n^R \bowtie_- N_n^D && (6d)
\end{aligned}$$

Recall that the original angle propinquity can be calculated by $N(v)^{\bowtie_+}$ for each $v \in V(G)$. Let N_{n+1} and N_n denote the neighboring vertices of the target vertex in two consecutive iterations. Then the incremental angle propinquity update process can be deduced from Equation 6. Note that, each operand in Equation 6 represents a set of unit propin-

quity updates. Equation 6a can be deduced from Equation 5. Given two disjoint sets, S_1 and S_2, according to the definition of cartesian product, we have $(S_1 + S_2)^{\bowtie_+} = S_1^{\bowtie_+} + S_2^{\bowtie_+} + S_1 \bowtie_+ S_2$. So we have Equation 6b. Two negatives make a positive, that is, $-S^{\bowtie_-} = S^{\bowtie_+}$ and $-(S_1 \bowtie_- S_2) = S_1 \bowtie_+ S_2$, so we can have Equation 6d.

$$\begin{aligned} & N_n^C(v_1, v_2)^{\bowtie_-} \\ = \; & (N_n(v_1) \sqcap N_n(v_2))^{\bowtie_-} \\ = \; & [(N_n^R(v_1) + N_n^D(v_1)) \sqcap (N_n^R(v_2) + N_n^D(v_2))]^{\bowtie_-} \end{aligned} \quad (7)$$

Incremental conjugate propinquity update seems rather simple. If a graph edge, say (v_1, v_2), is cut in a topology update, we only need to erase the conjugate propinquity it contributed in the previous iteration. The erasing operation can be done with Equation 7. In the case of edge emerging, a complemental formulation can be easily derived as in Equation 8.

$$\begin{aligned} & N_{n+1}^C(v_1, v_2)^{\bowtie_+} \\ = \; & (N_{n+1}(v_1) \sqcap N_{n+1}(v_2))^{\bowtie_+} \\ = \; & [(N_n^R(v_1) + N_n^I(v_1)) \sqcap (N_n^R(v_2) + N_n^I(v_2))]^{\bowtie_+} \end{aligned} \quad (8)$$

So far, it seems that we have covered all the propinquity update cases, including angle and conjugate propinquity, edge inserting and deleting. However there is another case that is easily neglected. An edge (v_1, v_2) may remain after the topology update, but the conjugate propinquity it contributed in the previous iteration may partly change because of the coherent neighborhood update involving v_1 and v_2. We can analyze this case by analogy of the incremental angle propinquity update. Similar to $N_n^R(v)$, $N_n^{C,R}(v_1, v_2)$ can be defined as the set of common neighbor vertices of v_1 and v_2 that remain unchanged from the n-th algorithm iteration to the $(n+1)$-th iteration. $N_n^{C,I}(v_1, v_2)$ and $N_n^{C,D}(v_1, v_2)$ are also defined analogically from $N_n^I(v)$ and $N_n^D(v)$ respectively. This part of conjugate propinquity update can be calculate by Equation 9, which is derived in the same way as in Equation 6.

$$\begin{aligned} & N_{n+1}^C(v_1, v_2)^{\bowtie_+} - N_n^C(v_1, v_2)^{\bowtie_+} \\ = \; & N_n^{C,I}(v_1, v_2)^{\bowtie_+} + N_n^{C,R}(v_1, v_2) \bowtie_+ N_n^{C,I}(v_1, v_2) \\ & + N_n^{C,D}(v_1, v_2)^{\bowtie_-} \\ & + N_n^{C,R}(v_1, v_2) \bowtie_- N_n^{C,D}(v_1, v_2) \end{aligned} \quad (9)$$

The current problem with R-conjugate update is how to calculate the three parts of common neighbor vertices, that is $N_n^{C,R}(v_1, v_2)$, $N_n^{C,I}(v_1, v_2)$ and $N_n^{C,D}(v_1, v_2)$. We can derive $N_n^{C,R}(v_1, v_2)$ by Equation 10. Equation 10a and 10b are derived by corresponding definitions. Equation 10c is derived by Equation 5. From the disjointedness between $N^R(v)$,$N^I(v)$ and $N^D(v)$, 10c can be simplified to 10d by polynomial reduction.

$$\begin{aligned} & N_n^{C,R}(v_1, v_2) \\ = \; & N_{n+1}^C(v_1, v_2) \sqcap N_n^C(v_1, v_2) && \text{(10a)} \\ = \; & [N_{n+1}(v_1) \sqcap N_{n+1}(v_2)] \sqcap [N_n(v_1) \sqcap N_n(v_2)] && \text{(10b)} \\ = \; & [N_n^R(v_1) + N_n^I(v_1)] \sqcap [N_n^R(v_2) + N_n^I(v_2)] \\ & \sqcap [N_n^R(v_1) + N_n^D(v_1)] \sqcap [N_n^R(v_2) + N_n^D(v_2)] && \text{(10c)} \\ = \; & N_n^R(v_1) \sqcap N_n^R(v_2) && \text{(10d)} \end{aligned}$$

With the help of Equation 10, the newly emerging part of the common neighbor vertices set between v_1 and v_2 can be deduced in Equation 11.

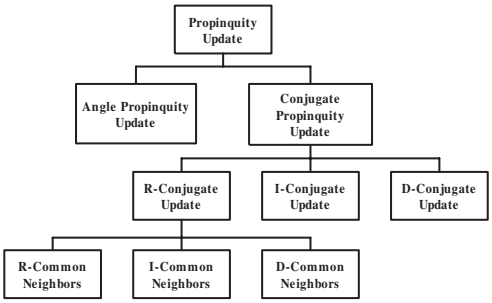

Figure 2: Incremental propinquity update.

$$\begin{aligned} & N_n^{C,I}(v_1, v_2) \\ = \; & N_{n+1}^C(v_1, v_2) - N_n^{C,R}(v_1, v_2) \\ = \; & [N_n^R(v_1) + N_n^I(v_1)] \sqcap [N_n^R(v_2) + N_n^I(v_2)] \\ & - N_n^R(v_1) \sqcap N_n^R(v_2) \\ = \; & N_n^R(v_1) \sqcap N_n^I(v_2) + N_n^I(v_1) \sqcap N_n^R(v_2) \\ & + N_n^I(v_1) \sqcap N_n^I(v_2) \end{aligned} \quad (11)$$

In a similar way, the disappeared part of the common neighbor vertices set between v_1 and v_2 can be deduced in Equation 12.

$$\begin{aligned} & N_n^{C,D}(v_1, v_2) \\ = \; & N_n^C(v_1, v_2) - N_n^{C,R}(v_1, v_2) \\ = \; & [N_n^R(v_1) + N_n^D(v_1)] \sqcap [N_n^R(v_2) + N_n^D(v_2)] \\ & - N_n^R(v_1) \sqcap N_n^R(v_2) \\ = \; & N_n^R(v_1) \sqcap N_n^D(v_2) + N_n^D(v_1) \sqcap N_n^R(v_2) \\ & + N_n^D(v_1) \sqcap N_n^D(v_2) \end{aligned} \quad (12)$$

We put the overall incremental propinquity update process in Figure 2. The update can be divided into two cases of angle and conjugate propinquity, as the coherent neighborhood propinquity was defined. The angle propinquity update is relatively simpler and can be calculated with Equation 6. The conjugate propinquity update is divided into three subcases. The conjugate propinquity update brought by cut and emerged edges are shown in Equations 7 and 8 respectively(*I-Conjugate Update* and *D-Conjugate Update* in Figure 2). The third part of conjugate propinquity update(*R-Conjugate Update* in Figure 2, Equation 9) is somewhat analogy to the angle propinquity. The three elements of Equation 9 should be further calculated with Equations 10, 11 and 12. Note that, all the operations are mapped to set intersection($\sqcap$), concatenation($+$) and cartesian product($\bowtie_+$ or $\bowtie_-$) operations on three simple disjoint set, that is $N^R(v)$, $N^I(v)$ and $N^D(v)$. In other words, the entire algorithm can be accomplished by a limited number of simple set operations.

4.1 Completeness

The correctness of the incremental angle propinquity update is obvious by the deduction in Equation 6. Now we focus on the conjugate propinquity. Let $\Theta_G(E)$ denote the conjugate propinquity value set calculated from each edge in $E(G)$, that is:

$$\Theta_G(E) \equiv \sum_{(v_i, v_j) \in E(G)} N^C(v_i, v_j)^{\bowtie_+} \quad (13)$$

We define E^R, E^I and E^D by E_{n+1}=$E_n + E^I - E^D$ and $N^R = E_n + E^D$. Then the overall conjugate propinquity

difference between two successive iterations can be deduced by Equation 14.

$$\begin{aligned} & \Theta_{T_{n+1}}(E_{n+1}) - \Theta_{T_n}(E_n) \\ = \; & \Theta_{T_{n+1}}(E^R + E^I) - \Theta_{T_n}(E^R + E^D) \\ = \; & [\Theta_{T_{n+1}}(N^R) - \Theta_{T_n}(N^R)] \\ & + \Theta_{T_{n+1}}(N^I) - \Theta_{T_n}(N^D) \end{aligned} \quad (14)$$

By expanding the three parts of Equation 14 with Equation 13, we can have precisely the first lines of Equations 9, 8, and 7 as the sum terms. Adding up the naïve direct connection propinquity, by now we have successfully proved the completeness of incremental propinquity update.

5. PARALLEL COMMUNITY DETECTION FROM LARGE DENSE NETWORKS

Without the assumption of graph sparsity, the complexity of propinquity dynamics is actually $O((|V|+|E|)(|E|/|V|)^2)$, where we use the average to approximate each vertex degree. However real networks are always dense, and the degree distribution is highly skewed. The Wikipedia linkage graph data we adopted in the experiment is a typical dense and skewed one. Besides, the intermediate propinquity values on dense graphs can be very large, and requires large memory space. All these facts make community structure discovery from such large, dense and skewed graph data a challenging task. Therefore, we implemented our algorithm under a parallel framework, and expect that the work load can be distributed on thousands of machines.

5.1 Parallel Model

The most intuitive way to parallelize a graph algorithm is by data parallelism, in which all vertices are evenly assigned onto multiple machines. Our model basically belongs to this class. Each vertex is regarded as a virtual process, and keeps its own programming logic and local data structures. A physical machine sequentially *executes* the vertices assigned to it by calling their logic and accessing their local data structures. Under this parallel framework we can implement the propinquity dynamics from the vertices' perspective. The vertices(virtual processes) have no idea about their physical location, and can exchange information from each other purely by message passing.

The propinquity dynamics is an iterative process and requires all vertices keeping synchronous from each other. So we also adopt the *bulk synchronous parallel*(BSP) model[19], which perfectly satisfies this requirement. In the BSP model, the overall computation proceeds in consecutive *supersteps*. Each participating *processor* can do three kinds of jobs in a single superstep: (1)Accessing the messages sent to it in the previous superstep; (2)Carrying out local computation and accessing local memory; (3)Send other processors messages, which will be available to the destination processor later in the next superstep. There is a *barrier* between two successive supersteps for the synchronization purpose. The barrier is also beneficial to the fault recovery. Combining the virtual-process technique and the BSP model by regarding graph vertices as the processors of the BSP model, we can have our parallel framework. The physical machines will execute each contained vertices once in each BSP superstep.

5.2 Parallel Implementation

Since we have the vertex oriented parallel framework, in this section we will describe the implementation of our algorithm from the vertex's perspective. Each vertex uses three vertex id sets: N^R, N^I and N^D to record its neighboring topology and topology update. In Section 4, we have mapped the entire incremental propinquity update process to operations on these three sets. As for the propinquity, each vertex keeps a hash map structure P for recording the pairwise propinquity values, which involve itself and are greater than 0. Note that each propinquity map entry is double duplicated at another vertex.

Besides the local data structures, we have to define two main types of messages before hand. The first message tells the receiving vertex to update its local propinquity map:

$$PU(v_d, V_u, +/-)$$

where v_d is the destination vertex id, and V_u is a vertex id set. For each vertex $v_i \in V_u$, the receiving vertex should increase/decrease the propinquity map entry $P[v_i]$ by one unit. The second message is used to send parts of the local neighbor vertex sets to another vertex:

$$DN(v_d, S^R, S^I, S^D)$$

where S^R,S^I and S^D are used to carry N^R, N^I and N^D respectively, all the three can be null. At the very beginning, a vertex, say v_s, has N^R storing its initial neighbor vertex set. Before stepping into the cruising incremental propinquity dynamic iterations, the propinquity map P according to the current topology should be first constructed by Algorithm 1. After that each vertex can run into the logic in Algorithm 2, where the input and output are both N^R and P therefore can be executed iteratively. In Algorithm 2 we omit the propinquity update message processing logic as it is exactly the same as in Algorithm 1. The $h(\cdot)$ in the pseudo code is a function that helps decide which vertex should initiate the neighborhood donation. For the sake of compactness, we put the lengthy set intersection and concatenation operation involving neighboring sets from different vertices in Table 1. In the pseudo code we use C^{XY} to refer the resulting set calculated from operation at X column and Y row in Table 1, where subscripts 1 and 2 are used to differentiate the source of the neighborhood sets. Besides, the original formulation numbers are marked next to the formula in Table 1.

Input: N^R
Output: P
Superstep 0:
  for each $v_i \in N^R$
    $PU(v_i, N^R - \{v_i\}, +)$
Superstep 1:
  for each received S^R, **for each** $v_i \in S^R$
    $P[v_i] \leftarrow P[v_i] + 1$
  for each $v_i \in N^R$
    if h(v_i)>h(v_s)
      $DN(v_i, N^R - \{v_i\}, \text{null}, \text{null})$
Superstep 2:
  for each received S^R
    $N^C \leftarrow N^R \sqcap S^R$
    for each $v_i \in N^C$
      $PU(v_i, N^C - \{v_i\}, +)$
Superstep 3:
  for each received S^R, **for each** $v_i \in S^R$
    $P[v_i] \leftarrow P[v_i] + 1$

Algorithm 1: Initial propinquity calculation

	R	I	D
R	$N_1^R \sqcap N_2^R$ **[10]**	na	na
I	$(N_1^R \sqcap N_2^I)+(N_1^I \sqcap N_2^R)$ $+(N_1^I \sqcap N_2^I)$ **[11]**	$(N_1^R + N_1^I) \sqcap$ $(N_2^R + N_2^I)$ **[8]**	na
D	$(N_1^R \sqcap N_2^D)+(N_1^D \sqcap N_2^R)$ $+(N_1^D \sqcap N_2^D)$ **[12]**	na	$(N_1^R + N_1^D) \sqcap$ $(N_2^R + N_2^D)$ **[7]**

Table 1: Intermediate pairwise neighborhood

Input: N^R,P,α,β
Output: N^R,P
Superstep 0:
  $N^I \leftarrow \emptyset; N^D \leftarrow \emptyset$
  for each $v_i \in P.keys$
    if $P[v_i] \leq \alpha$ and $v_i \in N^R$
      $N^D \leftarrow N^D + \{v_i\}$, $N^R \leftarrow N^R - \{v_i\}$
    if $P[v_i] \geq \beta$ and $v_i \notin N^R$
      $N^I \leftarrow N^I + \{v_i\}$
  for each $v_i \in N^R$
    $PU(v_i,N^I,+)$, $PU(v_i,N^D,-)$
  for each $v_i \in N^I$
    $PU(v_i,N^R,+)$, $PU(v_i,N^I - \{v_i\},+)$
  for each $v_i \in N^D$
    $PU(v_i,N^R,-)$, $PU(v_i,N^D - \{v_i\},-)$
Superstep 1:
  Update P with received angle propinquity update.
  for each $v_i \in N^R$
    if $h(v_i) > h(v_s)$, $DN(v_i,N^R,N^I,N^D)$
  for each $v_i \in N^I$
    if $h(v_i) > h(v_s)$, $DN(v_i,N^R,N^I,$null$)$
  for each $v_i \in N^D$
    if $h(v_i) > h(v_s)$, $DN(v_i,N^R,$null$,N^D)$
Superstep 2:
  for each received DN message from v_f
    if $v_f \in N^R$
      Calculate C^{RR}, C^{RI} and C^{RD} according to Table 1
      for each $v_i \in C^{RR}$
        $UP(v_i,C^{RI},+)$, $UP(v_i,C^{RD},-)$
      for each $v_i \in C^{RI}$
        $UP(v_i,C^{RR},+)$, $UP(v_i,C^{RI} - \{v_i\},+)$
      for each $v_i \in C^{RD}$
        $UP(v_i,C^{RR},-)$, $UP(v_i,C^{RD} - \{v_i\},-)$
    if $v_f \in N^I$
      Calculate C^{II} according to Table 1
      for each $v_i \in C^{II}$
        $UP(v_i,C^{II} - \{v_i\},+)$
    if $v_f \in N^D$
      Calculate C^{DD} according to Table 1
      for each $v_i \in C^{DD}$
        $UP(v_i,C^{DD} - \{v_i\},-)$
Superstep 3:
  Update P with received conjugate propinquity update.
  $N^R \leftarrow N^R + N^D$

Algorithm 2: Incremental propinquity update

5.3 Performance Issues

While mining very large network data, we found that the overall messages sent in a single superstep can easily exceed the memory quota. In order to guarantee the memory residence rather than relying on I/O, in real implementation, one single supersteps in the pseudo code description is actually broken into multiple supersteps. In the pseudo code, each message sending statement is preceded by a *for-each*, therefore bookmark variables can be introduced to resume the message sending progress in the next superstep. The question is how the vertices decide when to cease message sending, which directly influences the load balance. After tried several message bounding schemes, we adopt a message size estimate technique, in which before a macro superstep begins, all vertices collaboratively estimate the overall message size. Then by comparing with the physically available memory, a least required number of supersteps can be calculated. Each vertex can then break its local message into this number of parts. In this way, the overall load balance can be guaranteed.

Another performance improvement we made is to buffer the propinquity messages with a second map. Before leaving the iteration in Algorithm 2, each vertex merges the two propinquity maps. While the update part of the propinquity value list is relatively small, the buffering technique can save many map search operations on the original propinquity map.

6. EXPERIMENTS

In order to evaluate the efficiency and scalability of our algorithm on large graphs we mainly used the Wikipedia linkage graph dataset dumped on July 24, 2008. In this graph, vertices are normal wikipedia pages, and hyperlinks in Wikipedia pages pointing to other normal Wikipedia pages are extracted as undirected graph edges. There are some *redirect* pages that will be redirected to another normal page if accessed, e.g. *Mongol* is a redirect page pointing to *Mongols*. We utilized several passes of MapReduce to preprocess and clear the graph data including merging the redirect pages and removing self loop and multi edges. The statistics of the final Wikipedia linkage graph are shown in Table 2.

dataset	Wikipedia	eatRS	Erdos02	hep-th-new
# of vertices	2,491,887	23,219	6,927	27,770
# of edges	117,714,397	325,028	11,850	352,768
Avg. degree	94.48	28.00	1.71	12.70
Max. degree	409,511(in)	1,106	507	2,468
direction	Y	Y	N	Y

Table 2: Statistics of the graph datasets.

We also used other three neutral-sized graph datasets. Edinburgh Associative Thesaurus(EAT) is a word association network, which is composed by a large number of word norm stimulus and responds. Erdos02 is a co-authorship network, in which Paul Erdos is a prolific mathematician lying at the core of the entire network. hep-th-new is a citation graph data from KDD Cup 2003. It contains the citation tree of many articles related to the topic of high energy physics(hep) since 1974. The statistics of these three datasets are also in shown Table 2.

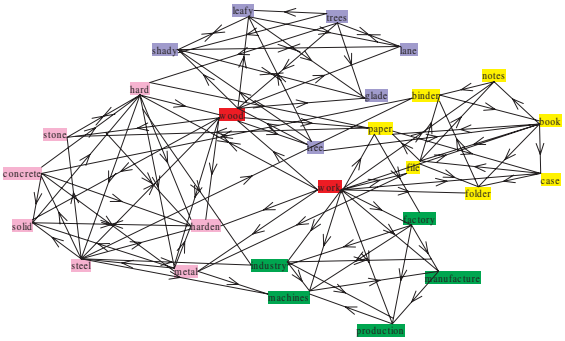

Figure 3: Overlapping community structures mined from word association network eatRS.

We first tested effectiveness of the propinquity dynamics based community detection on the neutral-sized eatRS data. Figure 3 shows part of the discovered community structures, which are differentiated by colors. The blue, pink, yellow, and green communities contain words related to forest landscape, stiff material, stationery, and manufacturing respectively. Besides, the red nodes represent the community overlapping, which may mean polysemous words, e.g., *wood* is a kind of hard material(pink) and is also essential as part of the forest(blue). As another overlapping example, the factory(green) and stationery(yellow) are both related with *work*. This experiment was done with α=5 and β=180 on 10 machines(1G CPU, 1G Mem)[1]. The computation can be finished by 2 iterations and 363 supersteps within 2 minutes.

Running on the Erdos02 co-authorship network, our algorithm also generates interesting results as partly shown in Figure 4. It contains the projection of 5 selected resulting community on the entire network. The communities and overlaps are also marked with different colors. Famous Paul Erdos is certainly the most obvious community overlap. Besides, there is another productive mathematician, Tarlok N. Shorey, in Figure 4 who has collaborated with two loosely interconnected groups of mathematicians, therefore can been seen as a community overlap. This experiment was done with α=2 and β=20 on 10 machines(1G CPU, 1G Mem). The computation was complete by 2 iterations and 169 supersteps within 1 minute. Through these two experimental results we can see that the propinquity dynamics can effectively identify the community structures and the overlapping vertices between them.

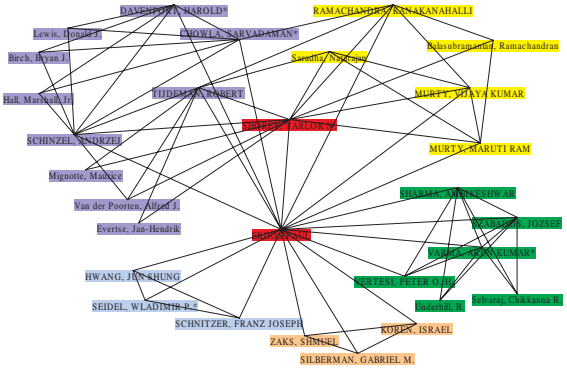

Figure 4: Overlapping community structures mined from the Erdos02 co-authorship network.

Compared with the previous two neutral-sized graph data, detecting community structures from the original Wikipedia linkage graph is rather challenging. The difficulty comes from the skewness of the vertex degree distribution. For example, the page of *United States* is pointed by 40,9511 other pages, and the page with title *List of endangered animal species* contains hyperlinks pointing to other 5,549 pages. The running time for a single vertex depends on the square of its degree, therefore the overall efficiency can be undermined by these abnormal vertices. By the principle of TF-IDF, vertices with very high degree contribute few information for the community detection. So we filter out part of edges by setting an upper bound on the in-degree and out-degree. We used 300 as a typical degree bound for guaranteeing both the community quality and running efficiency. By setting α=400 and β=1000, we obtained the community structures hidden in the Wikipedia. Like the skewness of the degree distribution, the community structure size distribution is also unbalanced. Table 3 lists three neutral-size ones.

Community 1	Community 2	Community 3
Pretty Good Privacy	Electrical impedance	Agriculture
GNU Privacy Guard	Impedance matching	Organic farming
Digital signature	Impedance bridging	Agronomy
Public key certificate	Damping factor	Sustainable agriculture
Web of trust	Input impedance	Weed control
Key signing party	Output impedance	Cultivation
Keysigning		

Table 3: Selected community structures mined from Wikipedia linkage graph.

Our community detection algorithm is implemented on a parallel platform, so it is time to verify the efficiency of the parallelization. We will use two group of experiments to test the speedups on datasets with different scale. The first group of experiments were conducted on the neutral-sized hep-th-new graph, with parameter setting: α=20, β=300. The termination condition ϵ was set to 1000, which constrains the running to 5 iterations and 655 BSP supersteps. We allocated 1G CPU and 1G memory for each machine. From Figure 5 we can see that the speedups on dozens of machines are very impressive. The reason we did not use more machines is because of the neutral size of the dataset. Over partitioning the graph vertices on too many machines will bring unaffordable overload.

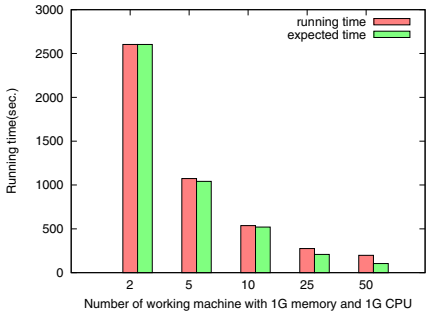

Figure 5: Speedups while running on neural-sized hep-th-new paper citation network.

Figure 6 presents the speedup results on large Wikipedia dataset while running on up to 1K machines. The parameter setting is: α=400, β=1000. For a reasonable waiting time, the running is constrained to the first 4 algorithmic iterations. Note that, when 62 and 125 machines are used, the memory quota has to be increased for the purpose that the overall memory space requirement can be satisfied.

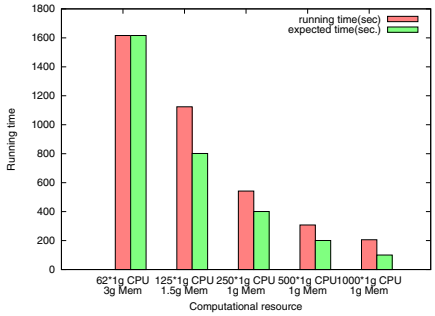

Figure 6: Speedups while running on large scale Wikipedia linkage graph.

[1]All our experiments are carried out on Google data centers, from which computational resources like CPU and memory can be accurately allocated.

In the final part, we will reveal the improvement brought by the incremental propinquity update technique. Under the configuration of α=400 and β=1000, the running time comparison between the incremental and non-incremental versions of propinquity calculation on 1000 standard machines with 1G CPU and 1G memory is drawn in Figure 7. Iteration 0 means the initial propinquity calculation from input graph topology. We can explain the efficiency improvement of incremental propinquity update over the non-incremental one by Figure 8, which presents the size evolving of the topology and propinquity maps. The size of N^I and N^D rapidly decreases while the iteration goes on, which is the precondition of the justification of our incremental technique. The size difference between propinquity map and the updated part proved the double propinquity map technique makes sense.

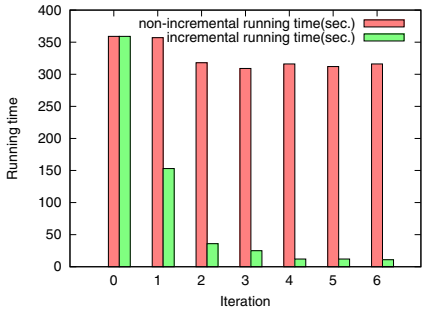

Figure 7: The effectiveness of the incremental propinquity update(Wikipedia linkage graph).

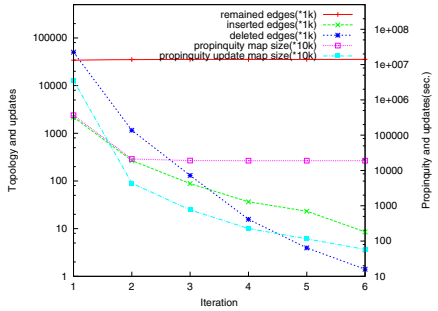

Figure 8: Topology and propinquity evolves with iteration.

7. CONCLUSION AND FUTURE WORK

In this paper, we proposed an intuitive community detection algorithm, which is based on the mutual update between the topology and propinquity. Under the definition of coherent neighborhood propinquity, a series of efficient calculation algorithms and incremental techniques are proposed. The entire algorithm was implemented under a virtual-vertex oriented bulk synchronous parallel framework, which can distribute the work load to thousands of machines. The experimental results show that the coherent neighborhood based propinquity dynamics can correctly identify overlapping community structures from real graph data, and the incremental propinquity update techniques are very effective. The speedups on various datasets and different numbers of machines are satisfactory.

In the future, we should prove the conditional convergency of propinquity dynamics. Besides, each vertex may be able to decide its local cutting and emerging threshold through referring its local neighborhood. Two threshold parameters control the granularity of the community detection, and may be changed in the middle of the propinquity dynamics for the purpose of exploring the community hierarchy.

8. REFERENCES

[1] S. Boccaletti, M. Ivanchenko, V. Latora, A. Pluchino, and A. Rapisarda. Detecting complex network modularity by dynamical clustering. *Phys Rev E Stat Nonlin Soft Matter Phys*, 75(4), 2007.

[2] A. Capocci, V. D. P. Servedio, G. Caldarelli, and F. Colaiori. Detecting communities in large networks. *Physica A: Statistical and Theoretical Physics*, 352(2-4):669–676, July 2005.

[3] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 70:066111, 2004.

[4] L. Donetti and M. A. Munoz. Detecting network communities: a new systematic and efficient algorithm. *Journal of Statistical Mechanics: Theory and Experiment*, 2004(10):P10012, 2004.

[5] J. Duch and A. Arenas. Community detection in complex networks using extremal optimization. *Physical Review E*, 72:027104, 2005.

[6] S. Fortunato and C. Castellano. Community structure in graphs. *Chapter of Springer's Encyclopedia of Complexity and System Science*, Dec 2007.

[7] S. Fortunato, V. Latora, and M. Marchiori. Method to find community structures based on information centrality. *Phys. Rev. E*, 70(5):056104, Nov 2004.

[8] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proc Natl Acad Sci U S A*, 99(12):7821–7826, June 2002.

[9] R. Guimerà, M. Sales-Pardo, and L. A. N. Amaral. Modularity from fluctuations in random graphs and complex networks. *Phys. Rev. E*, 70(2):025101, Aug 2004.

[10] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[11] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Phys Rev E Stat Nonlin Soft Matter Phys*, 69(6), 2004.

[12] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Phys Rev E Stat Nonlin Soft Matter Phys*, 74(3), 2006.

[13] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, 2004.

[14] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818.

[15] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *Proceedings of the eleventh ACM SIGKDD international conference on knowledge discovery in data mining*, pages 228–238, New York, NY, USA, 2005. ACM.

[16] P. Pons and M. Latapy. Computing communities in large networks using random walks. *Journal of Graph Algorithms and Applications*, 10(2):191–218, Dec 2006.

[17] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi. Defining and identifying communities in networks. In *Proceedings of the National Academy of Science of the United States of America*, volume 101-9, pages 2658–2663, 2004.

[18] J. Reichardt and S. Bornholdt. Statistical mechanics of community detection. *Phys Rev E Stat Nonlin Soft Matter Phys*, Mar 2006.

[19] Wikipedia. Bulk synchronous parallel — wikipedia, the free encyclopedia, 2008. [Online; accessed 23-December-2008].

[20] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Out-of-core coherent closed quasi-clique mining from large dense graph databases. *ACM Trans. Database Syst.*, 32(2):13, 2007.