2017 IEEE 37th International Conference on Distributed Computing Systems

Improving Efficiency of Link Clustering on Multi-Core Machines

Guanhua Yan
Binghamton University, State University of New York

***Abstract*—Link clustering groups different edges in a graph according to their similarities. Link clustering can reveal the overlapping and hierarchical organizations in a wide spectrum of networks. This work studies how to improve efficiency of link clustering along three dimensions, *algorithm*, *modeling*, and *parallelization*, on multi-core machines. We evaluate the efficiency improved due to each of the three dimensions using word association graphs extracted from a twitter dataset.**

***Keywords*-link clustering; algorithm; modeling; multi-core**

I. Introduction

The seminal paper by Ahn *et al.* has found that link clustering can be used to discover hierarchy and community organization in real-world networks, including social networks, biological networks, and word association networks [1]. However, the original work in [1] did not delve into the computational aspect of link clustering. Hence, in this work we explore multiple perspectives from which we can accelerate the link clustering algorithms on large graphs. More specifically, we pursue answers to the following key questions: Can we develop more efficient serial algorithm for link clustering? Can we leverage coarse-grained models to reduce the complexity of the problem itself? Can we parallelize the algorithm to run it on the multiple-core architecture, which is readily available on commodity multi-core machines? We propose solutions from three distinct perspectives, i.e., *algorithm*, *modeling*, and *parallelization*, and these methods complement each other and fit together into a unified, organic solution:

- **Algorithm:** We develop a new simple, yet efficient, single-linkage link clustering algorithm on graphs, and through rigorous analysis of its computational complexity, we demonstrate that our serial algorithm provides significant performance improvement.
- **Modeling:** We consider the cases where strict dendrograms from hierarchical link clustering are unnecessary. Under such an assumption, we focus on how to produce coarse-grained dendrograms where the rates at which clusters merge with each other from level to level do not exceed a certain threshold. We develop a novel method to predict the boundaries between different levels on the coarse-grained dendrogram.
- **Parallelization:** We parallelize our algorithm based on the multi-threading technology so as to take advantage of the multi-core architecture readily available on modern commodity PCs.

Our proposed solutions are evaluated based upon a corpus of tweets collected within a month. The experimental results reveal the following. (1) When the underneath graph is large, our proposed serial algorithm can outperform the standard algorithm by more than one order of magnitude. (2) The coarse-grained link clustering model not only incurs less computational overhead than the fine-grained one, but also allows us to distribute the computational workload over multiple cores. (3) Using multi-threading, our parallel algorithm takes advantage of multi-core architectures to achieve good strong-scaling. Our source code is available at http://www.cs.binghamton.edu/~ghyan/code/link-clustering.

II. Related Work

Graph clustering is a subfield of clustering, where the data points to be clustered are components of a graph. Due to its applications in community detection in networks, graph clustering has intensively investigated in the past. In [2], Schaeffer provides a comprehensive survey of existing graph clustering algorithms, including those that perform hierarchical clustering on graphs in a bottom-up fashion (i.e., agglomerative algorithms), such as [3]–[6]. The concept of coarse-grained hierarchical clustering considered in this study was however not considered in these efforts.

For the generic single-linkage hierarchical clustering problem, its *optimally efficient* solution has time complexity $O(n^2)$, where n is the number of data points to be clustered [7], [8]. The connection between the minimum spanning tree algorithm and single-linkage hierarchical clustering was explored in [9].

III. Problem Formulation

The link clustering algorithm proposed in [1] to cluster edges in a weighted graph $G(V,E)$ works as follows. The similarity between any two incident edges e_{ik} and e_{jk} is defined based on the Tanimoto coefficient:

$$S(e_{ik}, e_{jk}) = \frac{\mathbf{a}_i \cdot \mathbf{a}_j}{|\mathbf{a}_i|^2 + |\mathbf{a}_j|^2 - \mathbf{a}_i \cdot \mathbf{a}_j}, \tag{1}$$

where $\mathbf{a}_i = (\widetilde{A}_{i1}, ..., \widetilde{A}_{i|V|})$ corresponds to vertex X_{f_i} with:

$$\widetilde{A}_{ij} = \frac{1}{n_i} \sum_{i' \in N_i} w_{ii'} \delta(i=j) + w_{ij}, \tag{2}$$

where n_i and N_i denote the number of neighbors and the set of neighbors of vertex X_{f_i}, respectively. Note that the delta function $\delta(x)$ in Eq. (2) returns 1 if x is true or 0 otherwise. Hence, when $i \neq j$, $\widetilde{A}_{ij}$ is the weight on the edge if two vertices are adjacent, and 0 otherwise; when $i = j$, $\widetilde{A}_{ij}$ returns the average weight over all the edges between vertex X_{f_i} (or X_{f_j}) and its neighbors. The similarity between any two non-incident edges is defined to be always 0.

In [1], the single-linkage hierarchical clustering algorithm was used to cluster the edges. Single-linkage hierarchical clustering is the simplest type of hierarchical clustering, and its *optimally efficient* solution has time complexity $O(n^2)$ where n is the number of data points to be clustered [7]. For a big graph with a large number of edges (i.e., $|E|$ is large), it is still computationally prohibitive to apply these methods directly due to time complexity $O(|E|^2)$.

Example: word association network. Given a corpus of social media messages, $\mathcal{D} = \{d_1, d_2, ..., d_m\}$, we can construct a word association network from it. For instance, $\mathcal{D}$ can be the entire set of tweets published within a year, and each d_i corresponds to a single tweet. Moreover, each d_i is further comprised of words. The set of features of interest is represented by $\mathcal{F}$. An example of $\mathcal{F}$ can be the set of English words that are not stop words such as 'the' and 'a'. Correspondingly, we use a feature variable X_f for each feature $f \in \mathcal{F}$ to indicate that f appears ($X_f = 1$) or does not appear ($X_f = 0$) in a social media message. The set of all feature variables is denoted by $\mathcal{X}_{\mathcal{F}}$.

The word association network, which is represented as a weighted graph $G(V, E)$, is constructed as follows. Each node in graph G corresponds to a feature variable $X \in \mathcal{X}_{\mathcal{F}}$. An edge is added between two nodes corresponding to X_{f_i} and X_{f_j} if the following quantity is greater than 0:

$$w_{ij} = p(X_{f_i} = 1, X_{f_j} = 1) \log \frac{p(X_{f_i} = 1, X_{f_j} = 1)}{p(X_{f_i} = 1)p(X_{f_j} = 1)}. \quad (3)$$

The rationale behind this formula has its root from mutual information in information theory. If $w_{ij} > 0$, it means that the probability that both X_{f_i} and X_{f_j} appear in the same social media post (e.g., tweet) is higher than the probability that their appearances in a post are totally independent. Note that if the latter indeed occurs, we have $p(X_{f_i} = 1, X_{f_j} = 1) = p(X_{f_i} = 1)p(X_{f_j} = 1)$.

IV. Serial Algorithm

Our algorithm has two phases: *initialization* and *sweeping*.

A. Phase I: Initialization

The pseudo code of Phase I is given in Algorithm 1. During this phase, data structures are created that are amenable to hierarchical clustering later. According to Eq. (1), calculating the similarity between two incident edges involves the computation of the inner product of vectors $\mathbf{a}_i$ and $\mathbf{a}_j$, which takes $O(|V|)$ time. As similarity evaluation itself is a costly operation, we use an algorithm that traverses graph $G(V, E)$ three times to reduce the computational cost. During this phase, intermediate data structures (e.g., arrays H_1 and H_2) are used to avoid recalculation. (1) In the first pass (Lines 1-5), we populate two arrays H_1 and H_2, both of which have size $|V|$ and are initialized to contain all 0's. After the

Algorithm 1 Initialization phase of the serial algorithm

```
Require: Weighted undirected graph $G(V,E)$, where $V = \{v_i\}_{1 \le i \le |V|}$. The list of neighbors of vertex $v$ is denoted by $N(v)$.
 1: $H_1 \leftarrow \mathbf{0}$, $H_2 \leftarrow \mathbf{0}$
 2: for $1 \le i \le |V|$ do
 3:     $H_1[i] \leftarrow \frac{\sum_{v_j \in N(v_i)} w_{ij}}{|N(v_i)|}$
 4:     $H_2[i] \leftarrow H_1[i]^2 + \sum_{v_j \in N(v_i)} w_{ij}^2$
 5: end for
 6: $M \leftarrow \emptyset$                    ▷ Create an empty map $M$
 7: for $1 \le i \le |V|$ do
 8:     for $v_j \in N(v_i)$ do
 9:         for $v_k \in N(v_i)$ do
10:             if $v_k \le v_j$ then
11:                 Continue
12:             end if
13:             if $(v_j, v_k)$ is a key of map $M$ then
14:                 $M(v_j, v_k) \leftarrow (M(v_j, v_k)[1] + w_{ij}w_{ik}, M(v_j, v_k)[2] \cup \{v_i\})$
15:             else
16:                 $M(v_j, v_k) \leftarrow (w_{ij}w_{ik}, \{v_i\})$
17:             end if
18:         end for
19:     end for
20: end for
21: for edge $(v_i, v_j) \in E$ do
22:     if $(v_i, v_j)$ is a key of map $M$ then
23:         $M(v_i, v_j)[1] \leftarrow M(v_i, v_j)[1] + (H_1[i] + H_1[j])w_{ij}$
24:     end if
25: end for
26: for each key $(v_i, v_j)$ in map $M$ do
27:     $M(v_i, v_j)[1] \leftarrow \frac{M(v_i,v_j)[1]}{H_2[i]+H_2[j]-M(v_i,v_j)[1]}$
28: end for
```

first pass, $H_2[i]$ gives $|\mathbf{a}_i|^2$ in Eq. (1). (2) In the second pass (Lines 6-20), we aim to populate an empty map M. A key of map M is a vertex pair, and the corresponding value is a tuple, the first of which is the sum of weight products and the second a list of common neighbors shared by both vertices in the key. Here without loss of generality, for a tuple t, we use $t[1]$ to reference its first element, $t[2]$ its second one, and so on. (3) In the third pass (Lines 21-25), we aim to calculate, for every vertex pair (v_i, v_j) in map M, the inner product of $\mathbf{a}_i$ and $\mathbf{a}_j$ in Eq. (1), i.e., $\mathbf{a}_i \cdot \mathbf{a}_j$.

Finally, for each vertex pair (v_i, v_j) that is a key of map M, a similarity score is calculated and then stored as the first element of the corresponding value. Our key observation here is that in Eq. (1), the similarity between any two incident edges e_{ik} and e_{jk} depends on only vectors of vertices v_i and v_j (i.e., $\mathbf{a_i}$ and $\mathbf{a_j}$) but not that of the common neighbor v_k (i.e., $\mathbf{a_k}$).

B. Phase II: Sweeping

The pseudo code of the `sweeping` phase is given in Algorithm 2. In this phase, we first sort the vertex pairs in the keys of map M according to their associated similarity scores in a non-increasing order, and put the results on list L (Lines 1-5). For each element on list L, we also have a

Algorithm 2 Sweeping phase of the serial algorithm

```
Require: Map $M$ from Phase I.
 1: $L \leftarrow \emptyset$                                  ▷ Create an empty list $L$
 2: for each key $k$ in map $M$ do
 3:     Append $(M[k][1], k, M[k][2])$ to list $L$
 4: end for
 5: Sort list $L$ based upon the first elements of its items in non-
    increasing order
 6: $I \leftarrow \emptyset$                                  ▷ Create an empty map $I$
 7: for $1 \leq i \leq |E|$ do
 8:     $I[E_i] \leftarrow i$
 9: end for
10: $C \leftarrow \emptyset$                                  ▷ Create an empty array $C$
11: for $1 \leq i \leq |E|$ do
12:     $C[i] \leftarrow i$                       ▷ $C[i]$ is the cluster of edge $i$
13: end for
14: $r \leftarrow 0$                              ▷ $r$ is the current merging level
15: for each $x$ on sorted list $L$ do
16:     $(v_i, v_j) \leftarrow L[2]$                              ▷ Vertex pair
17:     $l \leftarrow L[3]$                           ▷ List of common neighbors
18:     for $v_k \in l$ do
19:         Call procedure MERGE$(I[(v_i, v_k)], I[(v_j, v_k)])$
20:     end for
21: end for
22:
23: procedure MERGE($i_1$, $i_2$)                     ▷ Merging two clusters
24:     $F_1 \leftarrow F(i_1)$, $F_2 \leftarrow F(i_2)$
25:     $c_1 \leftarrow \min\{F_1\}$, $c_2 \leftarrow \min\{F_2\}$, $c_{min} \leftarrow \min\{c_1, c_2\}$
26:     for $j \in F_1 \cup F_2$ do
27:         $C[j] \leftarrow c_{min}$
28:     end for
29:     if $c_1 \neq c_2$ then
30:         $r \leftarrow r + 1$                          ▷ Increase merging level $r$
31:         Output merging $c_1, c_2 \rightarrow c_{min}$ at level $r$
32:     end if
33: end procedure
34:
35: function F($i$)
36:     if $i = C[i]$ then
37:         Return $\{i\}$
38:     else
39:         Return $\{i\} \cup F(C[i])$
40:     end if
41: end function
```

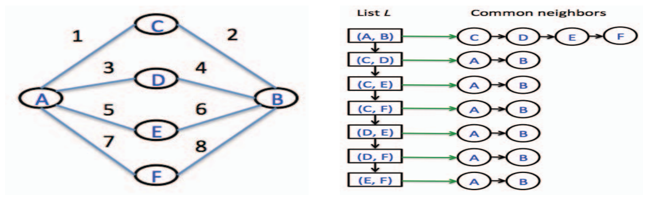
reference pointing to the list of neighbors they share from map M. List L created for an example graph is shown in Figure 1 (similarity scores omitted).

In addition to L, another important data structure used in the sweeping phase is array C. We enumerate the edges in G in a random order, and assign the order of each link in this permutation as its id. Let $I[e]$ map an edge e to its index (Lines 6-9). The array C is created of size $|E|$, with each $C[i]$, where $1 \leq i \leq |E|$, initialized to be i (Lines 10-13).

Function $F(i)$, where $i \in [1, |E|]$, is recursively defined:

$$F(i) = \left\{ \begin{array}{ll} \{i\} \cup F(C[i]) & i \neq C[i] \\ \{i\} & i = C[i] \end{array} \right. \quad (4)$$

Hence, $F(i)$ contains all the indices of edges on the chain from edge i to the one that points to itself in array C. We use counter r to keep the level at which a merging operation



(1) Example graph (2) Data structure

Figure 1. An example graph and the corresponding data structure

takes place. Counter r is initialized to be 0 (Line 14).

The remainder of Phase II is an iterative process until L becomes empty (Lines 15-21). In each iteration, we extract a vertex pair (v_i, v_j) from the head of L, along with the list of their common neighbors denoted as l. For every vertex v_k on list l, we apply a `MERGE` procedure to merge the two clusters that edges (v_i, v_k) and (v_j, v_k) belong to, and use its smallest index of the edges as the new cluster index.

More specifically, the `MERGE` procedure on two edge indices i_1 and i_2 works as follows (Lines 23-33). We obtain $F(i_1)$ and $F(i_2)$ according to Eq. (4), and then for every $j \in F(i_1) \cup F(i_2)$, we reset $C[j]$ to be $\min\{F(i_1) \cup F(i_2)\}$. If $\min\{F(i_1)\} \neq \min\{F(i_2)\}$, we increase counter r by 1 and then merge clusters indexed by $\min\{F(i_1)\}$ and $\min\{F(i_2)\}$ at level r of the dendrogram as follows:

$$r : \min\{F(i_1)\}, \min\{F(i_2)\} \rightarrow \min\{F(i_1) \cup F(i_2)\}. \quad (5)$$

C. Serial Algorithm Analysis

The correctness of the algorithm can be established by the following theorem (proof omitted):

Theorem 1: In the sweeping algorithm, $\min\{F(i)\}$ gives the right cluster id that edge i belongs to in array C.

We further analyze the complexity of the `sweeping` algorithm. We first define three properties of graph G:

Notation	Definition
K_1	Number of node pairs with at least a common neighbor
K_2	Number of pairs of incident edges in G
K_3	Number of pairs of distinct edges in G

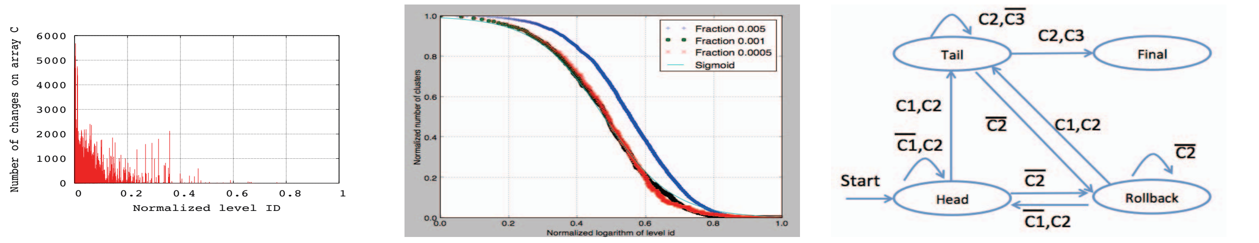
Clearly, for any graph, we have $K_1 \leq K_2 \leq K_3$, because there can be multiple edge pairs connecting two vertices of distance 2. For instance, in the graph shown in Figure 1, $K_1 = 7 < K_2 = 16 < K_3 = 28$.

The following theorem establishes the complexity of our serial algorithm (proof shown in the Appendix):

Theorem 2: Given graph $G(V, E)$ with K_2 pairs of distinct incident edges, the time and space complexity of our serial algorithm are $O(|V| + K_1 \log K_1 + \sqrt{K_2}|E|)$ and $O(K_2 + |E|)$, respectively.

V. Coarse-grained clustering

To speed up link clustering, we consider coarse-grained clustering which allows multiple clusters to merge at each level. We slightly change the algorithm in Section IV as follows. We first divide list L into m chunks $\widehat{L}_1, \widehat{L}_2, ..., \widehat{L}_m$, using a scheme described later. *Within each chunk, all vertex*



(1) Changes on array C (2) Model fitting (3) Mode transition machine

Figure 2. Chunk size estimation. In the second plot, the labels on the x-axis and the y-axis are “Normalized logarithm of level id” and “Normalized number of clusters”, respectively; the sigmoid curve overlaps with the curves corresponding to $\alpha = 0.0005$ or 0.001.

pairs are now deemed to have the same similarity measures. The algorithm processes the m chunks sequentially.

We use a different counter r' for coarse-grained hierarchical clustering. When a chunk $\widehat{L}_s$, where $1 \leq s \leq m$, is processed, we update counter r' to be s. We also collect the list of all incident edge pairs in this chunk, which are assumed to have the same similarity measure. For each incident edge pair (e_{ik}, e_{jk}) on the list, we apply the same MERGE procedure in Section IV-B, except that in Eq. (5) we output counter r' rather than r in the dendrogram.

We explore the dynamics between the number of clusters and the number of incident edge pairs processed, with a month of Twitter data from which a word association graph $G(V, E)$ is constructed with 3,132 vertices and 1,628,578 edges (see Section III). We divide the incident edge pairs into chunks of size 1,000 in the non-increasing order of similarity values. Figure 2(1) shows the number of changes on array C against the level identifier normalized between 0 and 1. Hence, most changes occur in lower half levels.

We further use experiments to gain intuition about the predictive models. We use the same month of Twitter data, and generate three input graphs by varying the fraction of words chosen to be in the graph among 0.0005, 0.001, and 0.005 (more details will be given in Section VII). We divide the incident edge pairs into chunks of equal length, and plot in Figure 2(2) the number of clusters against the *logarithm* of the level identifier. To align the three curves produced from different input graphs together, we normalize both axes to be between 0 and 1. From the figure, we observe that all the three curves have a similar shape: it decreases slowly at the beginning, sharply in the middle, and then slowly again at the tail. Such a phenomenon can be modeled well with the sigmoid function: $y = f(x) = \frac{a}{1+e^{-k(\log x - b)}} + c$, where x is the level identifier – or equivalently, the fraction of incident edge pairs that have already been processed, y is the number of clusters, and a, b, c, and k are model parameters. For instance, letting $a = -1$, $b = 0.48$, $c = 1$ and $k = 10$, the curve produced from the sigmoid function agrees well with the curves when the fraction is 0.0005 or 0.001 in Figure 2(2).

A. Mode transition machine

We define the soundness property of coarse-grained link clustering as follows: the ratio of the numbers of clusters between any two consecutive levels is no greater than a given threshold γ until the total number of clusters is smaller than a certain number ϕ. The soundness property ensures that *the clusters do not merge too quickly from the bottom of the dendrogram towards its root*. Finally, the initial chunk size is given by δ_0. Parameters (γ, ϕ, δ_0) define the shape of the produced coarse-grained dendrogram.

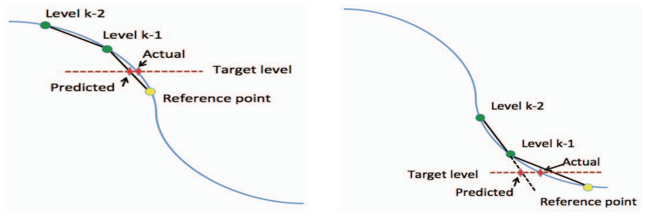
Our algorithm distinguishes three modes, `head`, `tail`, and `rollback`. The `head` mode is used to reflect the top half of the curves in Figure 2(2), where the number of clusters formed is no smaller than half of the edges in graph $G(V, E)$, i.e., $|E|/2$. By contrast, when in the `tail` mode, there are fewer than $|E|/2$ clusters formed by the edges in E. To strictly ensure that the clusters do not merge with a higher rate than γ from level to level, we let the algorithm enter a `rollback` mode when this occurs.

We use δ to denote the current estimated chunk size, which is initialized as δ_0. Moreover, Δ, which is initialized to be 0, denotes the sum of all the previous chunk sizes. We further use β to denote the number of clusters at the previous level, and ξ the number of incident edge pairs that have been processed. Initially, we have: $\beta = |E|$ and $\xi = 0$. A pointer p points to the starting vertex pair of list L.

With these notations, the next chunk size is estimated as follows in the coarse-grained sweeping algorithm. It iterates over all the vertex pairs on list L, and in each iteration, assuming that the list of common neighbors for the current vertex pair is l, it checks whether $\xi + |l| < \Delta + \delta$. If so, which means the chunk has not reached its current end yet, the algorithm updates ξ by increasing it by $|l|$ and then continues merging the incident pairs on list l using the MERGE procedure. Otherwise, the algorithm enters a new *epoch*, in which it uses array C to calculate the current number of clusters β' and then checks the following:

- $C_1 : \beta' \leq |E|/2$. This checks whether the algorithm should be in the `head` or the `tail` mode.
- $C_2 : \beta/\beta' \leq \gamma$. This checks whether the number of clusters at this level has been reduced too much against that at the previous level.
- $C_3 : \beta' \leq \phi$. This checks whether the number of clusters at this level is small enough that they can be merged into a single one at the root level.

These predicates decide the transitions among different



(1) Concave scenario (2) Convex scenario

Figure 3. Chunk size estimation in a rollback mode

modes as illustrated in Figure 2(3). Note that the negation of predicate x is given by $\bar{x}$.

Once the algorithm transits into a new epoch, it replaces β with the latest value, β', add δ to Δ (i.e., $\Delta \leftarrow \Delta + \delta$), and let pointer p points to the current vertex pair on list L. We call the tuple $Q = (\beta, \Delta, p, C)$ the current *epoch state*. Thereafter, the algorithm determines the mode of the current epoch according to the mode transition machine shown in Figure 2(3), and then operates as follows:

- *Case I:* the new mode is either `head` or `tail`. The current epoch state Q is finalized and is saved as Q^*, and the level of the dendrogram grows by 1.
- *Case II:* the new mode is `rollback`. The current epoch state, after being saved on a list $L_{rollback}$, is rolled back to the previously saved safe one Q^*.

Rollbacks in *Case II* introduce extra computational cost, as it has to reprocess some incident edge pairs from the previous safe state Q^*. To reduce recomputation, whenever the algorithm decides to roll back, it bookkeeps the current epoch state on list $L_{rollback}$. Then for *Case I*, before estimating the next chunk size, it first searches the epoch states on list $L_{rollback}$ for the one that meets two requirements: (1) its number of clusters β'' is smaller than that at the current level, i.e., β, and (2) it has the smallest number of clusters that satisfies $\beta/\beta'' \leq \gamma$. If such an epoch state is found, the algorithm jumps directly to that epoch state without having to recomputing it, and then restarts a new epoch from there.

B. Chunk size estimation

Estimation of the next chunk size depends on the mode:

The `head` mode: When in the `head` mode, the estimated size of the next chunk, i.e., α, grows exponentially as $\alpha \leftarrow \alpha\eta$, where factor η is initialized to be $\eta_0 > 1$. Moreover, whenever there is a transition from mode `head` to `rollback`, the factor η is updated to be $1 + (\eta - 1)/2$. Hence, $\eta - 1$ reduces by half whenever a rollback occurs.

The `rollback` mode: When the algorithm enters a `rollback` mode, its current epoch state is used as a *reference point* before it is rollbacked to a previous safe state. Figure 3 illustrates how to estimate the next chunk size based on the reference point along with the epoch states in the past two levels. Let the number of clusters at level k be β_k. Moreover, $\tilde{\gamma}$, where $1 \leq \tilde{\gamma} \leq \gamma$, is used to define a target merging rate of the coarse-grained dendrogram. Currently, we let $\tilde{\gamma}$ be $(1+\gamma)/2$.

There are two scenarios: *concave* and *convex*. In the *concave* scenario, we use the slope of the line formed by the reference point and the last level to extrapolate the next chunk size given the target number of clusters at the next level, which is $\beta_{k-1}/\tilde{\gamma}$; by contrast, in the *convex* scenario, we use the slope of the line formed by the previous two levels to extrapolate the next chunk size given the target number of clusters at the next level, which, again, is $\beta_{k-1}/\tilde{\gamma}$. In each scenario, we can use the steeper slope of the two lines, one between the last two levels and the other between the last one and the reference point, to predict the next chunk size. Hence, the estimated chunk size is expected to be smaller than the actual chunk size that achieves the target number of clusters at the next level. In cases where consecutive rollbacks occur, we further halve the difference between the current estimated value and the previous level (i.e., level $k-1$ in Figure 3), and redo the computation from the previous safe level.

The `tail` mode: When a new epoch is in a `tail` mode, it first checks whether there is an epoch state saved for the future on list $L_{rollback}$. If so, the epoch state s^* satisfying the following property is chosen as a reference point:

$$\tilde{\beta}(s^*) < \beta \wedge \forall s \in L_{rollback} \text{ with } \tilde{\beta}(s) < \beta : \tilde{\beta}(s^*) \geq \tilde{\beta}(s), \quad (6)$$

where $\tilde{\beta}(s)$ is the number of clusters in epoch state s. That is to say, the reference point is the closest epoch state saved for the future on list $L_{rollback}$. With this reference point, the same method used to estimate the next chunk size in a `rollback` mode is applied to predict the next chunk size.

When no such reference point can be found from list $L_{rollback}$, we use the slope between the previous two levels to extrapolate the next chunk size, which is the same as Figure 3(2) as the reference point is not used in extrapolation.

VI. Multi-threading

A. Parallelization of initialization phase

The initialization phase is comprised of three passes of graph G, each of which is parallelized as follows.

First pass: In this pass, two vectors H_1 and H_2 are created, and updated by iterating through each vertex in G and its neighbor list (Lines 2-5 of Algorithm 1). Hence, parallelizationcan be trivially done by partitioning the vertices in G into disjoint vertex sets of approximately the same size and then letting a thread process vertices in each set.

Second pass: In this pass, vertex pairs sharing the same neighbors are inserted into map M with accumulating weights. As updating the state of M simultaneously by multiple threads may not be safe, this pass is parallelized in two steps. In the first step, each thread maintains its own map, and similar to the first pass, it processes a disjoint set of vertices (Line 7 of Algorithm 1). In the second step, multiple threads are used to merge individual maps in a hierarchical fashion. Suppose that we have six maps $M_0, M_1, ..., M_5$ after the first step (so $T = 6$). In the second step, the first iteration uses three threads: the first one merges M_1 into M_0, the second M_3 into M_2, and the last M_5 into M_4; in the next iteration, as there are fewer than four maps left, a single thread is used to merge all of them into M.

Third pass: This pass traverses all the edges, and if the edge appears in map M, the map is updated within Lines 22-24 of Algorithm 1. To separate the regions updated by different threads, we partition the vertex pairs in map M into disjoint sets according to the first vertices in these pairs, and an update on Line 23 of Algorithm 1 is done by the thread only if the first vertex falls into the corresponding vertex set.

B. Parallelization of sweeping phase

Parallelizing coarse-grained hierarchical clustering involves two steps. In the first step, we keep T duplicate copies of array C. We then partition the current chunk into T disjoint sets of edge pairs, which are approximately of the same size. Each thread uses `MERGE` to process one of these sets of edge pairs on its own copy of array C.

In the second step, we merge the T copies of array C obtained from the previous step, which are denoted C_0, C_1, ..., C_{T-1}. Caution, however, must be exercised when this is done. Suppose that we want to merge C_1 into C_0, and consider the following scheme. We iterate through every edge in array C_1 from 0 to $|E|-1$; for each $i \in [0, |E|-1]$, we calculate $\min\{F_0(i), F_1(i)\}$ as f, where $F_0(i)$ and $F_1(i)$ are defined to be the same as $F(i)$ in Eq. (4) except that array C is replaced with C_0 and C_1, respectively, and then for each $e \in F_0(i) \cup F_1(i)$, we update $C_0[e]$ to be f. As an edge is always mapped to either itself or one that has a smaller identifier, the scheme seems to be a natural extension of the original `MERGE` procedure in the serial version.

However, the above scheme is flawed. Suppose $C_0 = [1 \to 1, 2 \to 2, 3 \to 2, 4 \to 1]$, and $C_1 = [1 \to 1, 2 \to 2, 3 \to 3, 4 \to 3]$. Using the scheme, after processing the first three edges 0, 1, and 2, C_0 becomes intact. When processing edge 4, as $F_0(4) = \{1, 4\}$ and $F_1(4) = \{3, 4\}$, we have $C_0 = [1 \to 1, 2 \to 2, 3 \to 1, 4 \to 1]$. Clearly, it has two clusters (i.e., 1 and 2), which is wrong as merging the original arrays C_0 and C_1 should have all four edges belong to the same cluster.

A slight change addresses the flaw. After f is calculated, for each $e \in F_0(i) \cup F_1(i) \cup F_0(\min F_1(i))$, we update $C_0[e]$ to be f. In the above example, as $F_0(\min F_1(i))$ contains both edges 2 and 3, edge 2 should also be mapped to edge 1, leading to a single cluster.

In the second step, with the above scheme to merge two arrays, a hierarchical structure can again be used to merge the T arrays obtained from the first step. To start with, we let all T arrays be active. In each iteration, with k active arrays, we form $\lfloor k/2 \rfloor$ disjoint pairs and use a single thread implementing the above scheme to merge each pair into a single active array. If k is odd, we carry the last active array to the next iteration. If in an iteration there are at most three active arrays, we use a single thread to merge them together and the iteration stops; otherwise, a new iteration is started to merge active arrays produced from the previous one.

VII. Experiments

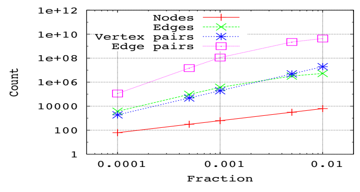
We use a Linux workstation with the 6-core Intel Xeon E5649 CPU and 64G RAM for experiments. We use the tweets collected during December 2011 to construct the word association network as the input to the link clustering algorithm. Some statistics of the dataset have been given in Section V. Only English tweets are considered when constructing the graph. We use the `nltk` library [10] to stem each word in a tweet (the porter stemming algorithm is used here), and remove those common stop words list in [11]. The words left remain as candidate words. We sort these candidate words in the non-ascending order of the number of appearances in all the tweets. Only a fraction α of these candidate words that appear most frequently in all English tweets are used to create vertices in graph $G(V, E)$. We use fraction α to control the graph size.

A. Serial algorithm evaluation

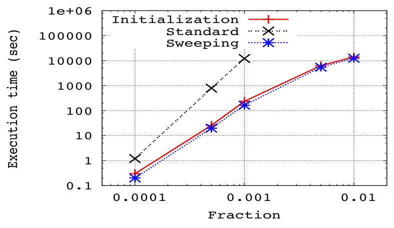
For the serial execution mode, we implement both our algorithm and a standard single-linkage hierarchical clustering algorithm as a single-thread process. For the standard algorithm, we choose to implement one based on a next-best-merge array [8], whose time complexity is $O(n^2)$. Hence the standard algorithm used here for comparison is optimally efficient, which is similar to the SLINK algorithm [7].

For performance comparison under different problem sizes, we choose α among 0.0001, 0.0005, 0.001, 0.005, and 0.01. Figure 4(1) shows the number of vertices and edges in graph $G(V, E)$. The density of an undirected graph is defined to be $\frac{2|E|}{|V|(|V|-1)}$. When α varies among 0.0001, 0.0005, 0.001, 0.005, and 0.01, the density of the graph is 1.0, 0.997, 0.963, 0.332, and 0.136, respectively. Hence, when we increase the number of vertices in the graph, the density of the graph decreases, suggesting that frequent words are more likely to appear simultaneously in the same tweet. In the same plot, we also show the number of vertex pairs on list L (i.e., K_1) as well as the number of distinct incident edge pairs (i.e., K_2). It is observed that K_2 dominates $|E|$ by only about 2 $\sim$ 4 orders of magnitude. According to Theorem 2, the performance of the sweeping algorithm is essentially $O(\sqrt{K_2}|E|)$, as we have $|V| \ll |E|$ here. Hence, when $|E|$ becomes large, the sweeping algorithm provides significant performance improvement over the standard algorithm, whose time complexity is $O(|E|^2)$.

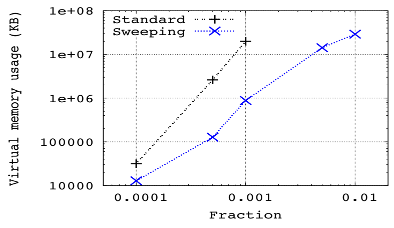
For each setup, we run the program for 10 times. The average execution times of both the sweeping algorithm and the standard algorithm are shown in Figure 4(2), as well as that of the initialization phase during which the similarity measures are calculated. It is observed that for the sweeping algorithm, its execution time is comparable to that of the initialization phase across different settings of fraction α.



(1) Statistics

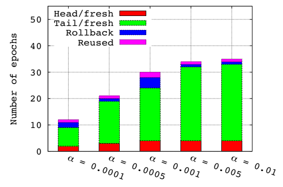


(2) Execution time

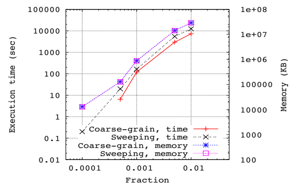


(3) Virtual memory usage

Figure 4. Performance in serial execution mode



(1) Epoch breakdown



(2) Execution performance

Figure 5. Execution performance of coarse-grained hierarchical clustering

However, using the machine described above, we can only finish the standard algorithm for fractions 0.0001, 0.0005, and 0.001. For these three fractions, the speedup of the sweeping algorithm over the standard algorithm is 2.0, 40.0, and 74.2, respectively.

Figure 4(3) presents the virtual memory usages of the program under different settings. We can see that when α = 0.001, the memory footprint of the standard algorithm is 19.9 GB, as opposed to only 881.2 MB used by the sweeping algorithm. Moreover, the sweeping algorithm can finish the entire procedure of hierarchical clustering even when $\alpha = 0.01$ with a memory footprint of 29GB, far less than the available memory on the machine running the experiments.

B. *Coarse-grained hierarchical clustering*

We set parameters described in Section V-A as follows: $\gamma = 2$, and $\phi = 100$. As we have $\tilde{\gamma} = (1+\gamma)/2$, the target merging rate $\tilde{\gamma}$ is thus 1.5. The initial chunk size δ_0 is set to be 100, 500, 1000, 5000, and 10000, respectively, when we vary fraction α among 0.0001, 0.0005, 0.001, 0.005, and 0.01. In the `head` mode, the initial factor η_0 by which the estimated chunk size grows is set to be 8.

Figure 5(1) presents the modes of these epochs under different α settings (`fresh` means that the new epoch state is *fresh* and no previous rollback states are used, and `reused` means that the epoch uses a previous rollback state). We observe that only a small fraction of epochs are in the `head` mode. The reasons are two-fold. First, during the `head` mode, the chunk sizes grow exponentially in each new epoch. Second, note that the x-axis in Figure 2(2) is shown at a logarithmic scale. Hence, the majority of incident edge pairs are actually processed in the `tail` mode.

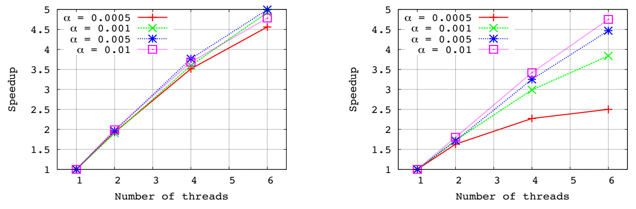
We show the execution performances of the coarse-grained sweeping algorithm in Figure 5(2). For comparison, the same figure also presents the execution performances of the original sweeping algorithm. We observe that the coarse-grained sweeping algorithm takes less time to finish than the original one, which sounds counter-intuitive as rollbacks introduce extra computational overhead to the coarse-grained sweeping algorithm. Note that for fraction $\alpha = 0.0001$, the execution time of the coarse-grained sweeping algorithm is so small that it is rounded to 0. This interesting observation can be explained by the fact that the long tails of the curves in Figure 2(2) contain a large number of incident edge pairs given that the x-axis is shown at a logarithmic scale. As the coarse-grained sweeping algorithm stops when the number of clusters goes below a certain threshold, which is 100 in our experiments, a large portion of incident edge pairs at the tail are not processed. For instance, when $\alpha = 0.005$, only 55.1% of the incident edge pairs need to be processed until the last level of the dendrogram.

C. *Multi-threading evaluation*

We vary the number of threads among 1, 2, 4, and 6. We define the speedup to be the ratio of the average execution time with a single thread to that with multiple threads. We ignore the cases when $\alpha = 0.0001$ as its serial execution time is trivial. Figure 6 shows the speedups of our multi-threading algorithm under different α settings. From Figure 6(1) we observe that the execution time of the initialization phase scales well with the number of threads, and the speedups over different α settings are comparable. For instance, across different settings of fraction α, the speedups are close to 2.0 with two threads, lie between 3.5 and 4.0 with four threads, and lie between 4.5 and 5.0 with six threads. The high scalability of the initialization phase, which we recall performs three passes over the input graph, can be explained as follows. In the first and the third passes, computation involves iterating the vertex list in the graph (first pass) and the vertex mappings in map M (third pass). Hence, workload balancing is achieved if workload assignment to the threads is done in a round robin fashion. In the second pass, the first step is also highly parallelizable because each thread deals with its own portion of vertices. The second step of the second pass may have some threads idle if they are not used for merging the individual maps.

References

[1] Y.-Y. Ahn, J. P. Bagrow, and S. Lehmann, "Link communities reveal multiscale complexity in networks," *Nature*, 2010.



(1) Initialization (2) Sweeping

Figure 6. Speedup on a multi-core workstation

[2] S. E. Schaeffer, "Graph clustering," *Computer Science Review*, vol. 1, no. 1, 2007.

[3] J. J. Carrasco, J. Joseph, C. Daniel, C. Fain, K. J. Lang, and L. Zhukov, "Clustering of bipartite advertiser-keyword graph," in *Workshop on Clustering Large Data Sets*, 2003.

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

[5] H. Du, M. W. Feldman, S. Li, and X. Jin, "An algorithm for detecting community structure of social networks based on prior knowledge and modularity," *Complexity*, 2007.

[6] J. Hopcroft, O. Khan, B. Kulis, and B. Selman, "Natural communities in large linked networks," in *ACM KDD'03*.

[7] R. Sibson, "SLINK: an optimally efficient algorithm for the single-link cluster method," *The Computer Journal*, 1973.

[8] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge university press, 2008.

[9] J. C. Gower and G. J. S. Ross, "Minimum spanning trees and single linkage cluster analysis," *Applied statistics*, 1969.

[10] http://www.nltk.org/.

[11] http://www.clips.ua.ac.be/pages/stop-words.

[12] D. de Caen, "An upper bound on the sum of squares of degrees in a graph," *Discrete Mathematics*, 1998.

Appendix

Proof. Without loss of generality, we assume that graph $G(V, E)$ is stored as an adjacency list with weights stored along with the edges. In the initialization step, assuming that M is implemented with a hash map where both lookup and insertion take $O(1)$ time, it can be created within $O(|V|+|E|+K_2)$ time using $O(K_2)$ space. Sorting the keys in map M can be done within $O(K_1 \log K_1)$ time. Initializing array C takes $O(|E|)$ time and $O(|E|)$ space.

As list L takes space $O(K_2)$ and each entry is visited once, it takes $O(K_2)$ time to access and update this data structure.

Next we analyze the overall time spent on accessing and updating array C in all the iterations of the `sweeping` phase. Note that after initialization, we have $\sum_{i=1}^{|E|} C[i] = |E|(|E|+1)/2$, and after the `sweeping` phase terminates, we must have $\sum_{i=1}^{|E|} C[i] \geq |E|$ (as every element in array C is at least 1).

It is observed that array C is accessed by in total K_2 times, each due to a pair of edges to be merged. Let $h_k^{(1)}$ and $h_k^{(2)}$ denote the cardinalities of $F(i_1)$ and $F(i_2)$, respectively, during the k-th time we access array C, where $1 \leq k \leq K_2$. For simplicity, let h_k be $(h_k^{(1)} + h_k^{(2)})/2$. As $|F(i_1) \cup F(i_2)| \geq \max\{|F(i_1)|, |F(i_2)|\} \geq h_k$, we have visited the elements of array C for at least h_k distinct edges. After we merge the edges, their corresponding values in array C are all reduced to their minimum. Hence, the reduction in $\sum_{e\in F(i_1)\cup F(i_2)} C[e]$ after merging is at least $h_k(h_k-1)/2$.

Therefore, we have the following:

$$\sum_{k=1}^{K_2} \frac{h_k(h_k-1)}{2} \leq \frac{|E|(|E|+1)}{2} - |E|, \tag{7}$$

as the RHS (Right Hand Side) in Eq. (7) is the total amount of reduction over all the iterations. Eq. (7) can further be simplified:

$$\sum_{k=1}^{K_2} h_k^2 - \sum_{k=1}^{K_2} h_k \leq |E|(|E|-1). \tag{8}$$

Applying the Cauchy-Schwarz inequality on the LHS of Eq. (8):

$$\sum_{k=1}^{K_2} h_k^2 - \sum_{k=1}^{K_2} h_k \quad \geq \frac{(\sum_{k=1}^{K_2} h_k)^2}{K_2} - \sum_{k=1}^{K_2} h_k. \tag{9}$$

With $X = \sum_{k=1}^{K_2} h_k$ and combining Eqs. (8) and Eq. (9):

$$X(X-K_2) \leq K_2|E|(|E|-1) < K_2|E|^2. \tag{10}$$

Note that in total the algorithm visits the elements in array C by $2X$ times. There are two cases. In the first one, $X \leq K_2$. In the second one when $X > K_2$, we have $X \leq K_2 + \sqrt{K_2}|E|$. As we have $K_2 \leq K_3 = |E|(|E|-1)/2$, we know that K_2 is within $O(|E|^2)$. Hence, $\sqrt{K_2}|E|$ dominates K_2 asymptotically. In other words, it takes $O(\sqrt{K_2}|E|)$ time to access and update array C in our algorithm.

Therefore, the overall time complexity of our algorithm is $O(|V| + K_1 \log K_1 + \sqrt{K_2}|E|)$, and the overall space complexity of our algorithm is $O(|E| + K_2)$. ■

The SLINK algorithm [7] provides an optimally efficient solution to the generic single-linkage clustering problem. When it is directly applied to link clustering, it takes $O(|E|^2)$ time. Hence, the performance gain of our algorithm hinges upon the relationships among K_1, K_2, and $|E|$. First, note that it is not necessarily true that $K_1 \geq |E|$ or $K_2 \geq |E|$. For example, in a graph with disjoint singular edges, $K_1 = K_2 = 0$ but $|E| = |V|/2$.

Assume that the degrees of the vertices in graph G are d_0, d_1, ..., $d_{|V|-1}$, respectively. We thus have:

$$K_2 = \frac{\sum_{i=0}^{|V|-1} d_i(d_i-1)}{2} = O(\sum_{i=0}^{|V|-1} d_i^2). \tag{11}$$

According to Caen [12], we have:

$$\sum_{i=0}^{|V|-1} d_i^2 \leq |E|(\frac{2|E|}{|V|-1} + |V| - 2). \tag{12}$$

Hence, we have $K_2 = O(|E|^2/|V| + |E||V|) = O(|E||V|)$. Hence $\sqrt{K_2}|E| = O(|E|^2\sqrt{|V|/|E|})$.

On the other hand, as $K_1 \leq K_2$ and $|E| \leq |V|(|V|-1)/2$, we have $K_1 \log K_1 = O(|E||V|log(|E||V|)) = O(|E||V|log|V|)$. Consider the case where $|E| = \omega(|V|log^2|V|)$ where ω is the negation of O in complexity theory. As $\log|V| = O(|V|^\epsilon)$, for any positive number ϵ no matter how small it is, we have $K_1 \log K_1 = O(|E|^2\sqrt{|V|/|E|})$. Also, due to $|E| = \omega(|V|)$, we must have $|V| = O(\sqrt{|V||E|}) = O(|E|^2\sqrt{|V|/|E|})$.

We thus establish the following corollary:

Corollary 1: If $|E| = \omega(|V|log^2|V|)$, the time complexity of our algorithm is $O(|E|^2\sqrt{|V|/|E|})$.

Here, with a large graph with $|E| = \omega(|V|log^2|V|)$, our algorithm outperforms SLINK by a factor of *at least* $\sqrt{|E|/|V|}$ asymptotically. Note that this provides only a lower bound. Next, we consider some example graphs to demonstrate the performance gain. (1) In a k-regular graph where $k = \omega(|V|^\epsilon)$ for any $\epsilon > 0$ (so $|E| = \omega(|V|^{1+\epsilon})$ and $K_1 \log K_1$ is dominated by $\sqrt{K_2}|E|$), for instance, $K_2 = |V|k(k-1)/4$, but $|E| = k|V|/2$; hence, $|E|^2$ dominates $\sqrt{K_2}|E|$ by a factor of $\sqrt{|V|}$ asymptotically. (2) If G is a complete graph, we have $K_2 = |V| \times \frac{|V|(|V|-1)}{2}$, suggesting that our algorithm takes time $O(|V|^{3.5})$; by contrast, the SLINK algorithm takes time $O(|V|^4)$ as $|E| = |V|(|V|-1)/2$. Hence, our algorithm still has a factor of $O(\sqrt{|V|})$ improvement over SLINK.