Parallel Filtered Graphs for Hierarchical Clustering

Abstract

Given all pairwise weights (distances) among a set of objects, filtered graphs provide a sparse representation by only keeping an important subset of weights. Such graphs can be passed to graph clustering algorithms to generate hierarchical clusters. In particular, the directed bubble hierarchical tree (DBHT) algorithm on filtered graphs have been shown to produce good hierarchical clusters for time series data, including financial time series.

We propose new parallel algorithms for constructing triangulated maximally filtered graphs (TMFG) and a scalable parallel algorithm for generating DBHTs that is optimized for TMFG inputs. In addition to parallelizing the original TMFG construction, which has limited parallelism, we also design a new algorithm that inserts multiple vertices on each round to enable more parallelism. We show that the graphs generated by our new algorithm have similar quality compared to the original TMFGs, while being much faster. Our new parallel algorithms for TMFGs and DBHTs are 39.75–208.9x faster than state-of-the-art implementations, while achieving up to 31.7x self-relative speedup on 48 cores with hyper-threading, and achieve better clustering results compared to the standard average-linkage and complete-linkage hierarchical clustering algorithms. We show that on two stock data sets, our algorithms produce clusters that align well with human experts' classification.

1 Introduction

Clustering is an unsupervised machine learning method that has been widely used in many fields including finance, biology, and computer vision, to discover structures in a data set. Often times, one is interested in exploring clusters at varying resolutions. A *hierarchical clustering* algorithm can be used to produce a tree, also known as a *dendrogram*, that represents clusters at different scales.

Running a metric clustering algorithm on a set of n points often involves working with $\Theta(n^2)$ pairwise distances, and is computationally prohibitive on large data sets. One approach to improving efficiency is to use a *filtered graph* that keeps only a subset of the pairwise distances, and then pass the resulting graph to a graph clustering algorithm. Filtered graphs reduce the number of distances considered while retaining the most important features, both locally and globally. Simply removing all edges with weights below a certain threshold may not perform well in practice, as the threshold may require significant tuning, and the importance of an edge is not only determined by its weight, but also its location in the graph. Several other methods for constructing filtered graphs have been proposed, including k-nearest neighbor graphs [18], minimum spanning trees [12, 25], and weighted maximal planar graphs [14, 24].

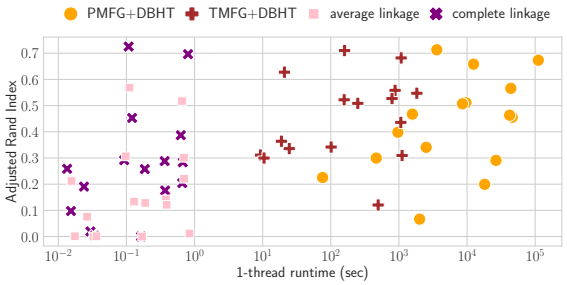

Figure 1: Sequential runtime vs. clustering quality for hierarchical clustering methods on different data sets.

A variant of weighted maximal planar graphs, called planar maximally filtered graphs (PMFG) [24], has been shown to perform well in practice in combination with the directed bubble hierarchy tree (DBHT) technique, especially on financial [16, 27, 29] and biological [3, 22] data sets. The PMFG is constructed by repeatedly adding an edge between the pair of points with the highest weight, while preserving planarity. The DBHT technique constructs a dendrogram based on certain triangles in the input graph, along with various shortest path calculations. It has the benefit that no prior information about the data is required, and no parameter tuning is needed. However, the state-of-the-art PMFG and DBHT algorithms are sequential and do not scale to large data sets. Massara et. al [14] proposed the triangulated maximally filtered graphs (TMFG), a variant of maximal weighted planar graph that can be generated more efficiently. Instead of repeatedly adding a single edge as in PMFG, the TMFG is constructed by repeatedly adding a single vertex to a triangle with its three edges to the triangle corners, while respecting planarity. However, the state-of-the-art TMFG implementation is also sequential, and to the best of our knowledge, the clustering quality of using DBHTs with TMFGs has not been evaluated.

One important type of data that PMFGs, TMFGs, and DBHTs have been shown to perform well on is *time series data*. Time series data arise in a multitude of applications, including in finance, signal processing, biology, astronomy, and weather forecasting. To extract insights from the data, one is often interested in finding correlations between different time series, and clustering the data based on these correlations. The *correlation matrix* stores the correlation between every pair of time series It is important to construct a filtered graph on the correlation matrix to enable efficient and scalable clustering. We show in Figure 1 the runtime and cluster quality (using the Adjusted Rand Index [7]) for PMFG and TMFG combined with DBHT, compared with the standard average-linkage and complete-linkage methods for hierarchical clustering, for a collection of time series data sets. For all methods, the dendrogram is cut such that the number of resulting clusters is the same as the number of ground truth clusters. We see that, while the runtimes of PMFG and TMFG are higher than those of average-linkage and complete-linkage, they are able to generate higher quality clusters. Thus, PMFG, TMFG, and DBHT are preferable for hierarchical clustering, when one is willing to use a larger time budget to obtain better quality clusters.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

To reduce the runtime of TMFG and DBHT, we propose new parallel algorithms and fast implementations for constructing TMFGs and a scalable parallel DBHT algorithm that is optimized for TMFGs. Besides parallelizing the original TMFG construction, which has limited parallelism, we also design a new algorithm that inserts multiple vertices on each round to enable more parallelism. We show that the filtered graph generated by our new algorithm has similar quality to that of the original TMFG, while being much faster. The key challenge in our algorithm is in resolving the conflicts when inserting multiple vertices into the TMFG. In TMFG construction, we resolve the conflict of vertices by designating a single triangle for each vertex based on edge weights. Our DBHT algorithm leverages the special topological structure of TMFGs. We are able to construct a rooted undirected bubble tree on the fly with a special invariant while constructing the TMFG, and use this invariant to efficiently direct the bubble tree with a recursive process. This reduces the complexity of these two steps from quadratic (which the original DBHT algorithm requires) to linear.

We compare the speed and quality of our parallel algorithms with parallelized versions of the average-linkage and complete-linkage hierarchical clustering algorithms. As a baseline, we also compare with k-means, which is a non-hierarchical clustering algorithm and only produces clusters at a single resolution. On a collection of 16 data sets generated from time series and image data, we find that the DBHT using TMFG produces similar and sometimes better clusters than the DBHT using PMFG. It also gives better clusters than average-linkage and complete-linkage clustering, and is competitive with k-means on most data sets.

We show that our new algorithm is up to 12075 times faster than the sequential DBHT on PMFG and have Adjusted Rand Index scores up to 0.65 higher than agglomerative clustering algorithms. We show that on time series data sets of stocks from 2013–2019 from both the US stock market and the Toronto Stock Exchange, DBHT on TMFG is able to produce clusters that align well with human experts' classification, and in some cases can produce better clusters than the original TMFG algorithm. On one thread, our new algorithm for constructing a TMFG variant and DBHT are 39.75–208.9 times faster than the state-of-the-art TMFG implementation, and achieves up to 31.7x self-relative speedup on 48 cores.

We summarize our contributions below:

- A new parallel algorithm for constructing a TMFG variant that produces high-quality graphs.
- A new parallel algorithm for constructing the DBHT, which is optimized for TMFG-like inputs.
- Experiments showing that our parallel algorithms achieve significant speedups over state-of-the-art algorithms, while producing clusters of similar or better quality.

Our source code and data is available at https://anonymous.4open.science/r/par-filtered-graph-clustering-85DB/.

2 Background

We briefly describe the sequential construction of PMFGs and TMFGs. We also introduce the notation and primitives that we use.

A ***planar*** graph can be drawn on the plane such that no edges cross each other. Both PMFGs and TMFGs are maximal planar subgraphs of a complete undirected, edge-weighted graph ("maximal" means that no additional edge can be added to the vertex set without violating planarity). They give an approximation to the NP-hard Weighted Maximum Planar Graph problem, which requires the sum of the edge weights kept to be maximized [6]. The complete edge-weighted graph input can also be viewed as a similarity matrix S, where $S[i, j]$ is the weight of edge (i, j) in the graph.

Planar Maximally Filtered Graph (PMFG). The sequential algorithm for constructing PMFGs [24] first sorts all of the edges by their edge weights. It then starts with an empty graph G and considers each edge in the sorted order, from highest to lowest weight. An edge is added to the graph if and only if it does not violate the planarity of G, which is checked by running a planarity testing algorithm. This process is computationally expensive due to the need to run a planarity testing algorithm $\Theta(n^2)$ times.

Triangulated Maximally Filtered Graph (TMFG). TMFGs [14] have been proposed to improve the efficiency of PMFG construction. Instead of considering edges one by one, the original sequential TMFG algorithm adds one vertex to a triangle in the graph, as well as an edge to each of the three corners of the triangle, on every iteration, and eliminates the vertex from consideration in future iterations. The vertex-triangle pair that gives the highest increase in total edge weight is chosen. This leads to only n iterations for TMFG, compared to $\Theta(n^2)$ for PMFG. The algorithm starts with four vertices with maximum weighted degree and all six edges connecting them. By definition, it ensures that edges added do not violate planarity, and so planarity testing is not needed. Figure 2(a) shows an example of a TMFG. If there were additional vertices to insert, they could go into any of the faces, which are all triangles.

Directed Bubble Hierarchy Tree (DBHT). After obtaining the TMFG or PMFG, we can then generate a dendrogram from it for hierarchical clustering using the DBHT [21] algorithm, which has been shown to produce high-quality dendrograms for financial time series [16, 27, 29] and biological [3, 22] data. The original, sequential DBHT algorithm first constructs a *bubble tree* [20], which is a tree where nodes correspond to planar subgraphs, and edges between two nodes correspond to triangles in the original graph that separate the two corresponding planar subgraphs. The DBHT algorithm then directs the tree edges by computing the strength of connection from the separating face to its interior and exterior. Next, the algorithm runs a number of shortest distance computations to assign vertices to bubbles, and finally uses complete-linkage clustering to generate a hierarchy. We will describe more details of the DBHT algorithm along with how we parallelize and optimize it in Section 4.

Notation and Terminology. Given a planar graph, a ***face*** of G is a maximal region bounded by edges, except for the ***outer face***, which is unbounded. For example, in Figure 2(a), $\{0, 3, 6\}$ is the outer face. A bounded face is a ***inner face***. We ***insert*** vertex v into a triangular face t with three corners v_x, v_y, and v_z by adding three weighted edges from v to each of v_x, v_y, and v_z, based on the pairwise distances. The ***gain*** of inserting v to t is the sum of these three edge weights. We say that a vertex $v \in V$ is the ***best vertex in*** V for face t if inserting v to t gives the maximum gain among all vertices in V. The weight of an edge (i, j) in graph G is denoted as $w(i, j)$. We will denote a clique with C and a bubble node in the bubble tree as b. The output of our algorithm is a tree called a ***dendrogram***, where the height of each node corresponds to the dissimilarity between the two clusters represented by its

Algorithm 1: Parallel TMFG

Input: $n \times n$ similarity matrix S, prefix size PREFIX ≥ 1
1 $C = \{v_1, v_2, v_3, v_4\}$
2 $\mathcal{E} = \{(v_1, v_2)(v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$
// The four triangular faces in the initial graph
3 $\mathcal{F} = \{\{v_1, v_2, v_3\}, \{v_1, v_2, v_4\}, \{v_1, v_3, v_4\}, \{v_2, v_3, v_4\}\}$
4 $V = \{v_5, \ldots, v_n\}$
// The best vertex in V for each triangle
5 GAINS = $\{\text{argmax}_{u \in V}(\sum_{v \in t} S[v, u])$ for $t \in \mathcal{F}\}$
6 Initialize bubble tree T with C
7 OUTERFACE = $\{v_1, v_2, v_3\}$
8 **while** $|V| > 0$ **do**
9 L = the PREFIX vertex-face pairs with the largest gains in GAINS using parallel sorting.
10 For vertices paired with multiple faces in L, only keep the pair with maximum gain, using a parallel filter and parallel sorting.
11 $V = V \setminus \{\text{vertices} \in L\}$
12 **parallel_for** $(v, t = \{v_x, v_y, v_z\}) \in L$ **do**
13 $\mathcal{E} = \mathcal{E} \cup \{(v, v_x), (v, v_y), (v, v_z)\}$
14 $\mathcal{F} = \mathcal{F} \cup \{\{v, v_x, v_y\}, \{v, v_y, v_z\}, \{v, v_x, v_z\}\} \setminus t$
15 **parallel_for** $t \in \{t' : GAINS[t'] = v\} \cup \{\{v, v_x, v_y\}, \{v, v_y, v_z\}, \{v, v_x, v_z\}\}$ **do**
16 GAINS$[t]$ = $\text{argmax}_{u \in V}(\sum_{v \in t} S[v, u])$
17 UpdateBubbleTree(v, t, T)
18 **return** $\mathcal{E}$

two children. Cutting the dendrogram at different heights gives subtrees, which correspond to clusters at different scales.

Parallel Primitives [8]. A parallel ***filter*** takes an array A and a predicate function f, and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order that they appear in A. A parallel ***sort*** takes an array A and a binary function $f(a, b)$ that returns true if $a < b$, and returns a new array containing $a \in A$ in non-decreasing order. A parallel ***maximum*** takes an array A and returns the largest element in A. WRITEADD is a priority concurrent write that takes as input two arguments, where the first argument is the location to write to and the second argument is the value to atomically add to the value at the first location.

3 Parallel TMFG

This section introduces our novel algorithm for parallelizing TMFG construction, whose pseudocode is shown in Algorithm 1. We first give a high-level overview. Our parallel algorithm simultaneously adds multiple vertices to the graph. Adding multiple vertices may give results that differ from the sequential TMFG algorithm because the sequential algorithm updates the graph after every insertion, giving future vertices more faces to choose from, whereas our parallel algorithm will only update the graph after all of the vertices in a round have been added. Therefore, the more vertices we add in parallel, the more we deviate from the sequential algorithm. This gives a tradeoff in the quality of the graph and the amount of available parallelism. We therefore consider adding only a prefix of the vertices that give the highest gain in total edge weight. The prefix size can be tuned for the application and available parallelism.

In the rest of the section, we describe the details of our parallel TMFG algorithm (Algorithm 1). The input is an $n \times n$ symmetric matrix S that represents the complete undirected graph, and a parameter PREFIX. The highlighted lines are used for building DBHT, and will be explained in Section 4. On Lines 1–4, we first find the four vertices that have the highest total sum across their rows in S, and add all edges among them to the resulting graph's edge list $\mathcal{E}$.

Algorithm 2: UpdateBubbleTree

1 **Function** UpdateBubbleTree(v, $t = \{v_x, v_y, v_z\}$, T):
2 b^* = new bubble $\{v, v_x, v_y, v_z\}$
3 b = the bubble that t is in
4 **if** t = OUTERFACE **then**
5 b.PARENT = b^*
6 Add b to b^*.CHILDREN
7 OUTERFACE = $\{v, v_x, v_y\}$
8 **else**
9 b^*.PARENT = b
10 Add b^* to b.CHILDREN

The four faces formed by the four vertices are added to the set of faces $\mathcal{F}$. The rest of the vertices are in set V, and will be added to the graph later. On Line 5, for each face in $\mathcal{F}$, we find its best vertex (the vertex that maximizes the gain if inserted into this face).

On Lines 8–18, we insert the remaining vertices into the graph in batches of up to size PREFIX. On Lines 9–11, we obtain the batch of vertices to insert and their corresponding faces to insert into. We first obtain the PREFIX vertex-face pairs with the largest gains in the GAINS array, and store the pairs in L. This can be done using a parallel sort on the GAINS array. If PREFIX = 1, this can be simplified to a single parallel maximum computation. On Line 10, we ensure that a given vertex is only added to a single face to avoid race conditions. Conflicts for a given vertex are resolved by only allowing the vertex to add itself to the face that gives the highest gain. We can use parallel sorting to group vertices based on which face (among all faces) gives the highest gain, and have each face pick the vertex (among all vertices that choose this face) that gives the highest gain. The available parallelism increases as the number of faces in the graph grows. On Line 11, we remove the vertices in L from V using a parallel filter.

On Lines 12–16, we loop over the vertex-face pairs (v, t) in L, add v to t, and update $\mathcal{E}$ and $\mathcal{F}$. To update $\mathcal{F}$, we add the three new faces created and remove t. We also update GAINS by computing the new best vertex among V for the three new faces and faces that previously had v as their best vertex. Unlike the original algorithm [14], which loops over all of the faces to find the faces that previously had v as their best vertex, we keep an array for each vertex that stores such faces to improve the efficiency of this step.

If we choose PREFIX = 1 in Algorithm 1, we obtain the same result as the sequential TMFG algorithm, but we still have some parallelism from the parallel maximum call on Line 9 and the parallel update of the GAINS array on Lines 15–16. This case gives the same parallel algorithm that is described (but not implemented) by Massara et al. [14]. However, we show in Section 5 that using PREFIX = 1 has limited parallelism because on each round only a single vertex can be inserted.

4 Parallel DBHT for TMFG

We describe our parallel algorithm for building the DBHT, which leverages the special structure of the bubble tree for TMFGs. The original DBHT construction of Song et al. [21] takes a planar graph, and two weights on each edge—a similarity measure and a dissimilarity measure. The similarity measure is from the similarity matrix S, and the dissimilarity measure needs to be additionally supplied. The construction has the following steps: building an undirected bubble tree [20] from a planar graph; assigning directions to the

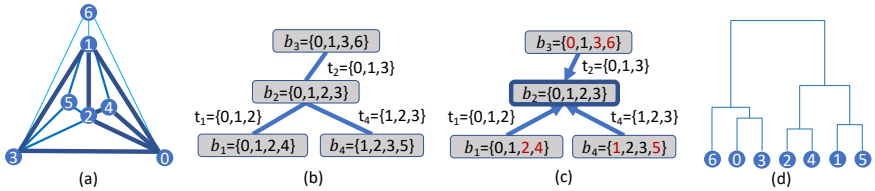

Figure 2: (a) An example TMFG. The edges have weights 0.8, 0.4, or 0.2. The darker and thicker edges have larger weights. For example, $w(0,1) = 0.8$, $w(2,3) = 0.4$, and $w(0,6) = 0.2$. **(b)** The undirected bubble tree. Each bubble node is a gray box and nodes are connected by blue edges. Each bubble is marked with vertices in the bubble and a name b_i. Each edge is marked with the triangle it corresponds to and a name t_i. **(c)** The directed bubble tree. The bolded node is the only converging bubble in this example. The red vertices are assigned to the bubble in which they are marked. **(d)** The dendrogram for DBHT.

bubble tree edges; assigning graph vertices to bubbles; and using complete-linkage clustering to generate a hierarchy. Below, we describe how we accelerate and parallelize each step.

4.1 Bubble Tree for TMFG

As described in Section 2, a *bubble tree* [20] is a tree where nodes correspond to planar subgraphs, and edges between nodes correspond to triangles in the original graph that separate the two corresponding planar subgraphs. Specifically, a ***bubble*** is a maximal planar graph whose triangles are non-separating [20]. The original DBHT construction is inefficient, as it involves first finding all of the triangles in the graph, and then testing for every triangle whether removing it would disconnect the graph. Such triangles are called ***separating*** triangles.

The key observation is that the bubble tree can be more efficiently constructed during TMFG construction, as a vertex that is added in the TMFG algorithm will correspond to exactly one node and one edge in the bubble tree. In TMFG, every time we insert a new vertex, we create a 4-clique, which is a bubble because all faces of a 4-clique are non-separating triangles. This will also produce a separating triangle, which corresponds to a new edge in the bubble tree. This separating triangle is shared by the 4-cliques corresponding to the two bubble nodes incident to it. Therefore, we incorporate the bubble tree construction into our parallel TMFG algorithm to save a significant amount of work.

Our undirected bubble tree for TMFG has the invariant that each bubble node has a parent and at most three children, except for the root, which does not have a parent. Moreover, all descendants of an edge are on the interior side of the separating triangle corresponding to the edge. This invariant is important for us to accelerate the direction computation, which we describe later.

Now we describe the details of bubble tree construction, which are the highlighted lines in Algorithm 1. On Line 6, we initialize our bubble tree with a single bubble node, which contains the 4-clique that we start our TMFG with. On Line 7, we initialize our outer face to be $\{v_1, v_2, v_3\}$. This outer face can be chosen arbitrarily among the four faces of C because the bubble tree's topological structure is independent of this choice [20]. On Line 17, we add a new node and a new edge to the bubble tree T for each vertex v inserted into face t. This can be done in parallel because each vertex v is inserted into a single, unique face t, and only one face is the outer face, so there will not be any conflicts.

Algorithm 2 shows the subroutine for building the bubble tree. On Line 2, we create a new bubble node b^*. On Line 3, we find the bubble b in T that t is in. This is the bubble that is created when t is created (in some previous call to UpdateBubbleTree). b^* is the bubble that the faces $\{v, v_x, v_y\}$, $\{v, v_y, v_z\}$, and $\{v, v_x, v_z\}$ are in. If t is the current outer face, this means we are inserting v into the outer face, and b is the current root of T. So on Lines 4–7, we let b's parent be b^*, and add b to b^*'s children. This maintains the invariant above because the vertices in the bubbles before inserting v are in the interior of the outer face, and after the procedure these bubbles become descendants of the edge corresponding to the outer face t. The OuterFace is then updated to be a face $\{v, v_x, v_y\}$ in the 4-clique corresponding to b^*. If t is not the current outer face, then we do not need to change the outer face, because the vertex is inserted into some inner face of b. In this case, on Lines 8–10, we let b^*'s parent be b, and add b^* to b's children. This maintains the invariant above because v must be in the interior of t and its ancestors, and the bubble b^* containing v is a descendant of the edge corresponding to t.

Building the bubble tree takes $O(n)$ work because we insert n vertices, and each insertion takes $O(1)$ work.

Example. We now give an example of the bubble tree construction by running our algorithm on the example TMFG in Figure 2. We will first describe inserting a single vertex, and then describe inserting multiple vertices in parallel. Suppose we start with the 4-clique $C = \{0, 1, 2, 4\}$. We have four faces $\{\{0, 1, 2\}, \{0, 1, 4\}, \{0, 2, 4\}, \{1, 2, 4\}\}$, where $t_1 = \{0, 1, 2\}$ is the outer face. We initialize the bubble tree T with node $b_1 = \{0, 1, 2, 4\}$, which corresponds to C. We insert vertices 3, 5, and 6, in order, into faces $\{0, 1, 2\}$, $\{1, 2, 3\}$, and $\{0, 1, 3\}$, respectively. We first insert 3 into $t = t_1 = \{0, 1, 2\}$. For this insertion, the new bubble b^* on Line 2 is $b_2 = \{0, 1, 2, 3\}$, and the b on Line 3 is b_1. Since t is the current outer face, we let b_2 be b_1's parent. The new outer face is $t_2 = \{0, 1, 3\}$. Now suppose we insert 5 and 6 into $\{1, 2, 3\}$ and $\{0, 1, 3\}$, respectively, in parallel. This is similar to inserting 3, and we add $b_3 = \{0, 1, 3, 6\}$ as b_2's parent and $b_4 = \{1, 2, 3, 5\}$ as b_2's child in parallel.

4.2 Directing Bubble Tree Edges

We now briefly describe how to direct the bubble tree edges after obtaining the undirected bubble tree from Algorithms 1 and 2, and how we significant improve the efficiency of this step over the original sequential DBHT algorithm. A more detailed description of our algorithm can be found in Appendix A. Each edge of the bubble tree corresponds to a separating triangle. The direction is decided by computing the sum over the weights of the edges in the TMFG connecting the triangle with its interior versus its exterior [21]. Figure 2(c) shows an example of directing the edges.

Let us denote the sum over the weights of the edges to the interior as inVal and to the exterior as outVal. In the original algorithm, the two values are computed by running a breadth-first-search

(BFS) on $G \setminus t$ for each separating triangle t to find out its exterior and interior, and then computing the sum of edge weights. This takes $\Theta(n^2)$ work because PMFGs and TMFGs have $\Theta(n)$ edges [21] and each BFS takes $\Theta(n)$ work.

In our bubble tree, the interior of a separating triangle contains all of the vertices in the descendants of the edge corresponding to this separating triangle, and the exterior contains all other vertices. Using this property, we can compute the direction of all edges in $\Theta(n)$ work using a recursive algorithm, starting from the root of bubble tree. At each bubble node b, we compute the direction of the edge from itself to its parent, and recursively call the bubble's children. Our algorithm is novel in that it takes only $\Theta(n)$ work because for each bubble, we do $O(1)$ work, and there are $\Theta(n)$ bubbles in total. This gives a significant improvement over the quadratic work of the original algorithm.

4.3 Building the Dendrogram

After computing the directed bubble tree, the DBHT algorithm computes two levels of discrete clusters (where each vertex is assigned to a unique cluster) and then builds a dendrogram [21]. A ***converging bubble*** is a bubble with only incoming edges, and no outgoing edges. The two levels of discrete clusters are computed by assigning vertices to bubbles using the similarity and dissimilarity measures supplied by the algorithm. The dissimilarity measure is used to compute the all-pairs shortest path distances between vertices in the TMFG. For the first level, each vertex is assigned to a converging bubble. For the second level of clusters, each vertex is assigned to a bubble, which is not necessarily a converging bubble. After the two levels of discrete clustering are computed, complete-linkage clustering is used to build the dendrogram within each discrete cluster and across different discrete clusters. More details of the parallelization of this algorithm can be found in Appendix A.

For example, in Figure 2(c), since there is only a single converging bubble b_2, all vertices are assigned to b_2 in the first level of clustering. The second level of clustering is $\{0, 3, 6\}$, $\{2, 4\}$, and $\{1, 5\}$. The red vertices are assigned to the bubble in which they are marked. We see that in Figure 2(d), the second-level clusters are merged together first, and then they are merged into the first-level cluster.

5 Experiments

Testing Environment. We perform experiments on a `c5.24xlarge` machine on Amazon EC2, with 2 Intel Xeon Platinum 8275CL (3.00GHz) CPUs for a total of 48 hyper-threaded cores, and 192 GB of RAM. By default, we use all cores with hyper-threading. For the C++ code tested, we use the `g++` compiler (version 7.5) with the `-O3` flag, and use ParlayLib [2] for parallelism in our code.

- ***PMFG-DBHT*** is the existing sequential PMFG and DBHT implementation in MATLAB, which we obtained online. The bottleneck of PMFG-DBHT is in constructing the PMFG. We tried implementing the same PMFG construction algorithm in C++, but found it slower than the MATLAB implementation, so we report the MATLAB runtime.
- ***SEQ-TDBHT*** is the state-of-art implementation of sequential TMFG and DBHT in MATLAB, which we obtained online.
- ***PAR-TDBHT*** is our implementation of parallel TMFG and DBHT in C++. We tested prefixes of size 1, 2, 5, 10, 30, and 50.
- ***COMP*** and ***AVG*** are the parallel C++ complete and average linkage implementations, respectively, by Yu et al. [30].
- ***K-MEANS*** is the k-means++ implementation from scikit-learn, parallelized using Cython and OpenMP.
- ***K-MEANS-S*** is the parallel k-means++ implementation from scikit-learn with a preprocessing step that computes a spectral embedding, which constructs its affinity matrix using a nearest-neighbors graph [10]. We choose the number of nearest neighbors that gives the best clustering quality. If the true number of clusters is c, then we project the original data onto the c-dimensional space, which we find to be a good heuristic for finding good clusters.

Evaluation. We evaluated the clustering quality using the Adjusted Rand Index (ARI) [7] and Adjusted Mutual Information (AMI) [26] scores. In our experiments, AMI showed similar trends as ARI, and so we only show the plots for ARI. Let n_{ij} be the number of objects in the ground truth cluster i and the cluster generated by the algorithm j, n_{i*} be $\sum_j n_{ij}$, n_{*j} be $\sum_i n_{ij}$, and n be $\sum_i n_{i*}$. The ARI is computed as $\frac{\sum_{i,j}\binom{n_{ij}}{2} - [\sum_i \binom{n_{i*}}{2} \sum_j \binom{n_{*j}}{2}]/\binom{n}{2}}{\frac{1}{2}[\sum_i \binom{n_{i*}}{2} + \sum_j \binom{n_{*j}}{2}] - [\sum_i \binom{n_{i*}}{2} \sum_j \binom{n_{*j}}{2}]/\binom{n}{2}}$. The ARI score is 1 for a perfect match, and its expected value is 0 for random assignments.

When computing the ARI for the hierarchical methods, we cut the dendrogram such that the number of resulting clusters is the same as the number of ground truth clusters. For the k-means methods, we set k to be equal to the number of ground truth clusters.

Data sets. We show results on 16 data sets from the UCR Time Series Classification Archive [4]. The data sets are summarized in Table 1. Since the UCR Archive is designed for classification tasks, not all data sets are suitable for clustering. We choose 16 data sets which have ARI scores of at least 0.25 for K-MEANS.

We also collected the closing daily prices of 1614 US stocks between January 1, 2013 and January 1, 2019 (1761 trading days) using the Yahoo Finance API. We used the Industry Classification Benchmark (ICB) to obtain ground truth clusters for this data set. Similarly, we collected the closing daily prices of 490 stocks from the same time period from the Toronto Stock Exchange (TSX), and used the Yahoo Sector from Yahoo Finance API for ground truth clustering.

We used the Pearson correlation coefficient p for the similarity measure and $d = \sqrt{2(1-p)}$ for the dissimilarity measure [13]. For normalized and zero-mean vectors, this dissimilarity measure is the same as the squared Euclidean distance.

5.1 Runtime

We show the running times of all hierarchical clustering algorithms and data sets in Figure 3 (sequential times are in the top plot and parallel times are in the bottom plot). PAR-TDBHT-1 is PAR-TDBHT with a prefix of size 1 and PAR-TDBHT-10 is PAR-TDBHT with a prefix of size 10 (we chose this prefix size for most experiments as it gives a good tradeoff between speed and cluster quality). Since PMFG-DBHT and SEQ-TDBHT are sequential, we only include it in the top plot.

We see that PMFG-DBHT and SEQ-TDBHT are both orders of magnitude slower than all other methods. PMFG-DBHT is 416.9–12075.3x slower than PAR-TDBHT-1 and 378.41–11520.85x slower than PAR-TDBHT-10 on a single thread. SEQ-TDBHT is 39.75–208.90x slower than PAR-TDBHT-1 and 37.66–189.87x slower than PAR-TDBHT-10

Table 1: Summary of UCR data sets used in the experiments. n is the number of objects, and L is the length or size of the object.

ID	Name	n	L	# of classes
1	Mallat	2400	1024	8
2	UWaveGestureLibraryAll	4478	945	8
3	NonInvasiveFetalECGThorax2	3765	750	42
4	MixedShapesRegularTrain	2925	1024	5
5	MixedShapesSmallTrain	2525	1024	5
6	ECG5000	5000	140	5
7	NonInvasiveFetalECGThorax1	3765	750	42
8	MoteStrain	1272	84	2
9	HandOutlines	1370	2709	2
10	UWaveGestureLibraryX	4478	315	8
11	CBF	930	128	3
12	InsectWingbeatSound	2200	256	11
13	UWaveGestureLibraryY	4478	315	8
14	ShapesAll	1200	512	60
15	SonyAIBORobotSurface2	980	65	2
16	FreezerSmallTrain	2878	301	2

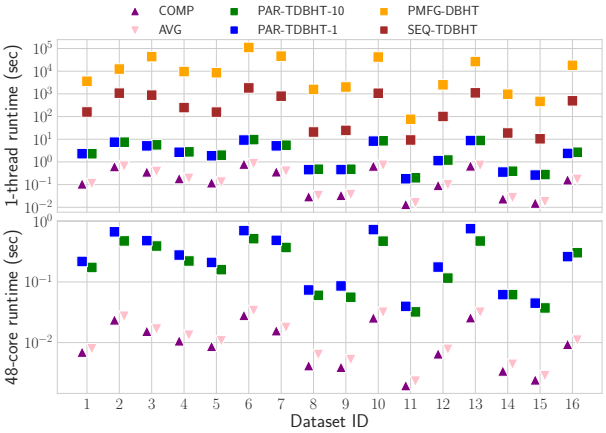

Figure 3: Time (seconds) of different methods on UCR data sets. The top plot shows run times on a single thread and the bottom plot shows run times on 48 cores with hyper-threading.

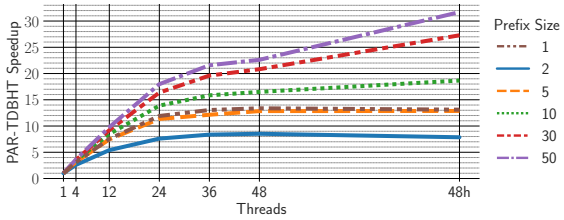

Figure 4: Self-relative parallel speedup vs. thread counts for PAR-TDBHT with different prefix sizes on the ECG5000 data set. "48h" indicates 48 cores with two-way hyper-threading.

on a single thread. We discuss the reasoning for this speedup in the Runtime Decomposition section below. PAR-TDBHT-1 and PAR-TDBHT-10 are slower than AVG and COMP, which is expected because DBHT uses complete-linkage clustering as a subroutine. However, we will see in Section 5.2 that PAR-TDBHT-1 and PAR-TDBHT-10 give significantly better clusters than AVG and COMP on most data sets.

K-MEANS-S and K-MEANS are not plotted because they do not generate a dendrogram, and one would need to run the algorithm multiple times to obtain clusters of different scales. On average

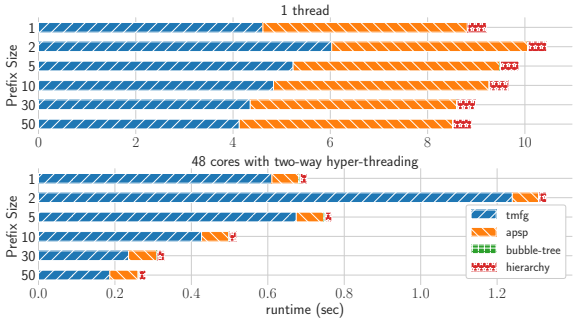

Figure 5: Breakdown of runtime across different steps of our algorithm on ECG5000. "tmfg" corresponds to TMFG construction (Algorithm 1); "apsp" corresponds to the all-pairs shortest paths computation; "bubble-tree" corresponds to computing the direction of bubble tree edges and assigning vertices to bubbles; and "hierarchy" corresponds to running the complete-linkage subroutine.

across the data sets, one run of K-MEANS is 1.82x slower than PAR-TDBHT-1 and 2.26x slower than PAR-TDBHT-10 on 48 cores with hyper-threading, and one run of K-MEANS-S is 4.94x slower than PAR-TDBHT-1 and 6.03x slower than PAR-TDBHT-10. We used the 12-thread times for K-MEANS and K-MEANS-S because they became slower past 12 threads due its parallel overheads.

Scalability with Thread Count. In Figure 4, we show the scalability of our algorithm vs. thread count on the ECG5000 data set. PAR-TDBHT with a prefix size of 50 achieves a self-relative speedup of 31.7x on 48 cores with two-way hyper-threading. In general, a larger prefix size results in higher scalability, because more insertions in the TMFG construction can be processed in parallel. However, using a prefix of size 2 is actually slower than using a prefix of size 1, which is the exact TMFG, for the following reason. When we only have a prefix of size 1, our implementation only needs to find the best vertex-face pair to insert using a parallel maximum, but when the prefix size is larger than 1, the algorithm needs to first sort all vertex-face pairs, and then find a prefix of the sorted pairs to insert. A prefix of size 2 is small, so there is not enough additional parallelism to offset the overheads of sorting. In our experiments, using a prefix of size 5 or larger gives better runtimes than using exact TMFG.

Scalability with Data Size. We observe that on the UCR data sets, the PAR-TDBHT runtimes scale with the data size n approximately as a function of $O(n^{2.27})$ on a single thread and $O(n^{1.77})$ on 48 cores with two-way hyper-threading. The scaling for parallel runtimes is better than for the single-threaded runtimes because we get more parallel speedups for larger values of n.

Runtime Decomposition. We show the breakdown of runtime across different steps of our algorithm in Figure 5 on the ECG5000 data set. (the different steps are described in the figure caption). Sequentially, the majority of the runtime is in the "tmfg" and "apsp" steps, while in parallel the majority of the runtime is in the "tmfg" step, especially for small prefix sizes where TMFG construction has limited parallelism. When the prefix size is larger, the runtime of the "tmfg" step is significantly shorter. Our "bubble-tree" for TMFG is very fast and the runtime is too small to be visible in the plot.

For comparison, SEQ-TDBHT requires 628.49s for "tmfg", 9.07s for "apsp", 68.99s for "bubble-tree", and 1135.78s for "hierarchy" on the same data set, and thus all steps of PAR-TDBHT, even on one thread,

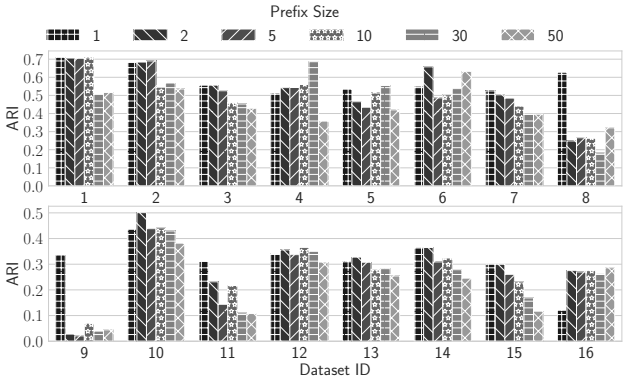

Figure 6: Clustering quality (ARI score) of PAR-TDBHT. Different shades represent different prefix sizes.

are faster than SEQ-TDBHT. Our "tmfg" step is faster because we use optimized data structures to update the gain table that do not require looping over all faces. The "apsp" step is faster because we used a different shortest paths algorithm. SEQ-TDBHT uses Johnson's algorithm from the Boost Graph Library, while we used the faster Dijkstra's algorithm from the same library. Our "bubble-tree" is step is faster because we optimized it for the TMFG topology. While this step is much slower than "apsp" in SEQ-TDBHT, its time becomes negligible in PAR-TDBHT. Our "hierarchy" step is faster because we use the optimized complete-linkage algorithm by Yu et al. [30], whereas SEQ-TDBHT uses a less efficient implementation. Across all data sets, our "tmfg" step is 23.11–3368.37x faster than SEQ-TDBHT and the remaining steps are together 1274.97–13513.88x faster than SEQ-TDBHT on 48 cores with hyper-threading. On a single thread, our "tmfg" step is 8.28–152.24x faster than SEQ-TDBHT and the remaining steps are together 67.29–441.30x faster than SEQ-TDBHT.

5.2 Clustering Quality

Prefix Size and Clustering Quality. Our prefix-based TMFG algorithm is able to produce graphs with weight very close to, and sometimes even higher than, the sequential TMFG and PMFG algorithms. In our experiments, our prefix-based TMFG algorithm produces graphs with edge weight sums that are 97.1–100.3% of edge weight sums produced by PMFG.

We present the ARI of using prefix sizes in Figure 6. We found that PAR-TDBHT with a prefix of size greater than 1 gives similar, and sometimes even better ARI than using a prefix of size 1 (which correponds to using the exact TMFG). Generally, using a larger prefix size results in lower ARI, but sometimes a larger prefix size can result in better quality as the clusters could become less sensitive to noise. For smaller data sets (e.g., data sets 8, 9, 11, and 15), the ARI degradation is larger. This is because the prefix is a larger percentage of all edges in the filtered graph. For larger data sets (e.g., data sets 2, 6, 10, and 13), the ARI degradation is smaller.

Hierarchical Methods. We show the clustering quality of all of the methods and data sets in Figure 7. We see that PAR-TDBHT-1 and PAR-TDBHT-10 often generates higher-quality clusters than both of the other hierarchical clustering algorithms, COMP and AVG. On data sets where the number of ground truth clusters is very small, such as data sets 8, 9, 15, and 16, COMP and AVG have low ARI score. This is because COMP and AVG are sensitive to agglomeration

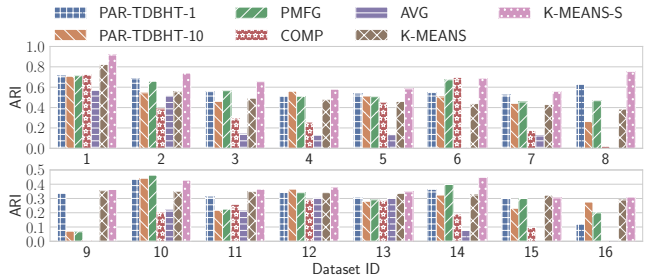

Figure 7: Clustering quality of different methods on UCR data sets. A few bars for COMP and AVG are hard to observe because their ARIs are close to 0.

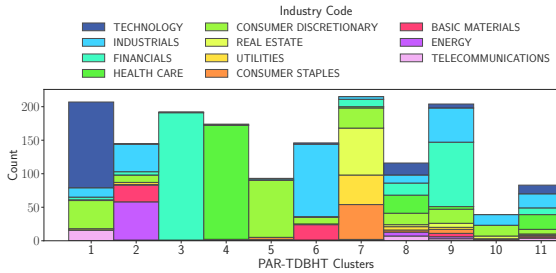

Figure 8: Clustering result on the US stock data set compared to ground truth using PAR-TDBHT with a prefix of size 30.

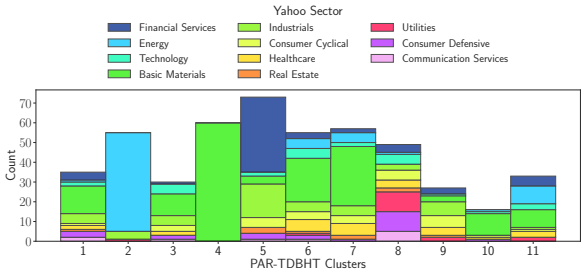

Figure 9: Clustering result on the TSX stock data set compared to ground truth using PAR-TDBHT with a prefix of size 5.

decisions, which only use local information, and when these decisions are wrong, they lead to poor ARI scores. On the other hand, DBHT's topological constraints (bubble and converging bubble) uses global information, which makes them less sensitive to wrong agglomeration decisions. DBHT can also suffer from the sensitivity of agglomeration decisions when the global information is not captured correctly (e.g., in data set 9).

***k*-Means.** PAR-TDBHT-1 and PAR-TDBHT-10 generate clusters of similar quality to K-MEANS across all data sets. However, K-MEANS is slower and also does not produce a dendrogram. If we want to find clusters at different scales, then we have to run the algorithm multiple times, which results in even longer running times.

Using the spectral embedding preprocessing step boosts the K-MEANS method to achieve the best quality on most data sets. However, we show in Appendix B that the quality of K-MEANS-S is highly sensitive to the number of nearest neighbors, and this parameter is hard to choose apriori. We also tried PAR-TDBHT on the embedded data sets, and found the ARI scores to be similar to those of the non-embedded data sets.

Example: Clustering Stocks. Figure 8 shows the clusters produced by PAR-TDBHT with a prefix size of 30 on the US stock data

set. We preprocess the daily stock prices by computing the detrended daily log-return using the method by Musmeci et al. [16]. We then compute a spectral embedding of the normalized detrended daily log-returns. Finally, we compute the Pearson correlation of the embedded data and run PAR-TDBHT on the correlation matrix. We see that PAR-TDBHT is able to accurately cluster the "financial" stocks, "health care" stocks, and "consumer discretionary" stocks. There is also a cluster with mostly "technology" stocks, and a cluster with mostly "industrials" stocks. We also see that almost all of the "energy", "utilities", "consumer staples", and "real estate" stocks are clustered together. The ARI score of this clustering is 0.36. In comparison, the ARI score of the exact TMFG clustering is 0.28.

We repeat the experiment on the TSX stock data set. In Figure 9, we show clustering result of PAR-TDBHT with a prefix size of 5. (we used a smaller prefix size than the size for US stocks because the TSX dataset is smaller). We see that our algorithm is able to find a cluster (cluster 2) that corresponds to energy stocks, and a cluster (cluster 4) that corresponds to industrial stocks. The ARI score of this clustering is 0.21. In comparison, the ARI score of the exact TMFG clustering is 0.25.

Time and Quality Trade-off. PAR-TDBHT provides a good trade-off between runtime and clustering quality. It is faster than the K-MEANS methods, and is able to produce clusters of similar quality. Although PAR-TDBHT is slower than AVG and COMP, its clustering quality is more stable and in most cases better. Furthermore, it is able to finish within 1.5 seconds for all of the data sets.

6 Related Work

The most popular hierarchical clustering algorithm variants are hierarchical agglomerative clustering algorithms (HAC) [5, 15, 28, 30]. HAC works by merging the closest pair of clusters in each round. Different variants of HAC use different linkage functions to compute the distance between clusters. Popular linkage functions include complete, average, single, centroid, and median linkage.

Song et al. [21] design a sequential hierarchical clustering algorithm called the directed bubble hierarchy tree (DBHT) using a graph-theoretic approach. The input graph to DBHT that Song et al. [21] use is the PMFG, and the DBHT construction is based on the observation that a maximal planar graph is the union of a set of special subgraphs called "bubbles" [20].

The PMFG is an approximate solution to the weighted maximal planar graph (WMPG) problem, which is NP-hard [6]. There have also been other approximate solutions designed for the WMPG. Massara et al. [14] propose the TMFG, which is based on local topological moves. Kataki et al. [9] design a heuristic that is based on a transformation to the connected spanning subgraph problem and the dual graph. Osman et al. [17] use a greedy random adaptive search procedure to solve the WMPG problem.

Besides filtering graphs based on topological constraints, there are other graph filtering methods that are used for clustering. For example, Ma et al. [11] use a low-pass filter to extract useful data representations for subspace clustering, and Tremblay et al. [23] use graph filtering for faster spectral clustering.

7 Conclusion

We designed new parallel algorithms for constructing TMFGs and DBHTs. We showed that our algorithms are faster than k-means based algorithms while giving similar clustering quality, and that although our algorithms are slower than complete and average linkage clustering, they are usually able to produce better clusters.

References

[1] Tomaso Aste. 2022. DBHT. [MATLAB Central File Exchange]https://www.mathworks.com/matlabcentral/fileexchange/46750-dbht.

[2] Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib-A toolkit for parallel algorithms on shared-memory multicore machines. In *SPAA*. 507–509.

[3] Matthew J Burton et al. 2015. Pathogenesis of progressive scarring trachoma in Ethiopia and Tanzania and its implications for disease control: two cohort studies. *PLoS Neglected Tropical Diseases* 9, 5 (2015), e0003763.

[4] Hoang Anh Dau et al. 2018. The UCR Time Series Classification Archive. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/.

[5] Laxman Dhulipala, David Eisenstat, Jakub Lacki, Vahab S. Mirrokni, and Jessica Shi. 2021. Hierarchical Agglomerative Graph Clustering in Nearly-Linear Time. In *ICML*. 2676–2686.

[6] John Giffin. 1984. Graph theoretic techniques for facilities layout. (1984).

[7] Lawrence Hubert and Phipps Arabie. 1985. Comparing partitions. *Journal of Classification* 2, 1 (1985), 193–218.

[8] Joseph Jaja. 1992. *Introduction to Parallel Algorithms*.

[9] Paulo AG Kataki, Márcia R Cappelle, Les R Foulds, and Humberto J Longo. 2020. A new algorithm for the Maximum-weight Planar Subgraph Problem. *Anais do LII Simpósio Brasileiro de Pesquisa Operacional, João Pessoa-PB* (2020).

[10] Małgorzata Lucińska and Sławomir T Wierzchoń. 2012. Spectral clustering based on k-nearest neighbor graph. In *IFIP CISIM*. 254–265.

[11] Zhengrui Ma, Zhao Kang, Guangchun Luo, Ling Tian, and Wenyu Chen. 2020. Towards Clustering-friendly Representations: Subspace Clustering via Graph Filtering. In *ACM MM*. 3081–3089.

[12] R. N. Mantegna. 1999. Hierarchical structure in financial markets. *The European Physical Journal B - Condensed Matter and Complex Systems* 11, 1 (1999), 193–197.

[13] Gautier Marti, Frank Nielsen, Mikołaj Bińkowski, and Philippe Donnat. 2021. A review of two decades of correlations, hierarchies, networks and clustering in financial markets. *Progress in Information Geometry* (2021), 245–274.

[14] Guido Previde Massara, Tiziana di Matteo, and Tomaso Aste. 2017. Network Filtering for Big Data: Triangulated Maximally Filtered Graph. *J. Complex Networks* 5 (2017), 161–178.

[15] Daniel Müllner. 2013. fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python. *Journal of Statistical Software* 53, 9 (2013), 1–18.

[16] Nicolo Musmeci, Tomaso Aste, and Tiziana Di Matteo. 2015. Relation between financial market structure and the real economy: comparison between clustering methods. *PloS One* 10, 3 (2015), e0116201.

[17] Ibrahim H Osman, Baydaa Al-Ayoubi, and Musbah Barake. 2003. A greedy random adaptive search procedure for the weighted maximal planar graph problem. *Computers & Industrial Engineering* 45, 4 (2003), 635–651.

[18] Jianhua Ruan, Angela K. Dean, and Weixiong Zhang. 2010. *BMC Systems Biology* 4, 1 (2010), 8.

[19] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing Contention Through Priority Updates. In *ACM Symposium on Parallelism in Algorithms and Architectures*. 152–163.

[20] Won-Min Song, Tiziana Di Matteo, and Tomaso Aste. 2011. Nested hierarchies in planar graphs. *Discrete Applied Mathematics* 159, 17 (2011), 2135–2146.

[21] Won-Min Song, Tiziana Di Matteo, and Tomaso Aste. 2012. Hierarchical Information Clustering by Means of Topologically Embedded Graphs. *PloS One* 7 (03 2012), e31929.

[22] Won-Min Song and Bin Zhang. 2015. Multiscale embedded gene co-expression network analysis. *PLoS Computational Biology* 11, 11 (2015), e1004574.

[23] Nicolas Tremblay, Gilles Puy, Pierre Borgnat, Rémi Gribonval, and Pierre Vandergheynst. 2016. Accelerated spectral clustering using graph filtering of random signals. In *ICASSP*. 4094–4098.

[24] Michele Tumminello, Tomaso Aste, Tiziana Di Matteo, and Rosario Mantegna. 2005. A tool for filtering information in complex systems. *PNAS* 102 (08 2005), 10421–10246.

[25] Michele Tumminello, Claudia Coronnello, Fabrizio Lillo, Salvatore Micciche, and Rosario Mantegna. 2007. Spanning Trees and Bootstrap Reliability Estimation in Correlation-Based Networks. *I. J. Bifurcation and Chaos* 17 (07 2007), 2319–2329.

[26] Nguyen Xuan Vinh, Julien Epps, and James Bailey. 2010. Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance. *J. Mach. Learn. Res.* 11 (dec 2010), 2837–2854.

[27] Gang-Jin Wang, Chi Xie, and Shou Chen. 2017. Multiscale correlation networks analysis of the US stock market: a wavelet analysis. *Journal of Economic Interaction and Coordination* 12, 3 (2017), 561–594.

[28] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. Fast Parallel Algorithms for Euclidean Minimum Spanning Tree and Hierarchical Spatial Clustering. In *SIGMOD*. 1982–1995.

Algorithm 3: ComputeDirection

```
1  Function ComputeDirection(bubble tree node b):
2      if b has a parent then
3          {v_x, v_y, v_z} = vertices shared by b and its parent
4          v = b \ {v_x, v_y, v_z}
5          r = {}
6          r[v_x] = w(v_x, v), r[v_y] = w(v_y, v), r[v_z] = w(v_z, v)
7          parallel_for b* ∈ b's children do
8              r* = {v*_x : val*_x, v*_y : val*_y, v*_z : val*_z} =
                 computeDirection(b*)
9              if v*_x ∈ r then WriteAdd(r[v*_x], val*_x)
10             if v*_y ∈ r then WriteAdd(r[v*_y], val*_y)
11             if v*_z ∈ r then WriteAdd(r[v*_z], val*_z)
12         inVal = r[v_x] + r[v_y] + r[v_z]
13         outVal = deg(v_x) + deg(v_y) + deg(v_z) − inVal −
             2(w(v_x, v_y) + w(v_x, v_z) + w(v_y, v_z))
14         if inVal > outVal then
15             Direct the edge from b's parent to b
16         else
17             Direct the edge from b to b's parent
18         return r
19     else /* b is the root                                   */
20         parallel_for b* ∈ b's children do
21             computeDirection(b*)
22         return {}
```

Algorithm 4: Parallel DBHT for TMFG

```
   Input: n × n dissimilarity matrix D, weighted undirected planar graph G
          generated by TMFG, bubble tree T
1  computeDirection(root of T)
2  Initialize fields v.g = (−∞, −∞), v.q = (−∞, −∞) for each v
3  BB = set of all bubble tree nodes
4  cvgBB = set of all bubble tree nodes with out-degree 0
5  parallel_for b ∈ bubbles do
6      Run directed BFS in T from b
7      Record for vertices in b which converging bubbles they can reach
   // all-pairs shortest paths in G with edge weights from D
8  A = allPairsShortestPaths(G, D)
   // assign vertices to converging bubbles
9  parallel_for b ∈ cvgBB do
10     parallel_for v ∈ b do
11         χ = computeChi(v, b)
12         WriteMax(v.g, (χ,b))
13 For each bubble b, let V^0_b be the vertices that have been assigned to b
14 Set v.g = (∞, ∞) for all v that is not in any converging bubble
15 parallel_for b ∈ cvgBB do
16     parallel_for v ⇀ b and have not been assigned a cluster do
17         L̄ = computeAverageShortestPath(v, V^0_b)
18         WriteMin(v.g, (L̄, b))
   // assign vertices to bubbles
19 parallel_for b ∈ BB do
20     χ_total = 0
21     for v ∈ b do
22         χ_v = computeChi'(v, b); χ_total += χ_v
23     for v ∈ b do
24         WriteMax(v.q, (χ_v/χ_total, b))
   // build hierarchy using complete linkage
25 {Z_1, ..., Z_n} /* initialize dendrogram nodes          */
26 parallel_for (b^c, b) ∈ BB × cvgBB do
27     Let Z_(b^c,b) = completeLinkage({Z_v : v.q = b ∧ v.g = b^c})
28 parallel_for b^c ∈ cvgBB do
29     Let Z_{b^c} = completeLinkage({Z_(b^c',*) : b^c' = b^c})
30 Z = completeLinkage({Z_{b^c} : b^c ∈ cvgBB})
31 dendrogram = computeHeight(Z)
32 return dendrogram
```

[29] Peter Tsung-Wen Yen and Siew Ann Cheong. 2021. Using topological data analysis (TDA) and persistent homology to analyze the stock markets in Singapore and Taiwan. *Frontiers in Physics* (2021), 20.

[30] Shangdi Yu, Yiqiu Wang, Yan Gu, Laxman Dhulipala, and Julian Shun. 2021. ParChain: A Framework for Parallel Hierarchical Agglomerative Clustering using Nearest-Neighbor Chain. *PVLDB* (2021).

A Details on DBHT

A.1 Directing Bubble Tree Edges

We now describe how to direct the bubble tree edges after obtaining the undirected bubble tree from Algorithms 1 and 2. Each edge of the bubble tree corresponds to a separating triangle. The direction is decided by computing the sum over the weights of the edges in the TMFG connecting the triangle with its interior versus its exterior [21]. Let us denote the sum over the weights of the edges to the interior as inVal and to the exterior as outVal. In the original sequential DBHT algorithm, the two values are computed by running a breadth-first-search (BFS) on $G \setminus t$ for each separating triangle t to find out its exterior and interior, and then computing the sum of edge weights. This takes $\Theta(n^2)$ work because PMFGs and TMFGs have $\Theta(n)$ edges [21] and each BFS takes $\Theta(n)$ work.

In our bubble tree, the interior of a separating triangle contains all of the vertices in the descendants of the edge corresponding to this separating triangle, and the exterior contains all other vertices. We will show that using this property, we can compute the direction of all edges in $\Theta(n)$ work by recursively calling the ComputeDirection function, shown in Algorithm 3, with the root of bubble tree as the argument to the initial call.

At each bubble node b, we compute the direction of the edge from itself to its parent. If b is the root, then it has no edge to its parent, and so we do not need to compute anything. We only need to initialize the computation on its children (Lines 20–22). Otherwise, we compute the inVal and outVal for the separating triangle corresponding to the edge from the bubble to its parent. If inVal > outVal, then the edge goes from b's parent to b, and vice versa (Lines 14–17).

On Lines 3–4, we obtain the vertices $\{v_x, v_y, v_z\}$ in the separating triangle represented by the edge from b to its parent, and let v be the remaining vertex in the bubble. On Lines 5–6, we initialize the sum of edges to the interior from each corner of the triangle with the edge weights from each corner to v since v is in the interior. Then, on Lines 7–11 we recursively, and in parallel, compute the sum of edge weights from the corners of b's children to the children's interior. Note that this computation is nested parallel, meaning that parallel tasks are recursively created. Since the children's interior is also b's interior, we can sum the weights of b's corners obtained from its children to compute the inVal of b. We do so by using WriteAdd to allow for concurrent updates. On Line 13, we compute outVal by subtracting the inVal and the edge weights of the triangle from the weighted degrees of the corners of the triangle. On Line 18, we return the weights at the corners to b's parent. This takes $\Theta(n)$ work because for each bubble, we do $O(1)$ work, and there are $\Theta(n)$ bubbles in total.

Now, we describe the derivation of the formula on Line 13 of Algorithm 3. Let $D = deg(v_x) + deg(v_y) + deg(v_z)$, I_k be the total weighted degree of v_k to the interior, and O_k be the total weighted degree of v_k to the exterior. We can rewrite D as follows:

$$\begin{aligned} D &= I_x + O_x + w(v_x, v_y) + w(v_x, v_z) + I_y + O_y + w(v_y, v_x) + w(v_y, v_z) \\ &\quad + I_z + O_z + w(v_z, v_x) + w(v_z, v_y) \\ &= (I_x + I_y + I_z) + (O_x + O_y + O_z) \\ &\quad + 2(w(v_x, v_y) + w(v_x, v_z) + w(v_y, v_z)) \\ &= \text{inVal} + \text{outVal} + 2(w(v_x, v_y) + w(v_x, v_z) + w(v_y, v_z)) \end{aligned}$$

Rearranging gives the following formula for computing OUTVAL:
$\text{OUTVAL} = D - \text{INVAL} - 2(w(v_x, v_y) + w(v_x, v_z) + w(v_y, v_z))$

Example. We continue to our example for Algorithm 3. Given the rooted bubble tree in Figure 2(b), we start our computation at the root b_3. Since b_3 does not have a parent, we recurse down to its child b_2. For $b = b_2$, the vertices shared with its parent are $\{v_x, v_y, v_z\} = t_2 = \{0, 1, 3\}$ and the remaining vertex is $v = 2$. Then, we initialize $r[0] = w(0, 2)$, $r[1] = w(1, 2)$, and $r[3] = w(3, 2)$ on Line 6. Next, we recurse down to b_2's children, b_1 and b_4. For b_1, the resulting r^* array should contain $r^*[0] = w(0, 4)$, $r^*[1] = w(1, 4)$, and $r^*[2] = w(2, 4)$ from the recursion on Line 8 because the shared vertices with its parent are $t_1 = \{0, 1, 2\}$ and the remaining vertex is 4. On Lines 9–11, since $v_x^* = 0 \in r$ and $v_y^* = 1 \in r$, we increment $r[0]$ by $w(0, 4)$ and $r[1]$ by $w(1, 4)$. Since $v_z^* = 2 \notin r$, we do not process it. Now $r[0] = w(0, 4) + w(0, 2)$ and $r[1] = w(1, 4) + w(1, 2)$. Similarly for b_4, we increment $r[1]$ by $w(1, 5)$ and $r[3]$ by $w(3, 5)$. Now $r[0] = w(0, 4) + w(0, 2)$, $r[1] = w(1, 4) + w(1, 2) + w(1, 5)$ and $r[3] = w(3, 2) + w(3, 5)$.

When we get to Line 12 for b_2, the sum of $r[0]$, $r[1]$, and $r[3]$ is INVAL because it contains the edge weights from the three corners of t_2 to its interior. OUTVAL is computed by summing the weights of all edges from t_2 and then subtracting INVAL and the weight of t_2's edges (once in each direction). In our example, we find that INVAL > OUTVAL, and so the edge is directed from b_3 to b_2 (Figure 2(c)). The other edges are directed similarly by comparing INVAL and OUTVAL.

A.2 Assigning Vertices

In this subsection, we first describe the assignment rules of the DBHT algorithm, and then describe our parallel algorithm for computing the assignment.

Intuitively, converging bubbles are the "end points of a directional path that follows the strongest connections" [21], so they are considered the center of local clusters. The first level of discrete clustering assigns each vertex to a unique converging bubble. If a vertex is in at least one converging bubble, then it is assigned to the converging bubble with the strongest attachment $\chi(v, b) = \frac{\sum_{u \in b} w(u,v)}{3(|b|-2)}$, where $3(|b| - 2)$ is the number of edges in the bubble b. For TMFG, all bubbles have the same size, and so we can simplify it to $\chi(v, b) = \sum_{u \in b} w(u, v)$. For a vertex that is not in any converging bubble, it is assigned to the converging bubble that has the minimum mean average shortest path distance:

$$\bar{L}(v, b) = mean\{l_D(u, v) \mid u \in V_b^0 \wedge v \rightharpoonup b\}$$

where $v \rightharpoonup b$ means vertex v is in some bubble that can reach b in the directed bubble tree, and $l_D(u, v)$ is the shortest path distance from u to v in the TMFG using the dissimilarity measure D as the edge weights. For all bubbles b, we let V_b^0 be the set of vertices in converging bubbles that have already been assigned to b from computing χ. Let vertices assigned to the same converging bubble b in the procedure above be a ***group***.

After this initial partitioning of vertices, we investigate how each of these groups is internally structured by performing a second level of discrete clustering. This time we assign each vertex to a unique bubble, but not necessarily a converging bubble. A vertex v is assigned to the bubble b that maximizes a different attachment score

$$\chi'(v, b) = \frac{\sum_{u \in b} w(u, v)}{\sum_{u', v' \in b} w(u', v')}.$$

The pseudocode for our parallel DBHT algorithm is shown in Algorithm 4. In the pseudocode, WRITEMIN is a priority concurrent write that takes as input two arguments, where the first argument is the location to write to and the second argument is the value to write; on concurrent writes, the smallest value is written [19]. WRITEMAX is similar, but writes the largest value. On Lines 1–24, we show how the two assignments can be computed in parallel. We first compute the necessary auxiliary data used in the vertex assignment computation. On Line 1, we compute the directions of bubble tree edges using Algorithm 3. On Line 2, we initialize the fields g and q for each vertex to $(-\infty, -\infty)$ to prepare for the WRITEMAX operation on them. g is the group assignment and q is the bubble assignment. On Line 4, we obtain all of the converging bubbles by using a parallel filter based on the out-degree of the tree nodes. On Lines 5–7, we run a BFS in the directed bubble tree for each bubble in parallel, and record for vertices in the bubbles which converging bubbles they can reach. This helps us to do the $v \rightharpoonup b$ computation later. On Line 8, we compute all-pairs shortest paths in the TMFG between vertices by running Dijkstra's shortest path algorithm from each bubble node in parallel.

Now that we have obtained all necessary auxiliary data, we start to compute the assignments. On Lines 9–18, we compute the vertex assignments to converging bubbles, i.e., the groups. First, we compute the group assignment for vertices in at least one converging bubble. In parallel for all converging bubbles b and all vertices v in the bubble, we compute the attachment score $\chi(v, b)$, and then use WRITEMAX to write the converging bubble with the largest attachment to each vertex's assignment (Lines 9–12). On Line 13, we compute V_b^0 for all converging bubbles b by first parallel integer sorting the vertices by group assignment, and then using a parallel filter to obtain the start and end indices of the groups in the sorted vertex array. On Line 14, we set $v.g = (\infty, \infty)$ for all v whose group g has not been assigned yet to prepare for using a WRITEMIN. Then, in parallel we loop over all converging bubbles b and all vertices v such that $v \rightharpoonup b$ and v has not been assigned to any converging bubbles yet. In the loop, we compute the mean average shortest path distance $\bar{L}$ and use a WRITEMIN to write the converging bubble with the smallest $\bar{L}$ to each vertex's assignment. Next, on Lines 19–24, we compute the bubble assignment. This is similar to Lines 9–12, except that we keep track of the total edge weights in bubbles using the variable χ_{total}, and re-weight the attachments with χ_{total} when looking for the largest attachment bubble.

Lines 4 and 9–24 take $O(n)$ work because there are $O(n)$ bubbles, and each bubble has four vertices. The integer sort on Line 13 takes $O(n)$ work. Lines 5–7 take $O(n^2)$ work because we run $O(n)$ BFS's. Line 8 takes $O(n^2 \log n)$ work as we run Dijkstra's shortest path algorithm from each bubble node. Therefore, the total work is $O(n^2 \log n)$.

Example. We continue with our example in Figure 2(c). Here we only have a single converging bubble b_2 because this is the only bubble with no outgoing edges. Therefore all vertices are assigned to this converging bubble on Lines 9–18. Next, we look at what happens on Lines 19–24. Vertices 4, 5, and 6 are only in a single bubble, and so they are assigned to the bubble they are in. Vertices

0, 1, 2, and 3 are in multiple bubbles, and so we compute the χ' score and assign each vertex to the bubble with the maximum χ'. For example, for vertex 3, we will compute $\chi'(3, b_2)$, $\chi'(3, b_3)$, and $\chi'(3, b_4)$. In our example, we find that $\chi'(3, b_3)$ is the largest, and so we assign vertex 3 to b_3.

A.3 Complete Linkage

Now we describe how to obtain the dendrogram from the vertex assignments. At a high level, we build the complete-linkage hierarchy at three levels: intra-bubble, inter-bubble, and inter-group [21]. For the complete-linkage algorithm, the distance between two vertices u and v is their shortest path distance $l_D(u, v)$, and the distance between two set of vertices V_1 and V_2 is $d(V_1, V_2) = \max\{l_D(u, v) | u \in V_1, v \in V_2\}$. We use the parallel complete-linkage algorithm by Yu et al. [30].

The steps for building the dendrogram are shown on Lines 25–32 of Algorithm 4. On Line 25, we initialize n dendrogram nodes, one for each vertex in the TMFG. We define a ***subgroup*** to be the set of vertices in a group that belong to the same bubble. On Lines 26–27, for each subgroup we run the complete-linkage algorithm on the dendrogram nodes corresponding to vertices within the subgroup (each subgroup belongs in the Cartesian product BB × CVGBB). The result is a dendrogram for each subgroup. We use subgroups so that vertices within the same group, but in different bubbles will not be processed together, which is consistent with the sequential algorithm [21]. Then, on Line 29, we run the complete-linkage algorithm on dendrogram nodes in each group. The result is a dendrogram for each group. Finally, on Line 30, we run the complete-linkage algorithm on the group dendrogram nodes to obtain the final dendrogram.

Dendrogram Heights. In conventional complete-linkage clustering, the dendrogram height is the distance between the two merged clusters. The dendrogram's height requires that nodes closer to the dendrogram root (clusters merged later) have a larger height than nodes further away from the root. However, since we are concatenating three levels of dendrograms, the shortest path distance might not satisfy the height requirement. For example, the maximum shortest path distance might be larger between two vertices in a bubble than between two bubbles. As a result, we will have to re-assign heights to the dendrogram nodes. We use the same height assignment in the implementation by Aste [1], which ensures that all nodes that contain exactly one group are at the same height.

For inter-group dendrogram nodes merged on Line 30, the height is the number of converging bubbles in its descendants. This can be computed in the top-down traversal from the root $\mathcal{Z}$.

Intra-bubble and inter-bubble dendrogram nodes that belong to the same converging bubble b^c have heights are chosen from $[\frac{1}{n_b-1}, \frac{1}{n_b-2}, \ldots, \frac{1}{2}, 1]$, where n_b is the number of vertices assigned to converging bubble b^c. Each bubble's dendrogram nodes have a contiguous segment of the heights in the height list. To obtain the dendrogram height for descendants of each $\mathcal{Z}_{(b^c)}$, we sort these dendrogram nodes such that nodes merged on Line 27 appear before nodes merged on Line 29. Nodes merged on Line 29 are sorted by the distance when they are merged. Nodes merged on Line 27 are sorted first by bubble assignment and then by the distance when they are merged. Now all dendrogram nodes in the same bubble are contiguous. We then assign heights $[\frac{1}{n_b-1}, \frac{1}{n_b-2}, \ldots, \frac{1}{2}, 1]$ to the dendrogram nodes in the sorted order.

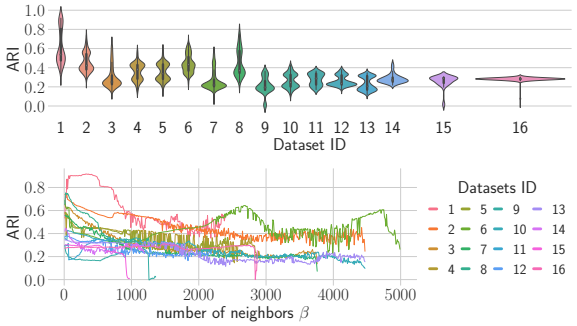

Figure 10: ARI of K-MEANS-S with different number of nearest neighbors on each data set. The top plot shows the distribution of ARI scores for different values of β. The bottom plot shows the ARI scores vs. the value of β, demonstrating the oscillating behavior.

Example. In Figure 2(c), since $\{2, 4\}$, $\{0, 3, 6\}$, and $\{1, 5\}$ are all assigned to the same bubble, we first have three dendrograms $\mathcal{Z}_{(b_2,b_1)}$, $\mathcal{Z}_{(b_2,b_3)}$, and $\mathcal{Z}_{(b_2,b_4)}$. Then we get a single dendrogram $\mathcal{Z}_{b_2}$ by running complete linkage on the three dendrogram roots. In this example, we only have a single converging bubble, and so we are done. If there were multiple converging bubbles, our algorithm would run complete linkage on the group dendrogram roots, which would all be at the same height 1, to produce the final dendrogram. The resulting dendrogram is shown in Figure 2(d).

B Additional Experiment Results

Limitation of K-MEANS-S. We show in Figure 10 that the quality of K-MEANS-S is highly sensitive to the choice of parameter β (the number of nearest neighbors). On the top plot of Figure 10, we see that for all data sets, using different values of β gives a wide range of ARI scores. On the bottom plot, we see that the ARI oscillates with different β for many data sets, and that the best β is very different for each data set. As a result, this parameter is hard to choose apriori.