A Scalable Algorithm for Single-Linkage Hierarchical Clustering on Distributed-Memory Architectures

William Hendrix[*] Northwestern University
Diana Palsetia[†] Northwestern University
Md. Mostofa Ali Patwary[‡] Northwestern University
Ankit Agrawal[§] Northwestern University
Wei-keng Liao[¶] Northwestern University
Alok Choudhary[‖] Northwestern University

Abstract

Hierarchical clustering is a fundamental and widely-used clustering algorithm with many advantages over traditional partitional clustering. Due to the explosion in size of modern scientific datasets, there is a pressing need for scalable analytics algorithms, but good scaling is difficult to achieve for hierarchical clustering due to data dependencies inherent in the algorithm. To the best of our knowledge, no previous work on parallel hierarchical clustering has shown scalability beyond a couple hundred processes. In this paper, we present PINK, a scalable parallel algorithm for single-linkage hierarchical clustering based on decomposing a problem instance into two different types of subproblems. Despite the heterogeneous workloads, our algorithm exhibits good load balancing, as well as low memory requirements and a communication pattern that is both low-volume and deterministic. Evaluating PINK on up to 6050 processes, we find that it achieves speedups up to approximately 6600.

Index Terms: I.5.3 [Information Systems Applications]: Clustering—Algorithms; D.1.3 [Programming Techniques]: Concurrent Programming—Parallel Programming;

1 Introduction

Hierarchical clustering is the problem of discovering the large-scale cluster structure of a dataset by forming a dendrogram that captures a full range of clustering behavior in the dataset, from the most general cluster that encompasses the entire dataset, to the most stringent clusters that only include a single data point each. Hierarchical clustering is a widely-used algorithm for evaluating the cluster structure of a dataset. Hierarchical clustering offers several advantages over partitional clustering in that the number of clusters does not need to be specified in advance and the structure of the resulting dendrogram can offer insight into the larger structure of the data, e.g., establishing a phylogenetic tree among a set of species. However, due to the high computational cost of hierarchical clustering, a scalable parallel approach is needed for clustering large datasets. Hierarchical clustering has been applied in document classification, bioinformatics, and chemoinformatics (e.g., [13, 24, 25]), as well as others. Despite its utility, hierarchical clustering is a challenging problem to parallelize, as the problem exhibits high data dependence—each level in the resulting dendrogram relies on all of the earlier levels to make sense. Moreover, many parallel algorithms for hierarchical clustering store the distance matrix for a dataset explicitly, limiting the size of the problem that can be tackled due to memory constraints. While other parallel hierarchical clustering algorithms exist, few of these algorithms have been evaluated on a large scale.

In this paper, we present PINK (parallel single linkage), a highly scalable parallel algorithm for single-linkage hierarchical clustering. As PINK does not explicitly store a distance matrix, it can be applied to much larger problem sizes and could be used as a component in a high performance, *in situ* data analytics pipeline. The overall strategy for PINK is to divide a large hierarchical clustering problem instance into a set of smaller sub-problems, calculate the hierarchical clustering dendrogram for each of these sub-problems, and reconstruct the solution for the original dataset by combining the solutions to the sub-problems. We evaluate our algorithm empirically on real and synthetic datasets with up to 5.2 million data points, and we find that it achieves an estimated speedup of up to 6596 on 6050 processes. The main contributions of this paper are:

- A scalable algorithm for parallelizing single-linkage hierarchical clustering
- A proof of correctness for our approach
- An empirical evaluation of the scalability of the computation, memory, and communication requirements for our algorithm on synthetic data
- Application of the algorithm to cluster cosmology data

The rest of the paper is organized as follows. We discuss related work in Section 2. We then describe our parallel algorithm and its correctness in Section 3 and 3.1, respectively. In Section 4, we present our scaling results on synthetic and cosmology data. Finally, we conclude in Section 5.

2 Related Work

One the most widely-used serial algorithms for the single-linkage hierarchical clustering (SHC) problem is SLINK [22]. SLINK calculates the dendrogram of the dataset by iterating through the data points one at a time, calculating the distance from the new data point to all of the previously added data points, and calculating the dendrogram for the extended dataset through the use of a pointer array (Π). The efficiency of SLINK is due to the fact that is calculates the distance between each pair of points just once, taking $O(n^2)$ time. Moreover, it has a space requirement of only $O(n)$ because it calculates and stores only a single row of the distance matrix at a time, making it ideal for solving larger hierarchical clustering problems.

One of the first parallel single-linkage hierarchical clustering algorithms was described by Bentley [3]. The algorithm was presented in terms of calculating the minimum spanning tree (MST) of a complete weighted graph with edge weights given by a distance function applied on the vertices, which is equivalent to solving the single-linkage hierarchical clustering problem. The algorithm assigns computation to various "tree" processors to calculate the MST.

[*]Email: whendrix@northwestern.edu
[†]Email: palsetia@u.northwestern.edu
[‡]Email: m-patwary@northwestern.edu
[§]Email: ankitag@eecs.northwestern.edu
[¶]Email: wkliao@ece.northwestern.edu
[‖]Email: choudhar@eecs.northwestern.edu

IEEE Symposium on Large Data Analysis and Visualization 2013
October 13 - 14, Atlanta, Georgia, USA
978-1-4799-1658-0/13/$31.00 ©2013 IEEE

Olson [19] presents two parallel algorithms for single-linkage clustering.The first of these merges clusters in order of increasing distance and is appropriate for a parallel RAM (PRAM, shared memory) architecture, while the second builds an MST starting from a random point and is appropriate for a parallel machine with a butterfly interconnection.

Hendrix et al. [15] present SHRINK, an algorithm for parallel single-linkage hierarchical clustering based on the SLINK algorithm [22]. SHRINK exhibits good scaling and communication behavior, but it trades repeated computation (up to double the original amount of work) for reduced communication cost. Moreover, the authors only evaluate SHRINK on up to 36 shared memory cores, achieving a speedup of roughly 19.

CLUMP [18] is a parallel clustering algorithm that related to single-linkage hierarchical clustering and minimum spanning tree problems. The authors report a speedup of up to 100 on 150 processors on a dataset of 1.2 million points; however, they claim that this result is very close to the maximum theoretical speedup for this size dataset, suggesting that this approach is somewhat limited in its scalability.

Dash et al. [9] present a parallel hierarchical algorithm based on partitioning the data points into groups based on a rectangular grid with overlapping cells. As pPOP computes a distance matrix for each cell, the algorithm has $O(n^2/c)$ space complexity, where n is the number of data points and c is the number of cells. The authors evaluate pPOP on up to 8 shared memory processors, achieving a speedup of about 5.7 on datasets of up to 30k points.

Du and Lin [12] present one of the few distributed parallel hierarchical clustering algorithms. Their algorithm, which relies on distributing the distance matrix and iteratively calculating the nearest pair of points, shows a speedup of 25 on 48 processors when clustering a microarray dataset with 277 data points in 7452 dimensions.

Johnson and Kargupta [16] describe an algorithm for the related problem of distributed hierarchical clustering, in which the data to be clustered exist at different sites with potentially different features, and the amount of communication between the data sites needs to be controlled. While their algorithm requires at most a constant factor overhead based on the number of sites across which the data is distributed, they do not present empirical results for their algorithm.

The design of PINK relies on the connection between single-linkage hierarchical clustering and the minimum spanning tree (MST) problem. This connection, discussed more thoroughly in [14], was also used in several of the preceding works. While much work has been done on parallel algorithms that explicitly calculate the MST of a graph, most present purely theoretical results (e.g., [7], [21], [20]) or focus on sparse graphs ([6], [1], [2]). However, Dehne and Götz [11] present and evaluate a distributed parallel algorithm for calculating the MST of dense graphs. The algorithm is based on dividing the edges of the graph, performing Borůvka's algorithm on the partitions, and combining the resulting MSTs. Dehne and Götz report a speedup of 3.5 on 4 processors on two graphs with 1k vertices and 400k edges.

3 Parallel Single-linkage Hierarchical Clustering

In this section, we detail our parallel algorithm for calculating the single-linkage hierarchical clustering (SHC) dendrogram. As we show formally in Section 3.1, calculating the SHC dendrogram of a dataset is equivalent to finding the minimum spanning tree (MST) of a complete weighted graph where the edge weights are given by the distance between the corresponding points.

The main parallelization strategy we employ in developing our parallel hierarchical clustering algorithm is to break a large problem instance into multiple subproblems of two different types, solve each sub-problem independently, and combine the subproblem solutions into a solution for the original problem instance. To form the subproblems, we first divide the data into several partitions (k). The first type of subproblem we form is composed of the bipartite complete subgraphs defined by every pair of partitions, while the second type consists of the complete subgraphs within each partition. In this way, we ensure that all edges (distances) are assigned to some subgraph. As each partition is roughly the same size, the complete bipartite graphs will contain slightly more than double the number of edges in the complete subgraphs. So, to achieve good load balance, we assign one complete bipartite subgraph or two complete subgraphs to each process. For an odd number of partitions, the "odd" complete subgraph will get assigned to a process by itself, so that one process may have a reduced workload. In total, partitioning the data into k partitions results in a total of $\left\lceil \frac{k^2}{2} \right\rceil$ distinct processes.

Algorithm 1: Outline of distributed single-linkage hierarchical clustering algorithm, PINK

```
1 Divide the data into $k$ subsets of the same size, $D_1, D_2, \ldots, D_k$.
2 Form $\binom{k}{2}$ subgraphs containing the complete bipartite graph
  on $(D_i, D_j)$ for each pair of subsets $D_i$ and $D_j$.
3 Form $\lceil k/2 \rceil$ subgraphs containing the union of the complete
  graphs for $D_{i-1}$ and $D_i$ for all even $i$. Also, if $k$ is odd, form a
  subgraph containing just the complete graph of $D_k$.
4 Have each process compute the MST of one subgraph using
  Prim's algorithm.
5 Sort the edges of each MST independently.
6 Combine the resulting MSTs by iteratively adding the edge
  with the minimum weight, eliminating edges that form a cycle.
  (Algorithm 2)
7 Repeat Step 6 until only one MST remains.
```

An illustration of this problem domain decomposition using $k = 4$ partitions (8 processes) appears in Figure 1. Note that while this decomposition partitions the edges of the graph, the data partitions will each be distributed to several processes at the outset of the algorithm. Also, the parallel problem decomposition reverts to a sequential MST (SHC) problem when we use $k = 1$ partition. An outline of our algorithm, which assumes that we have $p = \left\lceil \frac{k^2}{2} \right\rceil$ processes for some $k \geq 1$, appears in Algorithm 1.

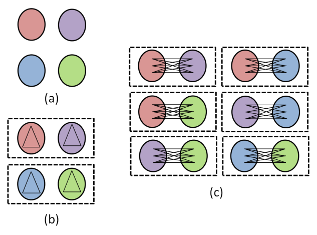

Figure 1: (a) Problem domain decomposition with $k = 4$ partitions; (b) Two processes are each assigned two complete subgraphs; (c) Six processes are assigned one bicomplete subgraph for the six pairs of partitions

In step 4 of Algorithm 1, we need to calculate the SHC dendrogram of a subproblem of the original data. As this dendrogram is equivalent to a MST of a related graph, we use Prim's algorithm to solve the subproblems. The advantages of Prim's are threefold. First, by using Prim's algorithm and finding an MST, rather than an algorithm specific to finding the SHC dendrogram, we can solve subproblems that are not complete weighted graphs; i.e., subgraphs that do not represent a single-linkage hierarchical clustering problem. Without this property, we would be forced to overlap the subproblems, as in [15], resulting in duplicated computation. The second advantage we gain by using Prim's algorithm is its efficiency in computing the MST. Lastly, we can implement Prim's so that both types of subproblems can be solved in-place, that is, without any data structures beyond what is required to store the output. We discuss our implementations of Prim's algorithm for each of the two types of subproblems in the next two paragraphs.

The first type of subproblem to solve is the case in which the subgraph consists of one or two complete subgraphs. As the MST of disconnected components will just be the union of the two respective MSTs, we just focus on how to apply Prim's algorithm to a complete subgraph in which edge weights are determined by the distance between pairs of data points. The main data structure we use is an array of (vertex, vertex, distance) triples. During the algorithm, these triples represent a vertex that has not been added to the current subgraph, the subgraph vertex that is nearest this vertex, and the distance between these vertices. To start, we select the first data point as the start of our subgraph, and populate the array with triples containing the distance of every other data point to this point. As we update these distances, we calculate the minimum distance and add the corresponding edge, with its endpoints and weight, to our MST. Afterwards, we update every other distance value with respect to the newly added vertex, simultaneously calculating the minimum distance. We continue to add vertices until our subgraph spans all of the data, and the array consists of the MST edges for the graph. By swapping the selected edges out of the "active" portion of the array every time we add an edge to the MST, the algorithm can operate using only the memory allocated for storing the MST edges. To further improve efficiency, we abort distance computations that exceed the current minimum distance to the subgraph for a vertex.

The other type of subproblem consists of the bipartite complete subgraph between two data partitions. The main difference between this subgraph and the previous case (aside from having twice as many vertices as a single complete graph), is that we separate the array into two parts, one for each data partition. We start by selecting the first data point in the first partition, computing distances to every vertex in the second partition, selecting the minimum distance for the second partition, and computing the distance between this vertex and every vertex in the first partition. Thereafter, when we select a vertex to add to the subgraph, we only need to update the distances for vertices in the opposite partition as the vertex, though we need to perform a scan of the distances in the same partition in order to find the second (new) minimum distance for that partition. Like the previous case, we swap the edge triples to the end of their respective halves of the array as we add edges to the MST, and we halt distance calculations early when they are too large.

In steps 5–7 of Algorithm 1, we iteratively combine MSTs into larger MSTs until we have the MST for the entire graph. We combine MSTs two at a time by iterating through their edges in sorted order, adding any edge that does not join vertices that are already in the same component. Effectively, we are applying Kruskal's algorithm to combine the MSTs. By sorting the MST edges (step 5) and using a Union-Find (or Disjoint Set) data structure to keep track of the component to which each data point belongs, we can combine MSTs efficiently [8]. A pseudocode description of this procedure appears in Algorithm 2. By combining the MSTs in a binary fashion, we can quickly eliminate edges from the candidate MSTs, increasing the efficiency of this phase of the algorithm (see Figure 2). Moreover, if we start by combining MSTs that have more vertices in common, we are even more likely to eliminate incorrect edges sooner. For this reason, we assign processes with one or two complete subgraphs to be located near the process with the bipartite complete graph between those partitions. Thus, when we combine the dendrograms from consecutive processes, these processes are very likely to have overlapping data partitions, and we can detect and eliminate edges that create cycles sooner, reducing the overall communication costs for the algorithm.

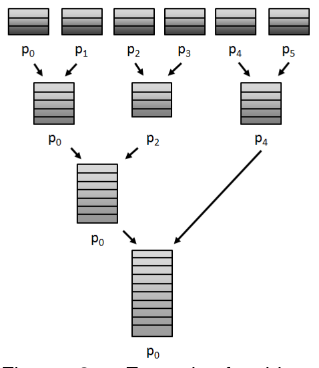

Figure 2: Example for binary merging of six partial dendrograms in three rounds. Each round, half of the processes pass their dendrograms, while the others merge them.

Algorithm 2: Pseudocode for merging two MSTs

```
   Input: $M_1$: Sorted list of candidate MST edges
   Input: $M_2$: Second sorted list of candidate MST edges
   Output: $M_3$: Sorted edge list for MST of $D_1 \cup D_2$
1  Algorithm: Merge($M_1$, $M_2$)
2  Initialize $M_3$ to be empty
3  Initialize Union-Find data structure
4  Let $(u_1, v_1, w_1)$ be the vertices and weight of the min edge in $M_1$
5  Let $(u_2, v_2, w_2)$ be the vertices and weight of the min edge in $M_2$
6  while $M_1$ and $M_2$ have edges remaining do
7      if $w_1 < w_2$ then
8          if union($u_1, v_1$) succeeds then
9              Add $(u_1, v_1, w_1)$ to $M_3$
10         Let $(u_1, v_1, w_1)$ be the next lightest edge in $M_1$
11     else
12         if union($u_2, v_2$) succeeds then
13             Add $(u_2, v_2, w_2)$ to $M_3$
14         Let $(u_2, v_2, w_2)$ be the next lightest edge in $M_2$
15     end
16 end
17 Add the remaining edges $(u, v, w)$ of $M_1$ or $M_2$ to $M_3$ as long as union($u, v$) succeeds
18 return $M_3$
```

3.1 Theoretical results

In this section, we discuss the correctness of our parallel algorithm. As much of the parallel algorithm and problem decomposition described in Section 3 is based on the idea of computing a minimum spanning tree (MST), the first result we present here formally establishes the connection between these two problems. In particular, we show that the single-linkage hierarchical clustering dendrogram for a dataset and the MST of the corresponding complete graph produce identical clusters when we apply the same threshold to each. This result is also useful in that it simplifies the terminology and intuition behind the proof of correctness later in this section.

First, though, we formalize some of the language we will use for this result. A dendrogram is a relationship on a dataset that induces a partitional clustering of the dataset for each height $h \in \mathbb{R}_{\geq 0}$. More formally, it is a function between $\mathbb{R}_{\geq 0}$ and the set of partitions on the dataset such that (1) the partition at height 0 groups the data into sets containing data points located at distance 0 from one another, (2) the group or equivalence class containing a given data point at a given height must be a superset of every group containing that data point at a lower height, and (3) if C_1 and C_2 are distinct sets in the partition at a given height h for which the minimum distance among any pair of points between C_1 and C_2 is $h' > h$, then all vertices of C_1 and C_2 will be in the same cluster at height h' and above. We assume that all data points are comparable; that is, they have some distance defined between them. As a consequence, every dendrogram must, beyond a certain height, partition the data into one set containing every data point. We refer to the sets or equivalence classes defined by the partitions as *clusters*, and we say that a dendrogram *merges* clusters C_1 and C_2 at height h' iff h' is the minimum distance among any pair of points between C_1 and C_2.

Theorem 3.1. *Let T represent the single-linkage dendrogram of the dataset D, and let M be the minimum spanning tree of the complete weighted graph induced by the distances between every pair of points in D. The set of clusters formed by T at height h are exactly the components of M formed by removing all edges with weight greater than h.*

Proof. We prove the claim by induction. Specifically, we show that two clusters C_1 and C_2 are merged at height h in T if and only if the corresponding components C_1' and C_2' in the MST are joined by at least one edge of weight h and no lighter edge. While the domain of the height and distance threshold is continuous, the dendrogram and the MST only produce different clusterings at finitely many values. As such, this condition will ensure that the dendrogram clusters and MST components produce equivalent partitions of the data at every height/distance.

At height $h = 0$, the clusters in T consist of the co-located (zero distance) points in the dataset. In the complete graph induced by distances between the data points, co-located points form cliques with edges of weight 0, from which some spanning tree will be chosen for the MST M. Moreover, any points in M that are connected by edges of weight 0 in M must be co-located due to the transitivity of distance metrics, so the components of M connected by edges of weight 0 will be exactly the set of co-located points.

Now, suppose that every pair of clusters in T and components in M partition the dataset equally at every height (edge weight) less than h for some $h > 0$. If C_1 and C_2 are merged at height h in T, then the minimum distance between C_1 and C_2 must be h. In particular, there must be at least one pair of data points u and v between the clusters that are separated by distance h. By the inductive hypothesis, the corresponding vertex sets C_1' and C_2' must form components in M with edges less than h. Since u and v are joined by an edge of weight h, Kruskal's Algorithm would join u to v at height h unless (u,v) created a cycle. While this case is possible if there are multiple edges with weight h between C_1' and C_2', the components C_1' and C_2' will be joined into a larger component by an edge of weight h, regardless. Moreover, there will be no edges with weight less than h between C_1' and C_2' since h is the minimum distance between clusters C_1 and C_2.

Conversely, suppose that the minimum-weight edge joining components C_1' and C_2' in M has weight h. By the inductive hypothesis, components C_1' and C_2' correspond to clusters C_1 and C_2 at height less than h. Since h is the minimum distance out of all pairs of vertices between C_1' and C_2', h will be the minimum distance between clusters C_1 and C_2. As such, C_1 and C_2 will be merged at height h. □

We now prove the correctness of Algorithm 1. We break this proof into two parts, first showing (in Lemma 3.2) that the output of Algorithm 1 is a spanning tree, then showing that the output must have the same weight as a true MST (in Theorem 3.3). Per Theorem 3.1, this implies that the dendrogram produced by the algorithm will merge the same clusters at the same height as the true dendrogram. Notably, in these proofs, we do not try to show that the output of Algorithm 1 will exactly match the output of a serial MST algorithm—or even match its own output using different process counts—just that the result will be a minimum-weight spanning tree. This distinction is useful as there are many cases in real datasets in which two or more pairs of data points may have equal distances between them. As the algorithm does not distinguish between distances (edge weights) that are equal, we do not guarantee that we will find the same MST every time.

Lemma 3.2. *The output of Algorithm 1 is a spanning tree.*

Proof. During the merging procedure (Algorithm 2), no edge will be included in the result if it creates a cycle. Thus, the output of Algorithm 1 will be acyclic, and we only need show that it spans the entire graph. Let u and v be two arbitrary points in the graph. We show inductively that some process will contain a (u,v)-path at every step during the merging procedure.

As the problem decomposition partitions the edges of the graph, there will be one subgraph that contains the edge (u,v). As every subgraph is connected, the MST calculated by the process assigned this subgraph must contain a (u,v)-path P at the outset of the merging procedure.

Suppose that intermediate results from two processes are being merged, where one of the two contains a (u,v)-path P. When the edges are merged, each edge (x,y) of P will be included in the merged result unless (x,y) would create a cycle; i.e., unless there is some (x,y)-path in the merged result composed of lighter edges. In either case, the result will contain an (x,y)-path for every edge (x,y) in the path P. After merging, these paths can be joined to form a (u,v)-walk, implying the existence of a (u,v)-path in the result. As every pair of vertices in the result must be joined by a path and the result contains no cycles, the output of algorithm must be a spanning tree. □

Theorem 3.3. *Let M be a true MST of the graph induced by the dataset, and let M' be the output of by Algorithm 1. The total weight of M' equals the total weight of M, that is, $w(M') = w(M)$.*

Proof. Clearly, as M' is a spanning tree, $w(M) \leq w(M')$ by virtue of the fact that M is a minimum spanning tree. In order to prove that $w(M) = w(M')$, we now show that $w(M') \leq w(M)$ by contradiction.

Suppose that $w(M') > w(M)$. Let E_M and $E_{M'}$ be the edges of M and M' that are exclusive to M and M', respectively. As spanning trees, M and M' must have the same number of edges, so since $w(M') > w(M)$, both E_M and $E_{M'}$ must be non-empty. Let $e = (u,v)$ be an arbitrary edge of E_M, and consider the graph $M' \cup \{e\}$. Since M' was a spanning tree, $M' \cup \{e\}$ contains a single cycle induced by e and the unique (u,v)-path in M'. Let E_h be the set of edges in this path not contained in M. Since M contains e but not the complete cycle, E_h is not empty. If there were always some $e' \in E_h$ such that $w(e') \leq w(e)$, $M'' = M' - \{e'\} \cup \{e\}$ would be a spanning tree with $w(M'') \geq w(M')$ that has one more edge in common with M, and since M and M' are finite, M' could be converted into M without reducing its weight, contradicting our assumption that $w(M') > w(M)$. Thus, if $w(M') > w(M)$, there must be some such set E_h and edge e with $w(e') > w(e)$ for all $e' \in E_h$. We now prove the contradiction by showing that the (u,v)-path in M' must not contain any edge of E_h.

Let $e = (u,v)$. As every edge is distributed to some subgraph in steps 2–3 of Algorithm 1, there will be some subgraph containing e. The MST for this subgraph must include some (u,v)-path P composed of edges with weight no more than $w(e)$. As the MSTs of the different processes are merged, an edge $e_p = (a,b)$ of P may be removed from the result if there is some (a,b)-path in the current solution that has weight no more than $w(e_p)$, but in either case, the merged result must contain a (u,v)-path composed of edges with weight no more than $w(e)$. As this observation holds true for every step of the merging procedure, the final merged output, M' must contain a (u,v)-path composed of edges with weight no more than $w(e)$. Notably, since $w(e') > w(e)$ for all $e' \in E_h$, this (u,v)-path cannot contain any edge of E_h, contradicting our definition of E_h as a set of edges along the (u,v)-path in M'. □

4 Experimental Results and Performance Analysis

In this section, we present empirical results on an implementation of PINK. We evaluate the scalability of the algorithm, as well as its memory and communication requirements. We do not report any measures of clustering quality, as single-linkage hierarchical clustering is a well-studied algorithm, with known strengths and weaknesses. In addition, we verified that all of our results were identical (up to the order of edges with the same distance) to the serial Prim's algorithm.

Our implementation of PINK is in C, with MPI for interprocess communication. The code was compiled and run on the Edison machine at NERSC. Edison is a Cray XC30 supercomputer at National Energy Research Scientific Computing Center (NERSC). It has 664 compute nodes with 64 GB memory per node. Each node consists of two 8-core Intel Sandy Bridge processors at 2.6 GHz. The high-speed interconnect has latency of 0.25 μs to 3.7 μs and MPI bandwidth of 8 GB/sec. PINK was compiled with Cray compiler version 1.01 using the `-O2` optimization flag.

All datasets are encoded as binary files, which are read in and distributed among the MPI processes using MPI-IO. In order for each process to read the correct points from the dataset, each process calculates the length and offset for its two data partitions, defines a file view using `MPI_Type_create_hindexed` and `MPI_File_set_view`, and then reads in the two partitions using `MPI_File_read_all`. The datasets are stored on the Lustre filesystem on Edison. The file striping configuration used in our experiments are 64 stripe count, 1MB stripe size, and the default stripe offset. The I/O peak bandwidth is 36 GB/sec.

In reporting our timing results, we split PINK into four main phases: initialization and reading in the data file, running Prim's algorithm to calculate the MST of the assigned subgraphs for the data and sorting the resulting edges, merging the partial MSTs to calculate the final answer, and outputting the result. We use the built-in `qsort` function to sort the MST edges produced by Prim's algorithm on each process. We evaluate our algorithm using squared Euclidean distance for our distance metric. As noted earlier, running PINK using one process results in applying Prim's algorithm to the entire dataset.

4.1 Datasets

We evaluated PINK on 12 synthetic datasets, as well as 4 sampled cosmology datasets. We generated two types of synthetic data, clustered and uniform random data, across both 10 and 20 dimensions in three different data sizes: 100k, 500k, and 1M data points. For the synthetic clustered data, we used 8 random cluster centers with equal-size Gaussian distributions around each.

To perform the experiments on real-world data, we used a testbed called *millennium-run-simulation*, which consists of four datasets from the database on Millennium Run, the largest simulation of the formation of structure with the ΛCDM cosmogony with a factor of 10^{10} particles [17, 23]. The four datasets, MPAGalaxiesBertone2007 (`MGal-B`) [4], MPAGalaxiesDeLucia2006a (`MGal-D`) [10], DGalaxiesBower2006a (`DGal-B`) [5], and MPAHaloTreesMhalo (`MHalo`) [4] are taken from the Galaxy and Halo databases. To be consistent with the size of the other categories, we have randomly selected 0.5% of the points from these datasets. We also selected the first 3 dimensions from each dataset, as these represent the particle coordinates and would produce a meaningful result when clustered using Euclidean distance. Table 2 shows the structural properties of the datasets.

4.2 Synthetic data

In this section, we present our scalability results on the twelve synthetic datasets described in Section 4.1.

Table 2: Structural properties of the millennium-run-simulation testbed. All datasets included 3 spatial dimensions.

Name	Full size	Sampled size
DGalaxiesBower2006a (`DGal-B`)	101,459,853	5,033,184
MPAGalaxiesBertone2007 (`MGal-B`)	105,592,018	5,242,900
MPAGalaxiesDeLucia2006a (`MGal-D`)	105,592,018	5,242,900
MPAHaloTreesMhalo (`MHalo`)	76,066,700	3,774,888

When reporting our results, we report timing results for the Prim's algorithm and sorting phase as "Prim's," the the merging phase as "Merge," the file reading and output phases as "I/O," and the sum of all four phases as "Total." We consider the time spent in each phase of the algorithm to be the maximum amount of time spent by any MPI process in this phase, which may belong to different processes for the "Total" cases. Additionally, we calculated speedup on p as:

$$S = \frac{mt_m}{t_p},$$

where t_p is the time taken to run PINK on p processes and m is the fewest number of processes used for a given dataset. Timing results for all of our datasets appear in Table 1.

Notably, the main computational part of the algorithm ("Prim's" phase) achieved near-linear or superlinear speedup across all datasets, which we believe is due to a progressively larger portion of the data fitting in the memory cache. Moreover, the time required to merge the subgraphs took a relatively small faction of the total computation time, even at high process counts. All together, the scalability was quite good, especially for the larger, more computationally-intensive runs. In general, datasets with more data points and dimensions, as well as clustered datasets, tended to outperform datasets that were smaller or had a uniform random distribution.

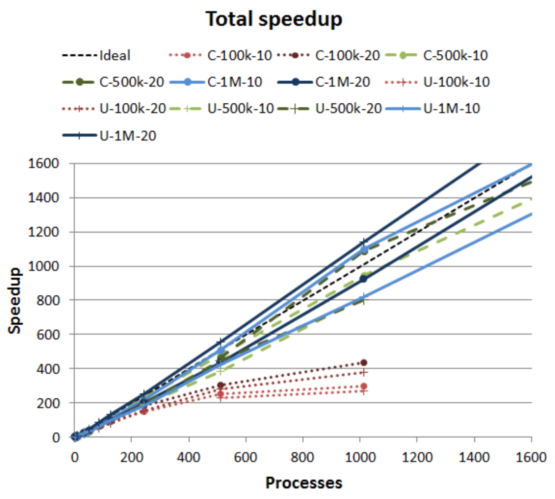

Figure 3: Speedup results for total time for PINK on all 12 synthetic datasets up to 1k processes

We used up to 1013 processes on the smallest datasets (100k points with 10 and 20 dimensions). From the results, we can see that the time required for the algorithm is largely dominated by the time required for Prim's algorithm, which ranges from 177 to 0.2 seconds, whereas the merging phase, which includes all of the communication performed by the algorithm, took at most 0.03 seconds. The time required for reading in, distributing the dataset, and producing the output was consistently between 0.1 and 0.2 seconds, forming a larger portion of the total time at high process counts.

Table 1: Timing summary (in seconds) for synthetic uniform random (U) and random clustered (C) datasets with 100k, 500k, and 1M points across 10 and 20 dimensions, as well as the four cosmology datasets. $p_{\min}$ and $p_{\max}$ represent the minimum and maximum number of processes on which we evaluated each dataset, respectively, and speedups were calculated by approximating the serial time as the product of $p_{\min}$ and the $p_{\min}$ Total time.

Dataset	$p_{\min}$	$p_{\min}$ Prim's	$p_{\min}$ Merge	$p_{\min}$ I/O	$p_{\min}$ Total	$p_{\max}$	$p_{\max}$ Prim's	$p_{\max}$ Merge	$p_{\max}$ I/O	$p_{\max}$ Total	Prim's Speedup	Total Speedup
`U-100k-10`	1	96.92	0.00	0.11	97.03	1013	0.12	0.030	0.205	0.36	801.66	272.59
`U-100k-20`	1	152.25	0.00	0.15	152.40	1013	0.18	0.03	0.19	0.40	842.41	380.13
`U-500k-10`	1	3326.54	0.00	0.50	3327.04	5101	0.59	0.20	0.75	1.54	5643.29	2154.14
`U-500k-20`	1	5873.83	0.00	0.68	5874.52	1013	5.93	0.17	1.23	7.32	990.78	802.23
`U-1M-10`	1	14651.04	0.00	1.16	14652.20	5101	2.82	0.45	1.17	8.78	5194.13	3210.17
`U-1M-20`	8	4289.99	0.19	1.24	4291.42	5101	2.82	0.45	1.38	7.12	6468.00	4821.43
`C-100k-10`	1	108.35	0.00	0.14	108.35	1013	0.15	0.03	0.19	0.18	743.07	299.74
`C-100k-20`	1	177.93	0.00	0.12	178.05	1013	0.20	0.03	0.18	0.41	889.67	438.38
`C-500k-10`	1	4320.02	0.00	0.46	4320.48	6050	0.56	0.19	0.77	1.52	7671.89	2836.07
`C-500k-20`	1	7206.01	0.00	0.59	7206.60	2048	3.15	0.18	0.67	4.01	2284.97	1798.64
`C-1M-10`	1	19882.84	0.00	1.02	19883.86	6050	2.47	0.41	1.27	4.15	8040.12	4787.01
`C-1M-20`	1	30062.03	0.00	1.13	30063.16	4050	7.20	0.41	1.28	8.89	4177.02	3382.76
`DGal-B`	32	8130.05	1.48	2.54	8134.06	6050	34.32	2.29	2.85	39.46	7580.90	6596.17
`MGal-B`	32	8573.54	1.53	2.76	8577.83	6050	36.29	2.40	2.98	41.67	7560.58	6587.16
`MGal-D`	32	8438.03	1.53	2.64	8442.20	1013	225.68	2.06	2.74	230.48	1196.48	1172.13
`MHalo`	32	5152.93	1.30	2.02	5156.25	2048	56.50	1.93	2.14	60.57	2918.62	2724.32

Overall, the scalability of the algorithm is quite good up to 242 processes, achieving speedups of 155–180. Beyond this point, Prim's algorithm finishes in less than half a second, and the scalability suffers.

On the medium-sized datasets (500k points with 10 and 20 dimensions), the time required for the merging phase is again a small portion of the overall time, though this fraction increases with the number of processes, up to roughly 13% of the total time when running dataset `U-500k-10` on 5101 processes. In addition, the I/O time required to read and output the data was relatively low, taking 5–6% of the total for all runs on 512 processes, though did increase to nearly 50% of the total time on the highest-process runs. The scalability for these datasets are also quite good, with a nearly linear end-to-end speedup on the clustered data and 80% of linear speedup for the uniform random datasets.

We tested PINK on the largest synthetic datasets using up to 6050 processes. From Table 1, we see that the amount of time taken by the merging phase of the algorithm increased very slowly, taking no more than 10% of the total time, while the I/O costs were again more prominent, taking up to roughly 31% of the total algorithm time at 6050 processes on dataset `C-1M-10`.

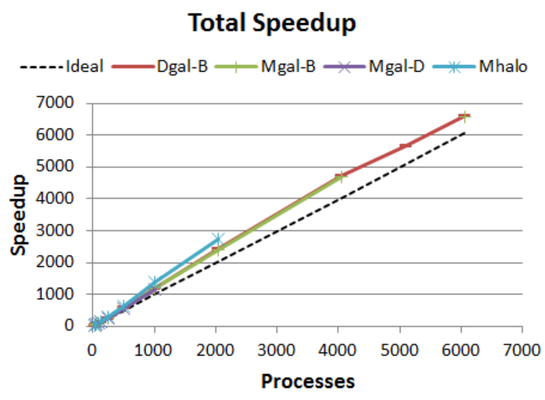

Figure 4: Empirical results for running PINK on the four sampled cosmology datasets.

The largest datasets we tested were the sampled cosmology datasets, with 3–5 million data points. With the increased amount of computation, we saw even stronger scaling results, with super-linear speedups on all four datasets up to 6050 processes, even after I/O and communication costs were included.

4.3 Memory usage

In addition to its demonstrated scalability, PINK has an advantage that it requires only linear amount of space. To reinforce the advantage of not storing an explicit distance matrix, we draw a comparison between the memory used by PINK and the amount of memory that would be required to store a distance matrix. We estimate the memory usage of our PINK implementation for each process by adding the memory allocated in all of the calls to `malloc` in terms of the data size and number of processes. We evaluate the resulting expression using the datasets described in Section 4.2, rounding up when the number of partitions divides the data points unevenly. Comparatively, a distance matrix requires $\binom{n}{2}$ double-precision values. The results of this comparison appear in Table 3.

Table 3: Estimated memory usage per process for serial and parallel PINK algorithm. All values are in MB, unless otherwise noted. The aggregate memory cost for storing a full distance matrix (DM) is included for comparison.

Data	1 proc	50	512	8192	DM
`U-100k-10`	12.6	6.18	5.08	4.70	37.3 GB
`U-500k-10`	62.9	30.9	25.4	23.5	931 GB
`U-1M-10`	126	61.8	50.8	47.0	3.64 TB
`U-1M-20`	202	77.1	55.6	48.2	3.64 TB

From the table, we can see that the memory that would be required to store the distance matrix for a dataset of 100k points is more than 35 GB, more than 900 GB for a dataset of 500k points, and more than 3.5 TB for a dataset of 1 million points. For a dataset of 1 million data points, even a fully-distributed distance matrix would use 466 MB per process, nearly 10 times the entire amount used by PINK. While PINK would use a grand total of 37.6 GB across 8192 processes for the 100k dataset, the algorithm finished in less than half a second on 512 processes, and it is not reasonable to run the algorithm using 8k processes for a dataset of 100k points.

4.4 Communication scalability

Lastly, we wished to evaluate the communication behavior of PINK. Recall that when two partial MSTs are being combined by the algorithm, some of the edges in one or both MSTs may be eliminated, making the overall communication behavior difficult to evaluate analytically. So, to test the communication behavior, we counted the number of edges received by process p_0, the process that collects the full MST at the end of the algorithm, during each step of the merging phase for the `C-1M-30` dataset.

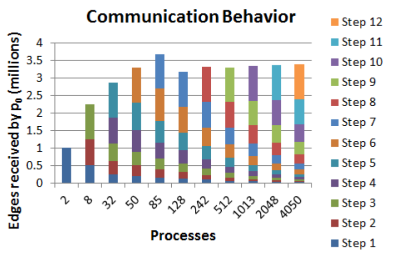

Figure 5: Communication behavior on random clustered dataset with 1 million data points and 20 dimensions, in terms of the number of edges received by p_0 during each step of the merging phase

The results, which appear in Figure 5, reveal that, in spite of the increasing number of MSTs received, the total data size (number of edges) increases very slowly beyond 128 processes (16 data partitions). This slowness in communication growth suggests that edges are being removed from the MSTs efficiently as they are being merged. Furthermore, we note that the communication volume increases following powers of two (when the binary merging tree gains an additional level), but decreases as the number of processes approaches the next power of two, due to the decreasing size of the individual MSTs at the outset of the merging process. These patterns were observed across all datasets tested.

5 Conclusion

In this paper, we have described PINK, a memory-efficient scalable parallel algorithm for single-linkage hierarchical clustering, including a formal proof of its correctness. Moreover, we evaluated PINK empirically, finding that it achieves near-linear or superlinear scaling when more than 6,000 processes on both real and synthetic data, due in part to its low communication costs.

Acknowledgements

This work is supported in part by the following grants: NSF awards CCF-0833131, CNS-0830927, IIS-0905205, CCF-0938000, CCF-1029166, and OCI-1144061; DOE awards DE-FG02-08ER25848, DE-SC0001283, DE-SC0005309, DESC0005340, and DESC0007456; and AFOSR award FA9550-12-1-0458. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *Journal of Parallel and Distributed Computing*, 65(9):994–1006, 2005.

[2] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 66(11):1366–1378, 2006.

[3] J. L. Bentley. A parallel algorithm for constructing minimum spanning trees. *Journal of Algorithms*, 1(1):51–59, 1980.

[4] S. Bertone, G. De Lucia, and P. Thomas. The recycling of gas and metals in galaxy formation: predictions of a dynamical feedback model. *Monthly Notices of the Royal Astronomical Society*, 379(3):1143–1154, 2007.

[5] R. Bower, A. Benson, R. Malbon, J. Helly, C. Frenk, C. Baugh, S. Cole, and C. Lacey. Breaking the hierarchy of galaxy formation. *Monthly Notices of the Royal Astronomical Society*, 370(2):645–655, 2006.

[6] S. Chung and A. Condon. Parallel implementation of boruvka's minimum spanning tree algorithm. In *Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International*, pages 302–308, apr 1996.

[7] R. Cole, P. N. Klein, and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, SPAA '96, pages 243–250, New York, NY, USA, 1996. ACM.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[9] M. Dash, S. Petrutiu, and P. Scheuermann. Efficient parallel hierarchical clustering. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 363–371. Springer Berlin / Heidelberg, 2004.

[10] G. De Lucia and J. Blaizot. The hierarchical formation of the brightest cluster galaxies. *Monthly Notices of the Royal Astronomical Society*, 375(1):2–14, 2007.

[11] F. Dehne and S. Götz. Practical parallel algorithms for minimum spanning trees. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 366–371, oct 1998.

[12] Z. Du and F. Lin. A novel parallelization approach for hierarchical clustering. *Parallel Computing*, 31(5):523–527, 2005.

[13] A. El-Hamdouchi and P. Willett. Comparison of hierarchic agglomerative clustering methods for document retrieval. *The Computer Journal*, 32(3):220–227, 1989.

[14] J. C. Gower and G. J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 18(1):pp. 54–64, 1969.

[15] W. Hendrix, M. M. A. Patwary, A. Agrawal, W. keng Liao, and A. Choudhary. Parallel hierarchical clustering on shared memory platforms. In *19th IEEE International Conference on High Performance Computing*, HiPC'12, 2012.

[16] E. Johnson and H. Kargupta. Collective, hierarchical clustering from distributed, heterogeneous data. In M. Zaki and C.-T. Ho, editors, *Large-Scale Parallel Data Mining*, volume 1759 of *Lecture Notes in Computer Science*, pages 803–803. Springer Berlin / Heidelberg, 2000.

[17] G. Lemson and the Virgo Consortium. Halo and galaxy formation histories from the millennium simulation: Public release of a VO-oriented and SQL-queryable database for studying the evolution of galaxies in the LambdaCDM cosmogony. *Arxiv preprint astro-ph/0608019*, 2006.

[18] V. Olman, F. Mao, H. Wu, and Y. Xu. Parallel clustering algorithm for large data sets with applications in bioinformatics. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 6:344–352, April 2009.

[19] C. F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21(8):1313–1325, 1995.

[20] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM Journal on Computing*, 31(6):1879, 2002.

[21] C. Poon and V. Ramachandran. A randomized linear work erew pram algorithm to find a minimum spanning forest. In H. Leong, H. Imai, and S. Jain, editors, *Algorithms and Computation*, volume 1350 of *Lecture Notes in Computer Science*, pages 212–222. Springer Berlin / Heidelberg, 1997.

[22] R. Sibson. Slink: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[23] V. Springel, S. White, A. Jenkins, C. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, et al. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature*, 435(7042):629–636, 2005.

[24] P. Tamayo, D. Slonim, J. Mesirov, Q. Zhu, S. Kitareewan, E. Dmitrovsky, E. S. Lander, and T. R. Golub. Interpreting patterns of gene expression with self-organizing maps: Methods and application to hematopoietic differentiation. *Proceedings of the National Academy of Sciences*, 96(6):2907–2912, 1999.

[25] J. Xu and A. Hagler. Chemoinformatics and drug discovery. *Molecules*, 7(8):566–600, 2002.