ParChain: a framework for parallel hierarchical agglomerative clustering using nearest-neighbor chain

ParChain: A Framework for Parallel Hierarchical Agglomerative Clustering using Nearest-Neighbor Chain

Shangdi Yu
MIT CSAIL
shangdiy@mit.edu

Yiqiu Wang
MIT CSAIL
yiqiuw@mit.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Laxman Dhulipala
MIT CSAIL
laxman@mit.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

ABSTRACT

This paper studies the hierarchical clustering problem, where the goal is to produce a dendrogram that represents clusters at varying scales of a data set. We propose the ParChain framework for designing parallel hierarchical agglomerative clustering (HAC) algorithms, and using the framework we obtain novel parallel algorithms for the complete linkage, average linkage, and Ward's linkage criteria. Compared to most previous parallel HAC algorithms, which require quadratic memory, our new algorithms require only linear memory, and are scalable to large data sets. ParChain is based on our parallelization of the nearest-neighbor chain algorithm, and enables multiple clusters to be merged on every round. We introduce two key optimizations that are critical for efficiency: a range query optimization that reduces the number of distance computations required when finding nearest neighbors of clusters, and a caching optimization that stores a subset of previously computed distances, which are likely to be reused.

Experimentally, we show that our highly-optimized implementations using 48 cores with two-way hyper-threading achieve 5.8–110.1x speedup over state-of-the-art parallel HAC algorithms and achieve 13.75–54.23x self-relative speedup. Compared to state-of-the-art algorithms, our algorithms require up to 237.3x less space. Our algorithms are able to scale to data set sizes with tens of millions of points, which existing algorithms are not able to handle.

PVLDB Reference Format:
Shangdi Yu, Yiqiu Wang, Yan Gu, Laxman Dhulipala, and Julian Shun. ParChain: A Framework for Parallel Hierarchical Agglomerative Clustering using Nearest-Neighbor Chain. PVLDB, 15(2): 285 - 298, 2022.
doi:10.14778/3489496.3489509

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yushangdi/parChain.

1 INTRODUCTION

Clustering is an unsupervised machine learning method that has been widely used in many fields including computational biology, computer vision, and finance to discover structures in a data set [2, 6, 24, 32, 39, 46, 64, 68]. To group similar objects at all resolutions, a *hierarchical clustering* can be used to produce a tree that represents clustering results at different scales. The resulting hierarchical cluster structure is called a *dendrogram*, which is a tree representing the agglomeration of clusters, as shown in Figure 1(b).

Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489509

There is a rich literature on designing hierarchical agglomerative clustering (HAC) algorithms [46]. Unfortunately, exact HAC algorithms usually require $\Omega(n^2)$ work, since the distances between all pairs of points have to be computed. To accelerate exact HAC algorithms due to their significant computational cost, there have been several parallel exact HAC algorithms proposed in the literature [23, 31, 33, 41, 42, 58, 66, 70], but most of them maintain a distance matrix, which requires quadratic memory, making them unscalable to large data sets. The only parallel exact algorithm that works for the metrics that we consider and uses subquadratic space is by Zhang et al. [70], but it has not been shown to scale to large data sets. In this paper, we propose the ParChain framework for designing parallel exact HAC algorithms that use *linear memory*, based on the classic nearest-neighbor chain algorithm.

The nearest-neighbor chain (NNC) algorithm [45] is a popular algorithm that can be used for a wide range of HAC metrics [5, 19, 28, 34, 53]. A ***nearest-neighbor chain (NNC)*** is a linked list of nodes, where each node represents a cluster and all except at most one node have a pointer to its nearest neighbor (its successor). The chain can start from an arbitrary cluster. If a node does not have a pointer, its nearest neighbor is not yet computed, and this node is called a ***terminal node***. If we follow the pointers on the nodes, we obtain a "chain" of clusters, which either terminates at a terminal node, or at a ***reciprocal nearest neighbor (R-NN) pair***, which is a pair of clusters that are each other's nearest neighbor. The sequential NNC algorithm [5, 19] works by iteratively adding a node to a single chain through finding the nearest neighbor of the terminal node until an R-NN pair is found. Each point is initially a singleton cluster and a terminal node of a single-node chain. The sequential algorithm picks an arbitrary node to start growing from. After an R-NN pair is found, the R-NN pair is then merged, and the chain is grown again to find another R-NN pair to merge. After $n-1$ merges, the algorithm finishes, producing a hierarchy of clusters.

Example. We now give an example of the definitions above by briefly describing running our ParChain framework for parallel HAC on the small data set in Figure 1. This example uses the complete linkage metric, where the distance between two clusters is the distance of the farthest pair of points, one from each cluster.

Our framework is based on the key insight that all R-NN pairs can be merged simultaneously, which provides parallelism. On each round, it merges all R-NN pairs in parallel (breaking ties lexicographically[1] to prevent cycles). Before the first round, each point in $\{a, \ldots, f\}$ is represented by a chain with only one node, and all points are singleton terminal nodes. The R-NN pairs are found by finding the nearest neighbors of all terminal nodes, which by definition are the clusters whose nearest neighbors are unknown at the

[1] We use the ID of the lexicographically first point in each cluster as the cluster's ID.

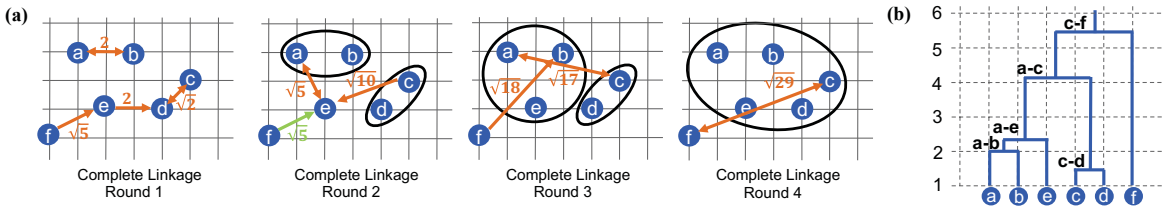

Figure 1: (a) The four rounds of the nearest-neighbor chain algorithm on a six point data set using the complete linkage metric. The black circles are clusters containing more than one point. An arrow from point x to point y means that x's cluster's nearest neighbor is y's cluster. The orange arrows are new neighbors found on this round; the green arrow means the nearest neighbor is not updated on this round due to the reducibility property. On Round 2, we avoided a nearest neighbor search for cluster $\{f\}$. The numbers on the arrows between pairs of points are the distances between the clusters that the points belong to according to the complete linkage metric (furthest distance between a pair of points, one from each cluster). Ties are broken lexicographically. **(b)** The dendrogram for complete linkage clustering. The label on each internal node corresponds to the furthest point pair in the two clusters that are merged in the algorithm, and its distance is equal to the node's height in the dendrogram.

beginning of a round. On the first round, we find the nearest neighbors for all points in parallel. Now we have two chains, $\{f, e, d, c\}$ and $\{a, b\}$. $\{e\}$, for example, is $\{f\}$'s successor. (a, b) and (c, d) are two R-NN pairs, and so we merge them in parallel and create dendrogram nodes for clusters $\{a, b\}$ and $\{c, d\}$. At the beginning of the second round, $\{a, b\}$, $\{c, d\}$, and e are terminal nodes. After we find their nearest neighbors and grow the chain, $(\{a, b\}, e)$ is the only R-NN pair (we broke the tie for e's two nearest neighbors, $\{a, b\}$ and f, by choosing $\{a, b\}$), and so we merge it and create a dendrogram node for cluster $\{a, b, e\}$. We do not need to find the nearest neighbor of f in this round, because it is not a terminal node and we know that its nearest neighbor e will not change (due to the reducibility property which will be defined more formally in Section 2). On the third round, $\{a, b, e\}$, $\{c, d\}$, and f are terminal nodes. We find the nearest neighbors for them and merge the R-NN pair $(\{a, b, e\}, \{c, d\})$. Finally, on the fourth round, the R-NN pair $(\{a, b, c, d, e\}, f)$ is merged.

ParChain achieves high space efficiency and parallelism, which enables it to scale HAC to large data sets that are orders of magnitude larger than those used in previous work. There are two challenges in achieving both space efficiency and high parallelism. The first challenge is to maintain all chains and merge reciprocal nearest neighbor clusters correctly and efficiently *in parallel*. Unlike Jeon and Yoon's algorithm [33], which is based on locks (and has limited parallelism for large core counts), we use lock-free approaches based on filtering and atomic operations (Section 3). The second challenge is to efficiently find the nearest neighbors of clusters when growing the chain, *without* storing the distance matrix. We introduce a range query optimization that significantly reduces the number of distance computations used to find the nearest neighbor of a cluster for low-dimensional data sets in Euclidean space (Section 4), as well as a new caching technique that stores a subset of previous distance computations that are likely to be reused to further accelerate nearest neighbor searches (Section 5). In the example in Figure 1, the range query optimization avoids computing the distance between clusters $\{e\}$ and $\{c, d\}$ in round 2 when $\{e\}$ searches for its nearest neighbor, because only clusters $\{a, b\}$ and $\{f\}$ will be within the range. The caching technique avoids storing all pairs of distances among the six points. In contrast, many previous methods [23, 31, 33, 51, 52, 58] require a quadratic-space distance matrix and compute distances to all other clusters when searching for the nearest neighbor of a cluster.

We apply ParChain to develop new linear-space parallel HAC algorithms for the complete, Ward's, and average linkage criteria. Our framework can be applied for any linkage criteria that satisfies the reducibility property, which ensures that the nearest neighbor distance of clusters can never be smaller as clusters merge (defined more formally in Section 2).

Though the worst case time complexity of our algorithms is $O(n^3)$, we observe that the running time is close to quadratic in practice on low-dimensional data sets because the range query is able to filter out many clusters. Many spatial, sensor, and computer vision data sets, where HAC is applicable, are low dimensional. In Section 6, we show experimentally on a variety of real-world and synthetic data sets (up to 16 dimensions) that our algorithms achieve 13.75–54.23x self-relative speedup on a 48-core machine with two-way hyper-threading. We also achieve 5.8–110.1x speedup over the state-of-the-art parallel implementations. Our algorithms use up to 237.3x less space than existing implementations, and are able to scale to larger data sets with tens of millions of points, which existing algorithms are not able to handle.

We summarize our contributions below:

- The ParChain framework for parallel HAC using linear space.
- A range query optimization for fast nearest neighbor search for the complete, Ward's, and average linkage criteria.
- A cache table optimization for reducing the number of cluster distance computations.
- Experiments showing that the algorithms in ParChain achieve significant speedups over state-of-the-art.

2 BACKGROUND

The input to the ***hierarchical agglomerative clustering (HAC)*** problem is a data set to be clustered and a linkage criteria that specifies how distances between clusters are computed. The output of HAC is a tree called a ***dendrogram***, where the height of each dendrogram node represents the dissimilarity between the merged two clusters according to the desired linkage criteria. A flat clustering, which assigns the same ID to every object in the same cluster and different IDs to objects in different clusters, can be obtained by cutting the dendrogram at some height. Thus, cutting the dendrogram at different heights gives clusterings at different scales. An example of a dendrogram is shown in Figure 1(b). In the rest of the section, we present our notations, the three linkage criteria considered in this paper, and some relevant techniques used by our algorithm.

Notation. Let v be a length-d vector in d-dimensional space, and let $\|v\|$ denote the L_2 norm of v, i.e., $\|v\| = \sqrt{\sum_{i=1}^{d} |v[i]|^2}$ where $v[i]$ is the i'th coordinate of v. $\bar{x}_A$ denotes the ***centroid*** of cluster A, i.e., $\bar{x}_A = \frac{1}{|A|}\sum_{x\in A} x$, where the x's are points in cluster A. $\mathrm{Var}(A)$ denotes the ***variance*** of cluster A, where $\mathrm{Var}(A) = \sum_{x\in A} \|x-\bar{x}_A\|^2$. $\Delta(A, B)$ denotes the ***distance*** between clusters A and B, and its formula depends on the linkage criteria.

2.1 Linkage Criteria

We now formally define the linkage criteria considered in this paper. We use the Euclidean distance metric for all linkage criteria. For average linkage, we also consider the squared Euclidean distance metric. The definitions of cluster distance under each linkage criteria and distance metric are included in Table 1. We also include the work of each distance computation, the radius, and criteria-specific optimizations used in our range query optimization for different linkage criteria. The radius and optimizations will be discussed in more detail in Section 4. We use *comp, Ward, avg-1, and avg-2* to refer to complete linkage, Ward's linkage, average linkage with Euclidean distance metric, and average linkage with squared Euclidean distance metric, respectively.

Complete Linkage. In complete linkage [51, 58], the distance between two clusters is the maximum distance between a pair of points, one from each cluster.

Ward's Linkage. In Ward's linkage [55, 67], or minimum variance linkage, the distance between two clusters is the increase in total variance if the two clusters merge.

Unweighted Average Linkage. In unweighted average linkage [39, 62], the distance between two clusters is the average distance between pairs of points, one from each cluster.

For Ward's linkage and average linkage with the squared Euclidean distance metric, we can be more space-efficient and compute the distance between two clusters in constant time by storing the mean and variance of every cluster, which takes only linear space overall. The newly merged cluster's mean and variance can be computed in constant time, where the new cluster's mean is an average of the means of the two original clusters, weighted by their sizes, i.e., $\bar{x}_{A\cup B} = \frac{|A|}{|A|+|B|}\bar{x}_A + \frac{|B|}{|A|+|B|}\bar{x}_B$. The variance is:

$$\mathrm{Var}(A\cup B) = \frac{|A|\|\bar{x}_A - \bar{x}_{A\cup B}\|^2 + |B|\|\bar{x}_B - \bar{x}_{A\cup B}\|^2}{|A|+|B|} + \frac{|A|\mathrm{Var}(A) + |B|\mathrm{Var}(B)}{|A|+|B|}$$

Lance-Williams Formula. Many clustering metrics can be described using the Lance-Williams formula [38]. Given the distance between three clusters A, B, and C, we can obtain the distance between $A\cup B$ and C using the following formula, with the coefficients for the metrics described above given in Table 2:

$$\Delta(A\cup B, C) = a_1\Delta(A, C) + a_2\Delta(B, C) + b\Delta(A, B) + c\,|\Delta(A, C) - \Delta(B, C)|$$

The Lance-Williams formula allows for constant time distance computation if we have the distances among clusters A, B, and C. However, maintaining all these distances requires a distance matrix that takes quadratic space.

Reducibility. We say a metric has the ***reducibility property*** [8, 33, 53, 54] if we have $\Delta(A, B) < \Delta(A\cup B, C)$ when $\Delta(A, B) < \Delta(A, C)$ or $\Delta(A, B) < \Delta(B, C)$. All of the metrics introduced above satisfy the reducibility property. The reducibility property ensures that the nearest neighbor of a cluster does not change unless one of the clusters merged is its nearest neighbor. For metrics that satisfy the reducibility property [53], we can perform clustering using the *nearest-neighbor chain algorithm* [5, 19, 28, 33, 34, 53] introduced in Section 1. The reducibility property provides the parallelism in the nearest-neighbor chain algorithms since we can merge multiple reciprocal pairs simultaneously.

2.2 Relevant Techniques

kd-tree. A ***kd-tree*** is a binary spatial tree where each internal node contains a splitting hyperplane that partitions the points contained in the node between its two children. The root node contains all of the points, and the kd-tree is constructed by recursing on each of its two children after splitting, until a leaf node is reached. A leaf node contains at most c points for a predetermined constant c. The kd-tree can be constructed in parallel by performing the split and constructing each child in parallel. The *bounding box* of a node is the smallest rectangular box that encloses all of its points.

Nearest-Neighbor Query. A ***nearest-neighbor query*** takes a set of points $\mathcal{P}$ and a query point q, and returns for q its nearest neighbor in $\mathcal{P}$ (besides itself if $q \in \mathcal{P}$). An ***all-nearest-neighbor query*** takes a set of points $\mathcal{P}$, and returns for all points in $\mathcal{P}$ its nearest neighbor in $\mathcal{P}$ besides itself. The all-nearest-neighbor query can be performed efficiently using a dual-tree traversal [14, 15, 47], which we have parallelized.

Range Query. A ***range query*** takes a set of points $\mathcal{P}$, constructs a data structure to store the points, and reports or counts all points in some range B. In this paper, we use balls to represent the ranges, and we use kd-trees to store the points.

Other Parallel Primitives. 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 ***reduce*** takes as input a sequence $[a_1, a_2, \ldots, a_n]$ and an associative binary operator $\oplus$, and returns the overall sum (using $\oplus$) of the elements $(a_1 \oplus a_2 \oplus \ldots \oplus a_n)$. A parallel ***hash table*** stores key-value pairs, and supports concurrent insertions, updates, and finds. 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 [61]. WriteMax is similar but writes the largest value.

3 THE PARCHAIN FRAMEWORK

In this section, we present our framework ParChain for parallelizing the nearest-neighbor chain (NNC) algorithm, which works for all linkage criteria that satisfy the reducibility property explained in Section 2.1. The NNC algorithm exposes more parallelism than the naive generic algorithm, where only the R-NN pair with minimum distance is merged, by allowing multiple R-NN pairs to be merged simultaneously. Hence, our framework grows multiple chains and merges all R-NN pairs simultaneously in parallel.

Jeon and Yoon's algorithm [33] uses a similar approach for to grow multiple chains in parallel, but it does merges R-NN pairs asynchronously. It designates some threads for updating chains, and other threads for updating the cluster distances. Their algorithm

Table 1: Definitions, work, radius value, and optimizations used in our range query for different linkage criteria.

Linkage	Cluster Distance $\Delta(A, B)$	Work	Radius	Optimizations
comp	$\max_{x\in A, x'\in B} \|x - x'\|$	$O(n^2)$	β	build kd-tree on all points
Ward	$\sqrt{2(\mathrm{Var}(A\cup B) - \mathrm{Var}(A) - \mathrm{Var}(B))} = \sqrt{\frac{2\|A\|\|B\|}{\|A\|+\|B\|}}\|\bar{x}_A - \bar{x}_B\|^2$	$O(1)$	$\beta\sqrt{\frac{\|C_i\|+n_{\min}}{2n_{\min}\|C_i\|}}$	maintain cluster centroids and sizes
avg-1	$\frac{1}{\|A\|\|B\|}\sum_{x\in A}\sum_{x'\in B}\|x - x'\|$	$O(n^2)$	β	–
avg-2	$\frac{1}{\|A\|\|B\|}\sum_{x\in A}\sum_{x'\in B}\|x - x'\|^2 = \|\bar{x}_A - \bar{x}_B\|^2 + \frac{\mathrm{Var}(A)}{\|A\|} + \frac{\mathrm{Var}(B)}{\|B\|}$	$O(1)$	$\sqrt{\beta}$	maintain cluster centroids, variances, and sizes

Table 2: Coefficients for the Lance-Williams Formula [38].

	a_1	a_2	b	c
Complete linkage	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{2}$
Ward's linkage	$\frac{\|A\|+\|C\|}{\|A\|+\|B\|+\|C\|}$	$\frac{\|B\|+\|C\|}{\|A\|+\|B\|+\|C\|}$	$\frac{-\|C\|}{\|A\|+\|B\|+\|C\|}$	0
Average linkage	$\frac{\|A\|}{\|A\|+\|B\|}$	$\frac{\|B\|}{\|A\|+\|B\|}$	0	0

Algorithm 1: ParChain Framework

Input: n points $\mathcal{P}$, distance structure $\mathfrak{D}$, and cache size s
Output: Dendrogram tree $\mathcal{T}_H$

1 Initialize n dendrogram leaf nodes $C_0, \ldots, C_{n-1}$ to each represent a singleton cluster (a point).
2 Initialize $\mathcal{L}$, a set of n chain nodes, where each $\mathcal{L}_i$ represents a singleton cluster.
3 Initialize $\mathcal{A} = \{C_0, \ldots, C_{n-1}\}$, the set of active clusters.
4 Create cache tables $\{\mathcal{H}_i\}$ for clusters, each of size s.
5 Terminal nodes $\mathcal{Z} = \{\mathcal{L}_0, \ldots, \mathcal{L}_{n-1}\}$.
6 $\mathcal{T}$ = kd-tree $\mathcal{T}_{\mathcal{P}}$
7 **while** $|\mathcal{A}| > 1$ **do**
8 $\quad\mathcal{E}$ = `find_nearest_neighbors`($\mathcal{T}$, $\mathfrak{D}$, $\mathcal{L}$, $\mathcal{Z}$)
9 $\quad$`// Below, `C_i` is the terminal node and `C_j` is its nearest neighbor.`
10 $\quad$**par_for** $(C_i, C_j, d) \in \mathcal{E}$ **do**
11 $\quad\quad\mathcal{L}_i$.succ = j
12 $\quad\quad$WriteMin($\mathcal{L}_i$.pred, (j, d)) `// The pair with the minimum d is written.`
13 $\quad\mathcal{M}$ = parallel_filter($\mathcal{E}$, is_R-NN())
14 $\quad$**par_for** $(C_i, C_j, d) \in \mathcal{M}$ **do**
15 $\quad\quad C_{i,j,\text{new}}$ = merge(C_i, C_j, d)
16 $\quad\quad$**if** $s > 0$ **then**
17 $\quad\quad\quad$**par_for** $(C_i, C_j, d) \in \mathcal{M}$ **do**
18 $\quad\quad\quad\quad$`update_cached_dists`(C_i, C_j, d)
19 $\quad\mathfrak{D}$.update($\mathcal{T}$, $\mathcal{M}$)
20 $\quad\mathcal{A}$ = parallel_filter($\mathcal{A}$, not_in_$\mathcal{M}$()) $\bigcup$ $\{C_{i,j,\text{new}} \mid (C_i, C_j, d) \in \mathcal{M}\}$
21 $\quad\mathcal{Z}$ = parallel_filter($\mathcal{A}$, is_terminal())
22 **return** dendrogram root node

also uses locks and requires quadratic memory for maintaining the distance matrix. In contrast, our algorithm proceeds in rounds where on each round, all chains are grown and all R-NN pairs are merged. Our algorithm is lock-free, and uses linear space as we avoid using the distance matrix. In addition, Jeon and Yoon's algorithm searches for the nearest cluster naively by computing the distances to all other clusters, whereas we have optimizations for finding the nearest clusters when growing the chain, which will be discussed in Sections 4 and 5.

3.1 ParChain Framework

We now formally describe the ParChain framework (Algorithm 1). ParChain gives rise to fast and space-efficient HAC algorithms. The main idea of ParChain is to avoid storing most cluster distances, and compute them on the fly using an optimized range search that considers only a small number of neighboring clusters. We also cache some of the cluster distances to reduce computational cost.

The input to the algorithm is a set of n points $\mathcal{P}$, a structure $\mathfrak{D}$ that is used to compute the distances between clusters based on the linkage criteria, and an integer $s \geq 0$ for the cache size. We store (cache) only $O(ns)$ cluster distances for an integer $s \geq 0$ chosen at run time. The highlighted parts of Algorithm 1 (Lines 4 and 16–18) are only required for $s > 0$, and we will discuss them in Section 5. $\mathfrak{D}$ is able to compute the distance between two clusters, and may maintain some extra data to accelerate distance computations, such as the means and variances of clusters. It also specifies the Lance-Williams formula if $s > 0$, which will be used for updating the entries of cached distances between clusters.

Initialization. On Lines 1–6, the algorithm initializes the required data structures. It first creates n dendrogram nodes (Line 1) and $\mathcal{L}$, a set of n chain nodes (Line 2). These nodes are used for the singleton clusters at the beginning. We create a set of active clusters $\mathcal{A}$, initialized to contain all of the singleton clusters (Line 3). We also create a parallel hash table for each cluster to cache cluster distances if $s > 0$ (Line 4). In each chain node, we store its successor (succ), its predecessor (pred), and the distance to its predecessor (pred.d) if there is one. All chain nodes initially do not have any successor or predecessor. $\mathcal{Z}$ represents the set of terminal nodes at the beginning of the round, and is initialized to contain the n singleton chains (Line 5). The algorithm also initializes a kd-tree on the points P. The kd-tree (Line 6) is used to accelerate nearest cluster searches.

Chain Growing and Merging. On Lines 8–15, in parallel we grow all of the chains using the information in $\mathcal{Z}$. We merge a node in $\mathcal{Z}$ with its nearest neighboring cluster if they form an R-NN pair. Specifically, on Line 8, to grow the chains we find the nearest neighbors of all current terminal nodes in $\mathcal{Z}$ using a kd-tree range search optimization, which will be described in Section 4. ParChain can quickly compute the distances of a cluster to other clusters by considering only a small number of candidates, without needing to maintain a distance matrix. The nearest neighbors and the distances are stored in $\mathcal{E}$. On Lines 10–12, we update the successor and predecessor of each terminal node in parallel to maintain the chains. If a terminal node is the nearest neighbor of multiple clusters, the WriteMin ensures that its predecessor is the cluster closest to it. Then on Line 13, we find all R-NN pairs using a parallel filter by checking for each terminal node if its successor has a successor that is itself. All R-NN pairs are stored in an array $\mathcal{M}$ with their distances. On Line 15, we create a new dendrogram node $C_{i,j,\text{new}}$ to represent the merged cluster for each R-NN pair in $\mathcal{M}$, which will have C_i and C_j as children, and store the distance between the merged clusters.

After the merges, we need to update the other data structures to prepare for next round. On Lines 16–18, we update the cache tables with new distances Section 5. On Line 19, we update the extra data structures, such as the kd-tree and clusters' mean and variance. On Line 20, we update the set of active clusters by including active clusters not in $\mathcal{M}$ (not merged this round) and the newly merged

Table 3: Worst-case work and space bounds of state-of-the-art HAC algorithms. *The authors of [43, 57, 63] do not report the work complexity.

Algorithm	Work	Space	Restrictions
ParChain	$O(n^3)$	$O(n)$	Reducibility
NN-Chain [33, 52, 65]	$O(n^2)$	$O(n^2)$	Reducibility
Generic [31, 52, 58, 59]	$O(n^2 \log n)$	$O(n^2)$	Lance-Williams
Althaus et al. [3]	$O(n^3)$	$O(n)$	Complete Linkage
Batch Processing [43, 57, 63]	*	$O(n^2)$	Disk-based

clusters from this round. Finally, on Line 21, we obtain the new set of terminal nodes $\mathcal{Z}$ using a parallel filter on the active clusters $\mathcal{A}$.

Work and Space Complexity. We summarize the time and space complexity of the state-of-the-art algorithms in Table 3. ParChain is the only algorithm that requires linear memory and works for a broad set of linkage criteria (any that satisfy the reducibility property). The main computational cost in our framework is in finding the nearest neighbors of the terminal nodes on each round, and updating the cache tables and other data structures maintained by the distance structure $\mathfrak{D}$. Sections 4 and 5 present our novel approaches for efficiently computing nearest neighbors efficiently with low space.

We now analyze the work of our framework. Let $\mathcal{Z}_i$ and $\mathcal{A}_i$ be the sets $\mathcal{Z}$ and $\mathcal{A}$ at the beginning of round i, respectively. The initialization (Lines 1–6) take $O(n \log n)$ work, dominated by the kd-tree construction work on Line 6. Lines 10–18 and Lines 20–21 take $O(\sum |\mathcal{Z}_i|)$ work across all rounds, plus the cost of all cluster distance computations, denoted by D. Line 19 takes $O(\sum |\mathcal{A}_i| \log |\mathcal{A}_i|)$ work because we need to re-construct the kd-tree of cluster centroids in this step. Finally, Line 8 takes $O(\sum |\mathcal{A}_i||\mathcal{Z}_i|)$ work because for each terminal node, we need a range query on the kd-tree of cluster centroids. Thus, we have that the work of ParChain is $O(D+M)$, where $M = \sum |\mathcal{A}_i|(|\mathcal{Z}_i| + \log |\mathcal{A}_i|)$. In the worst case, the work is $O(n^3)$, but we show in Section 6 that in practice both M and D are close to quadratic and ParChain is orders of magnitude faster than the $O(n^2)$ work algorithms [52], even using a single thread. We expect our algorithm to give improvements on most low-dimensional data sets.

The space usage of our framework is $O(n(1+s))$ because all data structures except the caches require linear memory, and the caches require $O(ns)$ memory. The kd-tree requires memory linear in the number of points in the tree.

4 NEAREST-NEIGHBOR FINDING

We will now describe how to efficiently perform nearest-neighbor finding (Line 8 of Algorithm 1) for the three linkage criteria: complete, Ward's, and average linkage. We assume that we compute distances between clusters on the fly. We describe an optimization in Section 5 that uses cache tables to store some of the distances.

While a standard nearest-neighbor search is done on points, we are searching for nearest neighbors of clusters with distances based on the linkage criteria. Our kd-trees store centroids of *clusters of points*, which we use to find nearby clusters to our query cluster. We then perform exact distance computations from our query cluster to these clusters. Unlike in standard nearest-neighbor searches, it is harder to prune in our case as the distances between clusters centroids do not necessarily correspond to distances between clusters. Instead, we compute a different search area for each cluster based on an upper bound on the distance between the query cluster and its nearest neighbor. This upper bound can be a distance between the query cluster and any other cluster. We provide a novel heuristic for finding a good upper bound on the distance to the nearest cluster, and only search within this distance in Sections 4.1–4.3. In Figure 2, we present the performance of using our optimized range query compared to the naive method of computing the distances to all other clusters to find the nearest neighbor. We see that our optimized range query gives a 7.8–1892.4x speedup on the two example data sets.

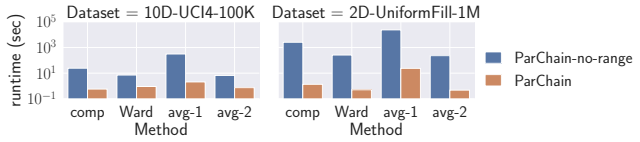

Figure 2: Runtimes on 48 cores with two-way hyper-threading of using our optimized range query compared to not using the range query and computing the distances to all other clusters on the fly to find the nearest neighbor.

Algorithm 2: Find Nearest Neighbor

Input: kd-tree $\mathcal{T}$, distance structure $\mathfrak{D}$, chains $\mathcal{L}$, and set of terminal nodes $\mathcal{Z}$
Output: nearest neighbors of nodes in $\mathcal{Z}$

1 Initialize $\mathcal{E}$ with a (Null, ∞) entry for each terminal node.
2 **par_for** $C_i \in \mathcal{Z}$ **do**
3 /* $\mathcal{L}_i$ is the chain node of C_i */
4 **if** $\mathcal{L}_i.pred \neq Null$ **then**
5 $\beta = \mathcal{L}_i$.pred.d
6 **else**
7 β = distance to a nearby cluster
8 // range query updates $\mathcal{E}$
9 range_query$(C_i, \mathcal{T}, getBall_{\mathfrak{D}}(i, \beta), \mathfrak{D}, \mathcal{E})$
10 **return** $\mathcal{E}$

Algorithm 3: RangeQuery

Input: query node C_i, kd-tree node Q, $Ball$, $\mathfrak{D}$, $\mathcal{E}$

1 **if** Q *does not overlap with Ball* **then return**
2 **if** Q is a leaf node **then**
3 **for** $\bar{x}_{C_j} \in Q$ and $\bar{x}_{C_j} \in$ Ball **do** update_nearest_neighbor(C_i, C_j, $\mathcal{E}$, $\mathfrak{D}$)
4 **else**
5 **par_do** (RangeQuery(C_i, Q.left, $Ball$, $\mathfrak{D}$, $\mathcal{E}$),
6 RangeQuery(C_i, Q.right, $Ball$, $\mathfrak{D}$, $\mathcal{E}$))

Algorithm 4: Update Nearest Neighbor

Input: cluster C_i, cluster C_j, distance structure $\mathfrak{D}$, cache tables $\mathcal{H}_i$ and $\mathcal{H}_j$, and set $\mathcal{E}$

1 **if** $s > 0$ **then**
2 d = get_cached_dist(i, j)
3 **if** $d \neq$ *NOT_FOUND* **then**
4 WriteMin ($\mathcal{E}$[cid$_i$], (cid$_j$, d))
5 WriteMin ($\mathcal{E}$[cid$_j$], (cid$_i$, d))
6 **return**
7 $d = \mathfrak{D}$.dist(C_i, C_j)
8 **if** $s > 0$ **then** insert {cid$_i$, i} into $\mathcal{H}_j$ and {cid$_j$, j} into $\mathcal{H}_i$
9 WriteMin($\mathcal{E}$[cid$_i$], (cid$_j$, d))
10 WriteMin($\mathcal{E}$[cid$_j$], (cid$_i$, d))

Algorithm. Given the kd-tree $\mathcal{T}$ built on the centroid of clusters, distance structure $\mathfrak{D}$, chain nodes $\mathcal{L}$, and set of terminal nodes $\mathcal{Z}$, Algorithm 2 finds the nearest cluster and the distance to it for each terminal node's cluster and stores them in $\mathcal{E}$.

When finding the nearest neighbor of cluster C_i, we search all points within some ball $Ball(\bar{x}_{C_i}, r)$ obtained from $getBall_{\mathfrak{D}}(i, \beta)$,

which is a ball centered at centroid $\bar{x}_{C_i}$ with radius r. The radius r depends on the linkage method of $\mathfrak{D}$ and an distance β between C_i and another cluster (Lines 2–9 of Algorithm 2). Since we use centroid distances, $\mathfrak{D}$ rebuilds $\mathcal{T}$ to be a kd-tree of only the centroids of current clusters at the end of each round of Algorithm 1 (Line 19 of Algorithm 1). If C_i has a predecessor, we set the distance β to be the distance between C_i and its predecessor (Lines 4–5 of Algorithm 2). Otherwise, we find the distance to another cluster for computing the radius for the search. Specifically, we use the distance to the cluster whose centroid is the closest to the current cluster, which can be computed using a parallelized nearest-neighbor query on the kd-tree of centroids [9] (Lines 6–7 of Algorithm 2).

For the range query on Line 9 of Algorithm 2, we use the parallel range query in Algorithm 3. Given a query cluster C_i, a kd-tree node Q, a ball representing the range, a linkage function $\mathfrak{D}$, and a set $\mathcal{E}$ of pairs of nearest neighbor candidates and distances of terminal nodes, the algorithm processes all of Q's points that are in the ball to update the nearest neighbor candidates in $\mathcal{E}$. Since we only process the points in the ball, on Line 1, the range query terminates if the bounding box of the tree node does not overlap with the query range. Otherwise, the range query will either process all of the points both in the node and in the ball using the *update_nearest_neighbor* subroutine (Algorithm 4) if it is a leaf node (Lines 2–3 of Algorithm 3) or recurse on its two children in parallel (Lines 4–6 of Algorithm 3).

In each update_nearest_neighbor(C_i, C_j) call, we check if some cluster C_j is closer to C_i than its current nearest neighbor candidate, and if so we update C_i's nearest neighbor in $\mathcal{E}$ with Algorithm 4. We also update C_j's nearest neighbor to be C_i if C_i is closer to C_j than its current nearest neighbor candidate. In Algorithm 4, if $s = 0$, we will compute the distance between C_i and C_j on the fly (Lines 7, 9, and 10). If $s > 0$, we will first check the cache and use a cached distance if possible (we describe more details in Section 5).

As an optimization for the first round, we know that the distances between clusters is exactly the same as the distances between their centroids in the first round, and thus we can efficiently prune searches in the kd-trees. Therefore, we use an all-nearest-neighbor query for the first round, which we implemented by parallelizing the dual-tree traversal algorithm by March et al. [47]. At a high level, our algorithm processes recursive calls of the dual-tree traversal in parallel and uses WriteMin to update the nearest neighbors of points. A dual-tree traversal allows more pruning than when running individual nearest neighbor queries for each point.

In the rest of the section, we will describe the radius of the search ball for each linkage method. We will show that a cluster's nearest neighbor must have its centroid inside the ball.

4.1 Ward's Linkage

In Ward's linkage, $\Delta(C_i, B)_{Ward} = \sqrt{\frac{2|C_i||B|}{|C_i|+|B|}||\bar{x}_{C_i} - \bar{x}_B||^2}$. For the range query, we can use a ball with radius $r = \beta\sqrt{\frac{|C_i|+n_{\min}}{2n_{\min}|C_i|}}$, where β is the distance between C_i and some cluster A and $n_{\min}$ is the size of the smallest current cluster. We can obtain $n_{\min}$ using a parallel reduce on the sizes of all clusters. Figure 3(a) illustrates the range search for Ward's linkage.

Since $\beta = \Delta_{Ward}(C_i, A)$, any cluster B that is closer to C_i than A must have $\|\bar{x}_{C_i} - \bar{x}_B\|^2 \leq \beta^2 \frac{|C_i|+|B|}{2|C_i||B|}$. The right-hand side of the inequality becomes smaller for larger $|B|$, thus we can upper bound the distance between C_i's centroid and B's centroid (i.e., $\|\bar{x}_{C_i} - \bar{x}_B\|$) by $r = \beta\sqrt{\frac{|C_i|+n_{\min}}{2n_{\min}|C_i|}}$. Therefore, no cluster outside of the ball centered at $\bar{x}_{C_i}$ with radius r can be closer to C_i than A, and thus we only need to search for C_i's nearest neighbor inside this ball.

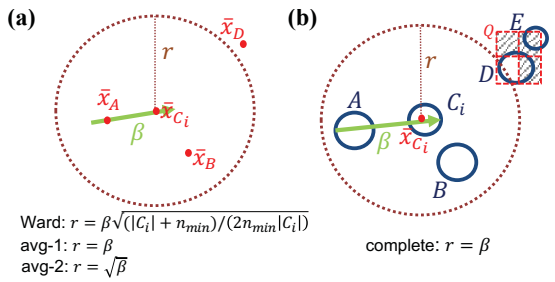

Figure 3: Search area ball with radius r in range queries for the linkage metrics. The blue circles are clusters. The red points are cluster centroids. The green arrows specify the predecessor A of C_i, and the distance between them is β. The red boxes are the bounding boxes of a kd-tree node Q and its four descendants. In (a), $\Delta(C_i, B)$ is computed because $\bar{x}_B$ is in the ball; however, $\Delta(C_i, D)$ is not computed because $\bar{x}_D$ is outside the ball. $n_{\min}$ is the size of the smallest current cluster. In (b), $\Delta(C_i, B)$ is computed because B is completely in the ball. $\Delta(C_i, D)$ will not be computed, because the three shaded bounding boxes of kd-tree nodes do not intersect with the ball, and so some of D's points will not be included in the count. We only compute the distance to a cluster if all of its points are included in the count.

4.2 Average Linkage

In average linkage, the distance between two clusters is the average distance between all pairs of points, one from each cluster. For the range query, we use a ball centered at $\bar{x}_{C_i}$ with radius $r = \beta$ and $r = \sqrt{\beta}$ for Euclidean distance metric and squared Euclidean distance metric, respectively. As before, β is the distance between C_i and some cluster A. The nearest neighbor B of C_i must have its centroid inside the ball, i.e., $\|\bar{x}_{C_i} - \bar{x}_B\| \leq \Delta_{avg\text{-}1}(C_i, B)$. The proof is provided in the full version of our paper [69].

Similarly, for the squared Euclidean metric, we have $\|\bar{x}_{C_i} - \bar{x}_B\|^2 \leq \Delta_{avg\text{-}2}(C_i, B)$, which leads to $\|\bar{x}_{C_i} - \bar{x}_B\|^2 \leq \Delta_{avg\text{-}2}(C_i, B) \leq \Delta_{avg\text{-}2}(C_i, A) = \beta = r^2$. $\|\bar{x}_{C_i} - \bar{x}_B\|^2 \leq \Delta_{avg\text{-}2}(C_i, B)$ holds since variances are non-negative and $\Delta(C_i, B)_{avg\text{-}2} = \|\bar{x}_{C_i} - \bar{x}_B\|^2 + \frac{\mathrm{Var}(C_i)}{|C_i|} + \frac{\mathrm{Var}(B)}{|B|}$. Figure 3(a) illustrates the range search for average linkage with the Euclidean and squared Euclidean distance metrics.

4.3 Complete Linkage

In complete linkage, the distance between two clusters is the maximum distance between a pair of points, one from each cluster. For the range query, we use a ball with radius $r = \beta$ centered at centroid $\bar{x}_{C_i}$, where β is the distance between C_i and some cluster. By definition of the complete linkage function, the cluster distance must be no smaller than distance between their centroids, and so the nearest neighbor of C_i has its centroid within the search ball.

Range Query Optimization. For complete linkage, we can reduce the number of cluster distance computations by only computing the distance to a cluster if it is completely within the search ball. With this observation, we can optimize the algorithm by keeping the kd-tree to be $\mathcal{T}_P$, the kd-tree of all points, and avoiding updating it to be the kd-tree of centroids on every round. Figure 3(b) illustrates

the optimized range search for complete linkage. We will prove the correctness of this optimization at the end of the subsection.

Since now $\mathcal{T}$ is always $\mathcal{T}_P$, we need to slightly modify Algorithm 2 and Algorithm 3. On Line 7 of Algorithm 2, we search for the point $p \notin C_i$ closest to $\bar{x}_i$ in $\mathcal{T}_P$, and let β be the distance between C_i and the cluster of this point. We can use a parallel union-find structure [22] to ignore points in C_i. For Algorithm 3, the range query might be able to terminate before Line 2 if the tree node satisfies some conditions. For each range search, we keep a count that eventually upper bounds the number of points within the ball for each cluster. In each for-loop on Line 3 of Algorithm 3, we now loop over points p instead of centroids, and we atomically increment the count for p's cluster by 1 because this means we have found one more point in this cluster that is within the ball. The cluster IDs can also be maintained and queried using the parallel union-find data structure. Right before Line 2 of Algorithm 3, if all points in the kd-tree node Q are from the same cluster C, we atomically increment the count of cluster C by the size of the node and prune the search; otherwise, we continue the search and recurse on the children. This gives an upper bound on the number of points in the cluster within the ball, because the ball lies inside the kd-tree bounding boxes traversed.

We preprocess the tree such that in the range search we can determine in constant time if all points in the node are from the same cluster, and if so which cluster it is. Specifically, we mark the kd-tree nodes with a cluster ID if all points in the node are from the same cluster, or with NULL if the points in the node belong to multiple clusters. This can be computed by recursively checking the ID of the two children of a node starting from the root, and storing the cluster ID of the children if all of their points are from the same cluster. We update this information on every round.

After processing a point or a node, if we incremented the count of a cluster C, we check if the count of C is equal to the size of C. If so, this means that all of C's points may be within distance $r = \beta$. In this case, we compute the distance between the C_i and this cluster, and use a WriteMin to update the nearest neighbor of C_i in $\mathcal{E}$ (Lines 7, 9, and 10 of Algorithm 4).

Finally, we show below that C_i's nearest neighbor B must be a cluster completely within search area by claiming that clusters with points outside the ball must have a distance larger than r to C_i. Since r is the distance between C_i and some cluster, B must have a distance no larger than r to C_i. Suppose the distance of the furthest point pair between C_i and B is $\Delta_{comp}(C_i, B) = d(p, q)$. Since the average Euclidean distance between points in two clusters is not smaller than the distance between their centroids (shown in the full paper [69]), applying this property to C_i and $\{q'\}$ for any point $q' \in B$, we see there must exists some $p' \in C_i$ such that $d(\bar{x}_{C_i}, q') \leq d(p', q')$. Since (p, q) is the furthest point pair, we have that $d(\bar{x}_{C_i}, q') \leq d(p', q') \leq d(p, q)$. Thus, if $\Delta_{comp}(C_i, B) = d(p, q) \leq r$, then all points in B must be within $Ball_{comp}(\bar{x}_{C_i}, r)$. As a result, we only need to consider a cluster as the nearest neighbor candidate of C_i and actually compute the distance to it if all of its points are inside the ball.

Dual-Tree Traversal. When computing cluster distances (Line 7 of Algorithm 4) for complete linkage, we use our parallel dual-tree traversal algorithm described earlier in the section. We need to find the distance of the farthest pair of points, and so we use WriteMax instead of WriteMin for storing the farthest distance seen. In order to perform the dual-tree traversals, $\mathfrak{D}$ creates a kd-tree for each cluster at the end of each round (Line 19 in Algorithm 1).

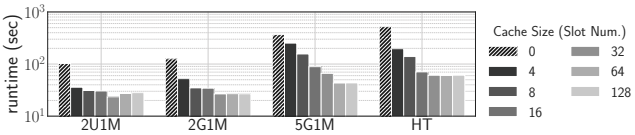

Figure 4: Running times of using ParChain with average linkage and the Euclidean distance metric using 48-cores with two-way hyper-threading for varying cache sizes (values of s). The data sets are labeled on the x-axis and are described in Section 6 (caption of Table 4).

5 CACHING INTER-CLUSTER DISTANCES

For some linkage function and metric combinations, such as average linkage with the Euclidean distance metric, computing inter-cluster distances can be expensive. We can avoid some recomputations of cluster distances by caching some previously computed distances for each cluster C_i using a cache table $\mathcal{H}_i$, represented using a parallel hash table. Users can specify a constant size s of each cache based on the available memory. The total memory usage is $O(n(1+s))$, which is less than the quadratic memory required by the distance matrix approaches. Sometimes, a larger table will lead to faster computations because we can cache more distances and avoid more recomputations. Due to the optimizations in Section 4, the distances that we compute will tend to be close to C_i, and hence stored in $\mathcal{H}_i$. These distances are more likely to be reused in future nearest neighbor queries.

We present a comparison of running times of average linkage with the Euclidean distance metric on several data sets using different cache sizes in Figure 4. We see that using caching improves the running times by up to a factor of 8.98x compared to not using caching. We found similar trends on other data sets. We will discuss more about our implementation's memory usage in Section 6.

In the rest of the section, we assume $s > 0$ and describe how to query cluster distances from the cache tables, insert new entries after computing cluster distances during nearest neighbor queries, and update the tables after merging clusters.

Querying and Inserting Distances between Clusters. The cache tables can be used to reduce cluster distance computations because we can insert the computed distances to the tables and query for them if we want to use them again. Now we describe how the cache is used to update the nearest neighbor candidate in the nearest neighbor search (Algorithm 4). With $s > 0$, we might have already cached the distance $\Delta(C_i, C_j)$ in one or both of the tables $\mathcal{H}_i$ and $\mathcal{H}_j$ when we find C_j in C_i's range. Therefore, we first query for the distance in the cache tables (Line 2), and only compute the distance if the return value is NOT_FOUND; otherwise we can directly use the queried distance to update the nearest neighbor candidate in $\mathcal{E}$ (Lines 3–6). If we compute the distance (Line 7), we will attempt to insert it into both of the tables (Line 8). The insertion may fail for a cache table if it is full, i.e., it already contains s entries. Since we insert distances between C_i and the clusters that are within its search range in all rounds so far, the distances stored in $\mathcal{H}_i$ are likely to be between C_i and nearby clusters. Thus in later rounds, these cached distances are more likely to be queried. On Lines 9–10, WriteMin updates the nearest neighbors of C_i and C_j in $\mathcal{E}$.

Algorithm 5: Updating Cached Distance

Input: C_k merged from C_i and C_j, and distance structure $\mathfrak{D}$

1 // C is either C_i (from $\mathcal{H}_i$) or C_j (from $\mathcal{H}_j$)
2 **par_for** $(d' = \Delta(C, C_\ell)) \in \mathcal{H}_i \cup \mathcal{H}_j$ **do**
3 **if** $\{C, C_\ell\} == \{C_i, C_j\}$ **then** continue
4 **if** C_ℓ *is merged in this round* **then**
5 C_g = the cluster that C_ℓ merged into
6 $d = \mathfrak{D}$ computes $\Delta(C_k, C_g)$ from the distances among C_i, C_j, and the children of C_g using d'
7 **else**
8 $C_g = C_\ell$
9 $d = \mathfrak{D}$ computes $\Delta(C_k, C_g)$ from the distances among C_i, C_j, and C_ℓ using d'
10 Insert d into $\mathcal{H}_k$ and $\mathcal{H}_g$

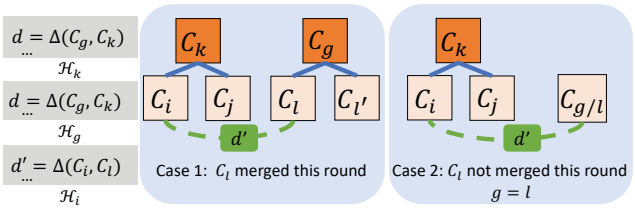

Figure 5: An illustration of the cache table update in Algorithm 5. The gray boxes show an entry in each of cache tables $\mathcal{H}_k$, $\mathcal{H}_g$, and $\mathcal{H}_i$. The dark orange boxes are clusters merged in this round; the light orange boxes are clusters merged in previous rounds. The blue lines connect dendrogram children to their parent. The dotted green lines and boxes mark the cached distance between clusters. In case (1), C_ℓ is merged in this round into C_g; in case (2), C_ℓ is not merged in this round, and it is the same as C_g.

When querying $\Delta(C_i, C_j)$ with get_cached_dist(i, j) (Line 2), we search for the entry with key i in $\mathcal{H}_j$, and the entry with key j in $\mathcal{H}_i$. If in a cache table, the key does not exist, then the query fails. If the queries in both tables fail, we return NOT_FOUND. If the search is successful in one of the tables, we return the distance stored in the table. We search in both cache tables since the caches are of limited size, and so the distance could potentially be stored in just one of the two tables.

Updating Cache Tables after Merging Clusters. We now describe how to update the entries in the cache tables after clusters are merged (Lines 16–18 of Algorithm 1). If during a round C_i and C_j are merged into a new cluster C_k, we will try to compute the distance between C_k and all clusters C_g whose subclusters' distance(s) with C_i or C_j are stored in $\mathcal{H}_i \cup \mathcal{H}_j$. These distances can be used to accelerate the computation of $\Delta(C_k, C_g)$ using the Lance-Williams formula described in Section 2.1.

The update_cached_dists function called on Line 18 of Algorithm 1 is presented in Algorithm 5. On Line 2, we loop over the distances d' in the cache tables of $\mathcal{H}_i$ and $\mathcal{H}_j$. Without loss of generality, assume $d' = \Delta(C_i, C_\ell)$ is a distance in $\mathcal{H}_i$ between C and some cluster C_ℓ (the case for an entry in $\mathcal{H}_j$ is similar). Line 3 skips over the entries that represent distances between C_i and C_j, since they are now merged. Otherwise, there are two cases. In case (1), C_ℓ is also a cluster merged in this round (Lines 4–6), and we let C_g be the cluster that C_ℓ merged into. We compute the new distance $\Delta(C_k, C_g)$ on Line 6 and insert the new distance into the caches of both clusters C_k and C_g on Line 10. In case (2), C_ℓ is not a new cluster merged in this round (Lines 7–9), and we have $C_g = C_\ell$. We can also use d' to accelerate the computation of $\Delta(C_k, C_g)$. Figure 5 illustrates one loop of the algorithm where the entry being considered is $d' = \Delta(C_i, C_\ell) \in \mathcal{H}_i$ (shown in the bottom gray box). In both cases, we store entry $d = \Delta(C_k, C_g)$ computed from d' into both $\mathcal{H}_k$ and $\mathcal{H}_g$ on Line 10.

Now we describe the update rule for computing d. For case (2), we can just directly apply the Lance-Williams formula [38] introduced in Section 2 and compute $\Delta(C_i \cup C_j, C_g)$ from $\Delta(C_i, C_g)$, $\Delta(C_j, C_g)$, and $\Delta(C_i, C_j)$. For case (1), we can apply the Lance-Williams formula and compute $\Delta(C_i \cup C_j, C_\ell \cup C_{\ell'}) = \Delta(C_k, C_\ell \cup C_{\ell'})$ from $\Delta(C_k, C_\ell)$, $\Delta(C_k, C_{\ell'})$, and $\Delta(C_\ell, C_{\ell'})$, where $C_{\ell'}$ is the cluster that C_ℓ merges with to form C_g. To compute $\Delta(C_k, C_\ell)$ and $\Delta(C_k, C_{\ell'})$, which are not cached since C_k is a newly merged cluster, we can apply the Lance-Williams formula again since $C_k = C_i \cup C_j$. Below we give the update rule for average linkage with Euclidean distance metric as an example. For case (2), where $C_g = C_\ell$, we have

$$d = \frac{|C_i|}{|C_i| + |C_j|}\Delta_{\text{avg-1}}(C_i, C_g) + \frac{|C_j|}{|C_i| + |C_j|}\Delta_{\text{avg-1}}(C_j, C_g). \quad (1)$$

For case (1), let C_ℓ and $C_{\ell'}$ be C_g's children. We have

$$d = \frac{|C_i||C_\ell|}{|C_k||C_g|}\Delta_{\text{avg-1}}(C_i, C_\ell) + \frac{|C_j||C_\ell|}{|C_k||C_g|}\Delta_{\text{avg-1}}(C_j, C_\ell) + \frac{|C_i||C_{\ell'}|}{|C_k||C_g|}\Delta_{\text{avg-1}}(C_i, C_{\ell'}) + \frac{|C_j||C_{\ell'}|}{|C_k||C_g|}\Delta_{\text{avg-1}}(C_j, C_{\ell'}). \quad (2)$$

If the distances between $C_{\ell'}$ (C_g) and one or both of C_i and C_j are also cached in case (1) (case (2)), we can also query them and accelerate the computation of d by avoiding some distance recomputation. For example, in update rule (1), if $d' = \Delta_{\text{avg-1}}(C_i, C_g)$ is cached, we can compute $d = \Delta_{\text{avg-1}}(C_k, C_g)$ by $d = \frac{|C_i|}{|C_i|+|C_j|}d' + \frac{|C_j|}{|C_i|+|C_j|}\Delta_{\text{avg-1}}(C_j, C_g)$. If $\Delta(C_j, C_g)$ is also cached, we can query for it and compute d in constant time; otherwise, we need $|C_j||C_g|$ point distance computations to find d, which is less than the $|C_k||C_g|$ point distance computation required by a brute force method.

During the nearest neighbor range search, two clusters C_i and C_j might find each other as nearest neighbor candidates, and both want to compute $\Delta(C_i, C_j)$ in parallel. A similar situation can happen when updating the cache entries for C_k and C_g. Our implementation avoids these duplicate distance computations by having each cluster first insert a special entry into the hash table $\mathcal{H}_{\min(i,j)}$ (or $\mathcal{H}_{\max(i,j)}$ if $\mathcal{H}_{\min(i,j)}$ is full), and then only compute the distance if the insertion was successful. The special entry can only be successfully inserted once for each pair of clusters C_i and C_j, and so $\Delta(C_i, C_j)$ will only be computed once.

6 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. We use the `g++` compiler (version 7.5) with the `-O3` flag, and use Cilk Plus, which is supported in `g++`, for parallelism in our code [40]. For parallel experiments, we use `numactl -i all` to balance the memory allocation across nodes. We also perform three runs of each parallel experiment and report the smallest running time. We allocate a maximum of 15 hours for each run of a running time test, and do not report the times for tests that exceed this limit.

We test the following implementations for HAC. We refer to complete linkage as ***comp***, Ward's linkage as ***Ward***, average linkage with Euclidean distance metric as ***avg-1***, and average linkage with squared Euclidean distance metric as ***avg-2***.

- ***PC*** Our parallel ParChain framework is implemented in C++, using the range query and caching optimizations.
- ***PC-mr*** A parallel C++ NNC implementation that uses a distance matrix and merges all R-NNs in each round. All cluster distances are obtained from the distance matrix. It uses the range query optimization to find the nearest neighbors.
- ***PC-m*** A naive parallel C++ NNC implementation that uses a distance matrix and merges all R-NNs in each round. All cluster distances are obtained from the distance matrix. A parallel reduce is used over the distance matrix rows to find the nearest neighbor instead of using a range query.
- ***scipy (sc)*** [51] Scipy's serial implementation in Cython, which uses the NNC algorithm with a distance matrix for all of the linkage criteria tested.
- ***sklearn (sk)*** [59] Scikit-learn's serial implementation in Cython, which uses a distance matrix. It has a heap for all distances and merges the global closest neighbor pair on each round.
- ***fastcluster*** [52] A serial C++ implementation of HAC with a Python interface. It contains two approaches of implementations of HAC—one generic implementation (***fc-gen***) uses the naive algorithm where the global R-NN pair is merged in each round, and the other (***fc-nnc***) is based on the NNC algorithm using a distance matrix. For Ward's linkage, fastcluster has a linear space implementation for the naive method that work by computing the cluster distances on the fly using the cluster centroids. We also wrote linear-space implementations for the NNC algorithm that compute distances on the fly for Ward's linkage and average linkage with squared Euclidean distance metric. We report the running time of the linear space implementation when available.
- ***fastprotein (fp)*** [31] A parallel C++ implementation that parallelizes the naive NNC algorithm by computing the global R-NN and updating the distance matrix in parallel on each of the $n-1$ rounds. It only supports complete and average linkage.
- ***Jeon (Je)*** [33] A parallel C++ implementation of the parallel NNC algorithm by Jeon and Yoon. [33], which only supports average linkage with the Euclidean distance metric.
- ***Althaus (Al)*** [3] Our parallel C++ implementation of Althaus et al.'s complete linkage algorithm that uses linear memory.

Data Sets. We use both synthetic and real-world datasets, all of which fit in the RAM of our machine. Let n be the number of points. The ***GaussianDisc*** data set contains points inside a bounding hypergrid with side length $5\sqrt{n}$. 90% of the points are equally divided among five clusters, each with a Gaussian distribution. Each cluster has its mean randomly sampled from the hypergrid, a standard deviation of $1/6$, and a diameter of $\sqrt{n}$. The remaining points are randomly distributed. The ***UniformFill*** data set contains points distributed uniformly at random inside a bounding hypergrid with side length $\sqrt{n}$. We generate the synthetic data sets with 10 million points for dimensions $d = 2$ and $d = 5$.

We also use two existing simulation datasets. ***UCI1*** [7] is a 10-dimensional data set with $19,020$ data points. This data set is generated to simulate registration of high energy gamma particles [29]. ***UCI4*** [35] is a 10-dimensional data set with $100,000$ data points. This data set has a pseudo-periodic time series for each of its dimension, and hence is likely to form long chains.

We use the following real-world data sets. ***GeoLife*** [71] is a 3-dimensional data set with $24,876,978$ data points. This data set contains user location data, and is extremely skewed. ***HT*** [30] is a 10-dimensional data set with $928,991$ data points containing home sensor data. ***CHEM*** [25] is a 16-dimensional data set with $4,208,261$ data points containing chemical sensor data.

When referring to the data sets in this section, we use a prefix to indicate its dimensionality and suffix to indicate its size. To obtain smaller data sets, we randomly sample from the corresponding larger data sets. The letter "U" indicates UniformFill and "G"indicates "GaussianDisc".

Cache Sizes. In our experiments, we use a cache size of $s = 64$ for avg-1 and $s = 0$ for complete, Ward, and avg-2 except otherwise noted. The choice of $s = 64$ will be discussed in more detail in Section 6.3 along with the benefit of our range query and caching optimizations. We use $s = 0$ for complete, Ward, and avg-2 to show our framework achieves good performance on cheaper linkage criteria even without caching.

6.1 Comparison with Other Implementations

Figure 6 shows the running times vs. number of threads for all of the serial and parallel implementations on three small data sets (2D-GaussianDisc-10K, 10D-UCI1-19K, and 10D-UCI4-100K). Implementations with a single data point are serial. We only compare them on the small data sets because the algorithms that require quadratic memory run out of memory for larger data sets.

We see that our implementation PC almost always outperforms existing implementations across all thread counts. Even the version of our algorithm using the distance matrix without the range query optimization (PC-m) is faster than all other implementations at higher thread counts. Unlike the existing parallel implementations, fastprotein (fp) and Jeon (Je), our implementations are more scalable since we merge all R-NN pairs on each round, and do not use locks. On the small data sets, using 48 cores with hyper-threading, PC is 5.8–88.0x faster than fp and 37.5–110.1x faster than Je. Table 4 shows the running times for PC, fc-gen, and fc-nnc on larger data sets (Je does not scale to these data sets due to its quadratic memory requirement), and we see that PC is 64.77–733.90x faster than fastcluster on these data sets. On a single thread, we find that PC is 2.19–47.92x faster than the next fastest implementation (except on 10D-UCI1-19K for complete linkage, where PC is 1.58x slower than fc-nnc and fc-gen).

In Figure 6, PC shows limited scalability on higher thread counts, because these data sets are small and the overhead of using more threads is high relative to the work of the algorithm. However, in the next subsection, we show that PC is able to achieve higher parallel scalability on larger data sets.

6.2 Scalability

Scalability with Thread Count. Table 4 and Figure 7 present the runtime and scalability of PC on different numbers of threads for larger data sets, which most existing implementations do not scale to. For average linkage with the Euclidean distance metric, the speedups for several data sets are not shown since the single-threaded experiments timed out. We see that using 48 cores with

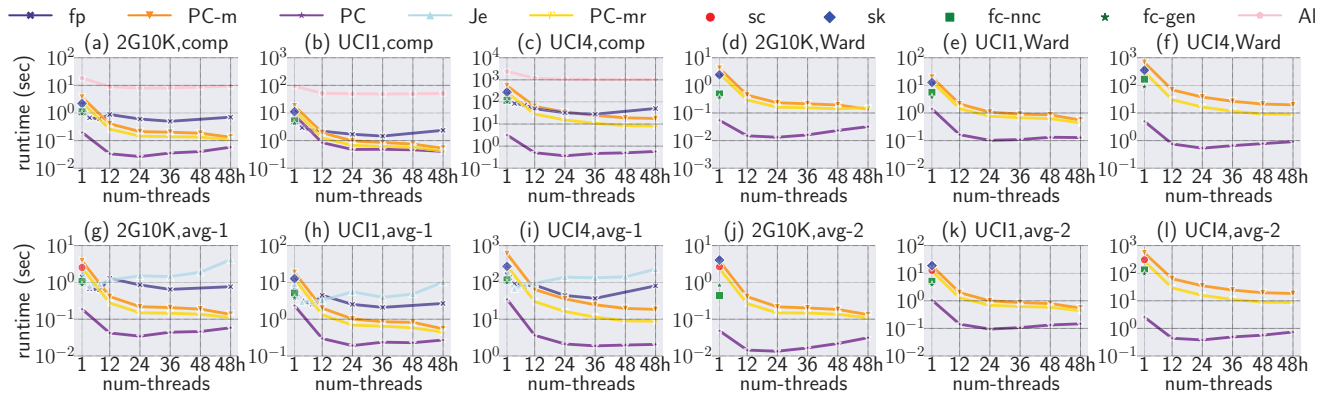

Figure 6: Runtimes (seconds) of our algorithms compared with other implementations with varying thread counts. (48h) indicates 48 cores with two-way hyper-threading. Implementations with a single data point are serial. Je only supports average linkage with Euclidean distance and fp only supports complete and average linkage with Euclidean distance. sc and sk have very similar running times and overlap on some plots. For avg-2, sk runs out of memory for the UCI4 data set. Our algorithm PC is faster than all other implementations for all number of threads, except on a single thread for the UCI1 data set using complete linkage. See Table 4 and Figure 7 for running times and scalability of PC on larger data sets.

Table 4: Runtimes (seconds) and self-relative parallel speedups of fastcluster and PC. "PC-1" is our runtime on 1 thread and "PC-48h" is our runtime using 48 cores with two-way hyper-threading. PC on avg-1 timed out on several larger data sets. For fastcluster on several larger data sets, we only have runtimes for Ward and avg-2 because it requires quadratic memory for comp and avg-1 and runs out of memory. For Ward, we report fc-gen because it is faster than fc-nnc. "–" means that the running time exceeds 15 hours. For the data set names, the first number indicates dimension, the letter "U" indicates UniformFill, "G" indicates "GaussianDisc", "1M" indicates 1 million data points, and "10M" indicates 10 million data points.

		2U1M	2G1M	5U1M	5G1M	HT	2U10M	2G10M	5U10M	5G10M	CHEM	Geolife
comp	PC-1 (sec)	47.21	50.47	2119.10	2282.00	56.80	677.88	609.08	53008.00	49052.00	4972.90	1948.90
	PC-48h (sec)	1.34	1.61	40.40	46.74	3.68	19.01	21.29	977.47	1038.82	126.54	123.34
	self-speedup	35.12	31.39	52.45	48.83	15.42	35.66	28.61	54.23	47.22	39.30	15.80
Ward	fc-gen (sec)	3537.11	3845.04	7284.11	6676.75	8760.04	–	–	–	–	–	–
	PC-1 (sec)	11.31	12.23	79.11	103.09	23.13	175.70	156.53	1475.50	1230.30	1096.40	681.63
	PC-48h (sec)	0.50	0.52	1.95	2.69	1.50	5.69	5.20	30.36	34.99	24.83	32.14
	self-speedup	22.42	23.70	40.52	38.36	15.43	30.90	30.11	48.59	35.17	44.16	21.21
avg-1	PC-1 (sec)	859.22	857.07	1627.80	1734.10	2652.70	–	–	–	–	–	–
	PC-48h (sec)	23.59	26.83	38.10	45.27	58.25	2969.93	3206.48	6323.56	5772.03	2323.38	19213.60
	self-speedup	36.42	31.95	42.72	38.30	45.54	–	–	–	–	–	–
avg-2	fc-nnc (sec)	4602.83	4022.04	8907.50	11425.99	13244.60	–	–	–	–	–	–
	PC-1 (sec)	10.28	11.11	62.10	81.21	18.05	159.65	141.52	1146.20	955.77	833.06	575.39
	PC-48h (sec)	0.47	0.49	1.65	2.27	1.31	5.30	4.80	24.60	28.51	20.93	28.34
	self-speedup	21.78	22.76	37.64	35.73	13.75	30.14	29.49	46.60	33.52	39.79	20.30

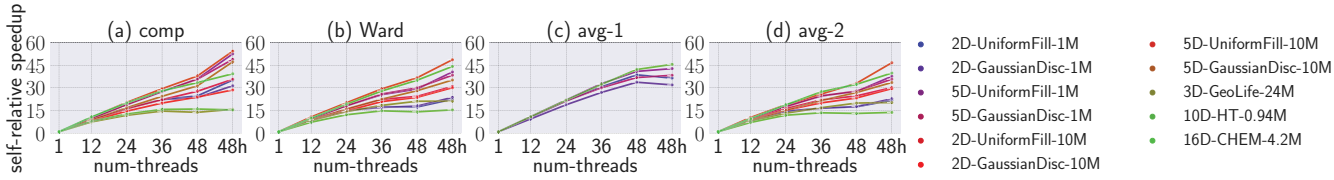

Figure 7: Self-relative parallel speedup vs. thread counts for complete, Ward's, and average linkage using our "PC" algorithm on large datasets. (48h) indicates 48 cores with two-way hyper-threading. For average linkage with the Euclidean distance metric, the speedups for several data sets are not shown since the single-threaded experiments timed out.

two-way hyper-threading, PC achieves 15.42–54.23x speedups on complete linkage, 15.43–44.16x speedups on Ward's linkage, 31.95–45.54x speedups on average linkage with Euclidean distance, and 13.75–46.6x speedups on average linkage using squared Euclidean distance. From Figure 7, we can see that on most data sets, our algorithm keeps scaling up until 48 threads.

Scalability with Data Size. Figure 8 shows the runtimes of our algorithm PC on three data sets of varying sizes using 48 cores with two-way hyper-threading. We observe that PC scales well with data

Figure 8: Runtimes (seconds) of our algorithm "PC" on three data sets of varying sizes using 48 cores with two-way hyper-threading. The "Ward" and "avg-2" lines overlapped because they have similar runtimes. The results show that the scalability is very similar across datasets.

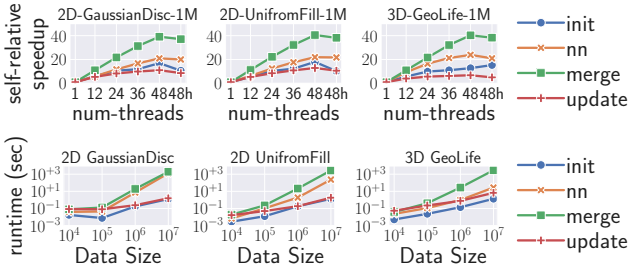

Figure 9: Self-relative parallel speedup and runtimes (seconds) of steps of avg-1 using 48 cores with two-way hyper-threading on different thread counts and data set sizes. From Algorithm 1, "init" corresponds to Lines 1–6; "nn" corresponds to Lines 8–12; "merge" corresponds to Lines 13–18; and "update" corresponds to Lines 19–21.

set size. The scalability is better for comp, Ward, and avg-2 than for avg-1 because avg-1 always requires quadratic work to compute cluster distances, while Ward and avg-2 require constant time to compute them and comp usually requires less than quadratic time to compute them due to pruning.

Runtime Decomposition. We now describe the breakdown of running time across different steps of ParChain, as well as the scalability of each step. Figure 9 shows the speedups and running times of steps of avg-1 using 48 cores with hyper-threading on different data sets. From Algorithm 1, "init" corresponds to Lines 1–6 where we initialize the data structures; "nn" corresponds to Lines 8–12 where we find the nearest neighbors of all terminal nodes and update the chains; "merge" corresponds to Lines 13–18 where we merge the R-NNs and update the cache tables if $s > 0$; and "update" corresponds to Lines 19–21 where we update the data structures to prepare for next round.

From Figure 9 (top), we see that the "nn" and "merge" steps are more scalable with respect to thread count than the "update" and "init" steps. The reason is that in the "nn" step, we find the nearest neighbor of all terminal nodes in parallel, and in the "merge" step, we merge all R-NNs in parallel. The numbers of terminal nodes and R-NNs are usually much larger than the number of available threads, and thus there is a lot of opportunity for parallelism. The "init" and "update" step are less scalable because they have less work to be divided across threads.

From Figure 9 (bottom), we see that the "nn" and "merge" steps are less scalable with respect to data size than the "update" and "init" steps. This is because the "nn" and "merge" steps asymptotically dominate the work of the whole algorithm.

6.3 Analysis of Our Framework

We now discuss the effects of our range query and caching optimizations and show our running time is close to quadratic in practice. From Figure 6, PC-mr is 1.67x faster on average than PC-m on 48 cores with two-way hyper-threading, which shows that the benefit of using our optimized range query is larger than its overhead even on these small datasets. PC is 15.06x faster on average than PC-mr because although PC needs to compute some distances on the fly, PC avoids the overhead of computing the distance matrix and updating the matrix in each round. This shows the benefit of avoiding a distance matrix.

To further show the benefit of our range query and caching optimizations, we measure the maximum average cache usage

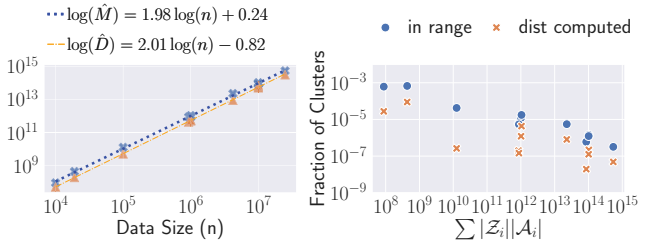

Figure 10: Statistics for avg-1 on different datasets with $s = 64$. (Left) Blue crosses denotes $M = \sum(|\mathcal{Z}_i| + \log|\mathcal{A}_i|)|\mathcal{A}_i|$ for different data sets. Orange triangles denote the total number of distance computations between points (D). $\hat{M}$ and $\hat{D}$ are least squares fitted lines of M and D, respectively. (Right) Fraction of clusters in the range queries and fraction of distance computations required between clusters.

Table 5: Memory usage (MB) vs. data set size for 2D-GaussianDisc data sets for the different implementations. The smallest memory for each linkage criteria and data set is in bold.

	n	fc-gen	fc-nnc	PC	PC-m	PC-mr	sc	sk	fp	Je	Al
comp	1K	17.5	17.5	5.8	8.5	8.7	21.6	31.4	4.2	–	**3.40**
	3K	49.5	49.5	11.1	41.4	42.1	85.8	95.5	36.4	–	**7.29**
	10K	414.6	414.4	27.6	407.7	410.2	814.2	824.0	401.1	–	**20.75**
Ward	1K	15.4	15.4	**3.5**	8.6	8.7	21.6	31.4	–	–	–
	3K	20.2	20.2	**4.8**	41.4	41.9	85.8	95.5	–	–	–
	10K	27.6	27.6	**9.2**	407.7	409.3	814.2	824.0	–	–	–
avg-1	1K	17.5	17.5	6.0	8.5	8.7	21.6	31.4	3.2	**3.0**	–
	3K	49.5	49.5	**12.5**	41.3	42.1	85.8	95.5	21.3	20.7	–
	10K	414.7	414.5	**32.7**	407.7	410.2	814.2	824.0	210.7	208.7	–
avg-2	1K	17.5	15.4	**3.6**	8.6	8.7	21.6	47.4	–	–	–
	3K	49.5	20.2	**5.1**	41.3	42.1	85.8	239.5	–	–	–
	10K	414.7	27.6	**10.2**	407.7	410.2	814.2	2423.9	–	–	–

$\max_{r \in \text{rounds}} \left(\underset{C \in \text{clusters}}{\text{avg}} \; (\#\text{ of cache slots used by } C \text{ in round } r) \right)$ for avg-1 using $s = 256$ for all clusters (we used $s = 128$ for GeoLife due memory limitations). We use a larger cache size than our previous experiments so that fewer clusters hit the size limit, which gives us a more accurate analysis of cache usage. We found that the maximum average cache usage ranges from 6.7–84.5 slots. This explains why runtimes stop decreasing for cache sizes larger than 64 in Figure 4. If the cache is too large and many of the entries are empty, the runtime could be slower than using a smaller cache because when we merge the caches, we need to filter out the non-empty entries, and thus larger caches incur more overhead.

We use $s = 64$ for avg-1 in all other experiments to show that our framework can achieve good performance with a relatively small cache so that our memory overhead is minimal. Figure 4 shows that $s = 64$ gives good performance across different data sets. Section 6.4 shows our memory usage is very small with $s = 64$.

We now show that our running time is close to quadratic in practice. As described in Section 3, the work of our framework is bounded by $M = \sum |\mathcal{A}_i|(|\mathcal{Z}_i| + \log|\mathcal{A}_i|)$ plus the cost of distance computations, D. Figure 10 (left) shows that M and D are quadratic in the number of points in practice. Figure 10 (right) shows that only a very small fraction of clusters are included in the range queries and an even smaller fraction of distance computations between clusters are required in practice.

6.4 Memory Usage

Table 5 shows the memory usage (in megabytes) vs. data set size for 2D-GaussianDisc data sets for the different implementations. We measure the memory usage using the Valgrind Massif heap profiler tool [56]. For PC and PC-m, we use 48 cores with two-way hyper-threading. For fastprotein (fp) we use 36 threads, for Jeon (Je) we use 4 threads, and for Althaus (Al) we use 36 threads. The number of threads are chosen for best performance based on Figure 6.

Our algorithm PC shows linear memory increase vs. data size on all methods while most other methods, except Al, fc-gen for Ward, and fc-nnc for Ward and avg-2 (which require linear space), show quadratic memory increase vs. data size, which is consistent with the fact that they use a distance matrix. PC uses less memory than all other implementations, except that Al uses less memory for complete linkage, and Je uses less memory for the small 1K data set for avg-1. However, Al only works for complete linkage and is orders of magnitude slower than PC (Figure 6). PC uses less than 2x of the memory used by Al. Overall, PC uses up to 237.3x less memory than existing implementations.

7 RELATED WORK

There is a rich literature in designing HAC algorithms. In the most naive algorithm, a distance matrix is used to maintain all pairwise distances between clusters. On each iteration, the matrix is searched to find the closest pair of clusters, which are then merged, and distances to this newly merged cluster are computed. The algorithm runs for $n-1$ iterations, after which a single cluster remains. A straightforward implementation of this algorithm gives $O(n^3)$ time, but it can be improved to $O(n^2 \log n)$ time by storing matrix entries in heap-based priority queues [52, 58].

The two popular Python libraries scipy [65] and scikit-learn [59] both provide sequential algorithms for HAC. The two libraries' implementations both compute and store a distance matrix. Fastcluster [52] contains three implementations of HAC—two heap-based naive algorithms for general linkage functions, where one uses the distance matrix and the other compute cluster distances on the fly, and an NNC algorithm that uses the distance matrix. Lopez-Sastre et al. [44] propose a sequential NNC algorithm that speeds up the chain construction using a dynamic slicing strategy that only searches for the nearest neighbor within some slices. Their algorithm only works for linkage functions where the distance can be expressed using centroids and variances.

There have been implementations that focus on reducing the in-memory space usage of HAC from quadratic to linear by writing the quadratic-space distance matrix to disk, and loading it into memory in smaller chunks [43, 57]. These algorithms are sequential, and only merge one pair of clusters at a time. In contrast, our algorithm is parallel, and also does not require writing or loading additional information to and from disk. Moreover, the algorithms above are designed to take advantage of sparse distance metrics, where only some distances between data points are defined while other distances are considered to be "missing" and the points have "large" dissimilarity between them, making them less suitable for the Euclidean distance or squared Euclidean distance metrics.

There have also been many parallel algorithms developed for HAC, although it is difficult to parallelize in theory [27]. Olson [58] gives parallel algorithms, some of which parallelize the NNC algorithm by finding the nearest neighbor in parallel on each round, but still only merges one pair per round, and so there will always be $n-1$ rounds. Li [41] gives parallel HAC algorithms that store the distance matrix, based on an older theoretical model for a SIMD machine with distributed memory. Li and Fang [42] give parallel HAC algorithms on hypercube and butterfly network topologies. Du and Lin [23] give a parallel HAC algorithm on a cluster of compute nodes. Zhang et al. [70] propose a distributed algorithm for HAC that partitions the datasets using kd-trees or quadtrees, and then for each leaf node, finds a region where the R-NN pairs might exist. In parallel, each compute node finds the local R-NN pairs in a region, and then global R-NN pairs are found from the local pairs. This method merges multiple R-NN pairs, but their paper does not specify how the distances between clusters are updated or computed after merges. Fastprotein [31] is a naive parallelization of fastcluster. Sun et al. [63] develop a parallel version of algorithms that write the distance matrix to disk and load chunks of it into memory [43, 57]. However, they still only merge one pair of clusters at a time. Jeon and Yoon [33] present a parallel NNC algorithm using a distance matrix, which we discussed earlier. Althaus et al. [3] present a parallel complete linkage algorithm that uses linear main memory; however their algorithm requires $n-1$ rounds because they only merge the global R-NN pair on each round.

Besides the linkage criteria considered in this paper, other popular criteria for HAC include single, centroid, and median linkage. Single linkage with the Euclidean metric is closely related to the Euclidean minimum spanning tree problem, and can be solved efficiently using variants of minimum spanning tree algorithms [47, 66]. Centroid and median linkage do not satisfy the reducibility property and cannot take advantage of the NNC algorithm. There has also been work on other hierarchical clustering methods, such as partitioning hierarchical clustering algorithms and algorithms that combine agglomerative and partitioning methods [10, 18, 41, 48, 60]. Finally, there has been work on analyzing the cost function of the HAC problem [13, 16, 17, 21, 50] and approximating the HAC problem on various linkage criteria and metrics [1, 4, 11, 12, 20, 26, 36, 37, 49].

8 CONCLUSION

In this paper, we presented ParChain, a framework that supports fast and space-efficient parallel HAC algorithms based on the nearest-neighbor chain method. We introduced two key optimizations for efficiency, a range query optimization and a caching optimization. Using ParChain, we designed new parallel HAC algorithms for complete, average, and Ward linkage that outperform existing parallel implementations by 5.8–110.1x, while using up to 237.3x less space. It would be interesting future work to study how to improve the efficiency of ParChain by allowing approximation.

ACKNOWLEDGMENTS

This research was supported by DOE Early Career Award #DESC0018947, NSF CAREER Award #CCF-1845763, NSF Award #CCF-2103483, Google Faculty Research Award, Google Research Scholar Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] Amir Abboud, Vincent Cohen-Addad, and Hussein Houdrougé. 2019. Subquadratic high-dimensional hierarchical clustering. In *Conference on Neural Information Processing Systems*.

[2] Charu C. Aggarwal and Chandan K. Reddy (Eds.). 2014. *Data Clustering: Algorithms and Applications*. CRC Press.

[3] Ernst Althaus, Andreas Hildebrandt, and Anna Katharina Hildebrandt. 2014. A greedy algorithm for hierarchical complete linkage clustering. In *International Conference on Algorithms for Computational Biology*. 25–34.

[4] Mohammad Hossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, Mohammad Taghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. 2017. Affinity clustering: Hierarchical clustering at scale. In *Proceedings of the International Conference on Neural Information Processing Systems*. 6867–6877.

[5] Jean-Paul Benzécri. 1982. Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques. *Cahiers de l'analyse des données* 7, 2 (1982), 209–218.

[6] Pavel Berkhin. 2006. A survey of clustering data mining techniques. In *Grouping Multidimensional Data*. Springer, 25–71.

[7] R. K. Bock, A. Chilingarian, M. Gaug, F. Hakl, T. Hengstebeck, M. Jiřina, J. Klaschka, E. Kotrč, P. Savickỳ, S. Towers, A. Vaiciulis, and W. Wittek. 2004. Methods for multidimensional event classification: a case study using images from a Cherenkov gamma-ray telescope. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 516, 2-3 (2004), 511–528.

[8] Michel Bruynooghe. 1977. Méthodes nouvelles en classification automatique de données taxinomiques nombreuses. *Statistique et Analyse des données* 2, 3 (1977), 24–42.

[9] Paul B. Callahan. 1993. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *IEEE Symposium on Foundations of Computer Science*. 332–340.

[10] Rebecca J. Cathey, Eric C. Jensen, Steven M. Beitzel, Ophir Frieder, and David Grossman. 2007. Exploiting parallelism to support scalable hierarchical clustering. *Journal of the American Society for Information Science and Technology* 58, 8 (2007), 1207–1221.

[11] Evangelos Chatziafratis. 2020. *Hierarchical clustering with global objectives: Approximation algorithms and hardness results*. Stanford University.

[12] Michael Cochez and Hao Mou. 2015. Twister tries: Approximate hierarchical agglomerative clustering for average distance in linear time. In *Proceedings of the ACM SIGMOD international conference on Management of Data*. 505–517.

[13] Vincent Cohen-Addad, Varun Kanade, Frederik Mallmann-Trenn, and Claire Mathieu. 2019. Hierarchical clustering: Objective functions and algorithms. *J. ACM* 66, 4 (2019), 1–42.

[14] Ryan Curtin, William March, Parikshit Ram, David Anderson, Alexander Gray, and Charles Jr. 2013. Tree-Independent Dual-Tree Algorithms. *International Conference on Machine Learning* (04 2013).

[15] Ryan R. Curtin, Dongryeol Lee, William B. March, and Parikshit Ram. 2015. Plug-and-Play Dual-Tree Algorithm Runtime Analysis. *Journal of Machine Learning Research* 16, 101 (2015), 3269–3297.

[16] Sanjoy Dasgupta. 2016. A cost function for similarity-based hierarchical clustering. In *Proceedings of the ACM Symposium on Theory of Computing*. 118–127.

[17] Sanjoy Dasgupta and Philip M. Long. 2005. Performance guarantees for hierarchical clustering. *J. Comput. System Sci.* 70, 4 (2005), 555–569.

[18] Manoranjan Dash, Simona Petrutiu, and Peter Scheuermann. 2004. Efficient parallel hierarchical clustering. In *European Conference on Parallel Processing*. 363–371.

[19] C. De Rham. 1980. La classification hiérarchique ascendante selon la méthode des voisins réciproques. *Cahiers de l'analyse des données* 5, 2 (1980), 135–144.

[20] Daniel Defays. 1977. An efficient algorithm for a complete link method. *Comput. J.* 20, 4 (1977), 364–366.

[21] Laxman Dhulipala, David Eisenstat, Jakub Lacki, Vahab S. Mirrokni, and Jessica Shi. 2021. Hierarchical Agglomerative Graph Clustering in Nearly-Linear Time. In *Proceedings of the International Conference on Machine Learning*. 2676–2686.

[22] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 653–667.

[23] Zhihua Du and Feng Lin. 2005. A novel parallelization approach for hierarchical clustering. *Parallel Comput.* 31, 5 (2005), 523–527.

[24] Michael B. Eisen, Paul T. Spellman, Patrick O. Brown, and David Botstein. 1998. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences* 95, 25 (1998), 14863–14868.

[25] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.

[26] Sean Gilpin, Buyue Qian, and Ian Davidson. 2013. Efficient hierarchical clustering of large high dimensional datasets. In *Proceedings of the ACM International Conference on Information & Knowledge Management*. 1371–1380.

[27] Raymond Greenlaw and Sanpawat Kantabutra. 2008. On the parallel complexity of hierarchical clustering and CC-complete problems. *Complexity* 14, 2 (2008), 18–28.

[28] Ilan Gronau and Shlomo Moran. 2007. Optimal implementations of UPGMA and other common clustering algorithms. *Inform. Process. Lett.* 104, 6 (2007), 205–210.

[29] Dieter Heck, Johannes Knapp, Jean-Noel Capdevielle, Gerd Schatz, and T. Thouw. 1998. A Monte Carlo code to simulate extensive air showers. *Report FZKA* 6019 (1998).

[30] Ramón Huerta, Thiago Schiavo Mosqueiro, Jordi Fonollosa, Nikolai F. Rulkov, and Irene Rodríguez-Luján. 2016. Online Humidity and Temperature Decorrelation of Chemical Sensors for Continuous Monitoring. *Chemometrics and Intelligent Laboratory Systems* 157, 169–176.

[31] Ling-Hong Hung and Ram Samudrala. 2014. fast_protein_cluster: parallel and optimized clustering of large-scale protein modeling data. *Bioinformatics* 30, 12 (2014), 1774–1776.

[32] Anil Kumar Jain, Musti Narasimha Murty, and Patrick J. Flynn. 1999. Data Clustering: A Review. 31, 3 (Sept. 1999), 264–323.

[33] Yongkweon Jeon and Sungroh Yoon. 2014. Multi-threaded hierarchical clustering by parallel nearest-neighbor chaining. *IEEE Transactions on Parallel and Distributed Systems* 26, 9 (2014), 2534–2548.

[34] J. Juan. 1982. Programme de classification hiérarchique par l'algorithme de la recherche en chaîne des voisins réciproques. *Cahiers de l'analyse des données* 7, 2 (1982), 219–225.

[35] Eamonn J. Keogh and Michael J. Pazzani. 1999. An indexing scheme for fast similarity search in large time series databases. In *International Conference on Scientific and Statistical Database Management*. 56–67.

[36] Drago Krznaric and Christos Levcopoulos. 2002. Optimal algorithms for complete linkage clustering in d dimensions. *Theoretical Computer Science* 286, 1 (2002), 139–149.

[37] Meelis Kull and Jaak Vilo. 2008. Fast approximate hierarchical clustering using similarity heuristics. *BioData Mining* 1, 1 (2008), 1–14.

[38] Godfrey N. Lance and William Thomas Williams. 1967. A general theory of classificatory sorting strategies: 1. Hierarchical systems. *Comput. J.* 9, 4 (1967), 373–380.

[39] Bastian Leibe, Krystian Mikolajczyk, and Bernt Schiele. 2006. Efficient clustering and matching for object class recognition.. In *BMVC*. 789–798.

[40] Charles E. Leiserson. 2010. The Cilk++ concurrency platform. *The Journal of Supercomputing* 51, 3 (2010).

[41] Xiaobo Li. 1990. Parallel algorithms for hierarchical clustering and cluster validity. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12, 11 (1990), 1088–1092.

[42] Xiaobo Li and Zhixi Fang. 1989. Parallel clustering algorithms. *Parallel Comput.* 11, 3 (1989), 275–290.

[43] Yaniv Loewenstein, Elon Portugaly, Menachem Fromer, and Michal Linial. 2008. Efficient algorithms for accurate hierarchical clustering of huge datasets: tackling the entire protein space. *Bioinformatics* 24, 13 (2008), i41–i49.

[44] Roberto J. Lopez-Sastre, Daniel Onoro-Rubio, Pedroo Gil-Jimenez, and Saturnino Maldonado-Bascon. 2012. Fast Reciprocal Nearest Neighbors Clustering. *Signal Processing* 92 (2012), 270–275.

[45] Nil Mamano Grande. 2019. *New Applications of the Nearest-Neighbor Chain Algorithm*. Ph.D. Dissertation. UC Irvine.

[46] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[47] William B. March, Parikshit Ram, and Alexander G. Gray. 2010. Fast Euclidean minimum spanning tree: algorithm, analysis, and applications. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 603–612.

[48] Nicholas Monath, Avinava Dubey, Guru Guruganesh, Manzil Zaheer, Amr Ahmed, Andrew McCallum, Gokhan Mergen, Marc Najork, Mert Terzihan, Bryon Tjanaka, et al. 2020. Scalable Bottom-Up Hierarchical Clustering. *arXiv preprint arXiv:2010.11821* (2020).

[49] Nicholas Monath, Ari Kobren, Akshay Krishnamurthy, Michael R. Glass, and Andrew McCallum. 2019. Scalable hierarchical clustering with tree grafting. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1438–1448.

[50] Benjamin Moseley and Joshua R. Wang. 2017. Approximation bounds for hierarchical clustering: Average linkage, bisecting k-means, and local search. In *Proceedings of the International Conference on Neural Information Processing Systems*.

[51] Daniel Müllner. 2011. Modern hierarchical, agglomerative clustering algorithms. *arXiv preprint arXiv:1109.2378* (2011).

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

[53] Fionn Murtagh. 1983. A survey of recent advances in hierarchical clustering algorithms. *Comput. J.* 26, 4 (1983), 354–359.

[54] Fionn Murtagh and Pedro Contreras. 2017. Algorithms for hierarchical clustering: an overview, II. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 6 (2017), e1219.

[55] Fionn Murtagh and Pierre Legendre. 2011. Ward's hierarchical clustering method: clustering criterion and agglomerative algorithm. *arXiv preprint arXiv:1111.6285* (2011).

[56] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 89–100.

[57] Thuy-Diem Nguyen, Bertil Schmidt, and Chee-Keong Kwoh. 2014. SparseHC: a memory-efficient online hierarchical clustering algorithm. *Procedia Computer Science* 29 (2014), 8–19.

[58] Clark F. Olson. 1995. Parallel algorithms for hierarchical clustering. *Parallel Comput.* 21, 8 (1995), 1313–1325.

[59] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[60] Sanguthevar Rajasekaran. 2005. Efficient parallel hierarchical clustering algorithms. *IEEE Transactions on Parallel and Distributed Systems* 16, 6 (2005), 497–502.

[61] 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.

[62] Robert R. Sokal. 1958. A statistical method for evaluating systematic relationships. *Univ. Kansas, Sci. Bull.* 38 (1958), 1409–1438.

[63] Yijun Sun, Yunpeng Cai, Li Liu, Fahong Yu, Michael L. Farrell, William McKendree, and William Farmerie. 2009. ESPRIT: estimating species richness using large collections of 16S rRNA pyrosequences. *Nucleic Acids Research* 37, 10 (2009), e76–e76.

[64] Michele Tumminello, Fabrizio Lillo, and Rosario N Mantegna. 2010. Correlation, hierarchies, and networks in financial markets. *Journal of Economic Behavior & Organization* 75, 1 (2010), 40–58.

[65] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, Aditya Vijaykumar, Alessandro Pietro Bardelli, Alex Rothberg, Andreas Hilboll, Andreas Kloeckner, Anthony Scopatz, Antony Lee, Ariel Rokem, C. Nathan Woods, Chad Fulton, Charles Masson, Christian Häggström, Clark Fitzgerald, David A. Nicholson, David R. Hagen, Dmitrii V. Pasechnik, Emanuele Olivetti, Eric Martin, Eric Wieser, Fabrice Silva, Felix Lenders, Florian Wilhelm, G. Young, Gavin A. Price, Gert-Ludwig Ingold, Gregory E. Allen, Gregory R. Lee, Hervé Audren, Irvin Probst, Jörg P. Dietrich, Jacob Silterra, James T. Webber, Janko Slavič, Joel Nothman, Johannes Buchner, Johannes Kulick, Johannes L. Schönberger, JoséVinícius de Miranda Cardoso, Joscha Reimer, Joseph Harrington, Juan Luis Cano Rodríguez, Juan Nunez-Iglesias, Justin Kuczynski, Kevin Tritz, Martin Thoma, Matthew Newville, Matthias Kümmerer, Maximilian Bolingbroke, Michael Tartre, Mikhail Pak, Nathaniel J. Smith, Nikolai Nowaczyk, Nikolay Shebanov, Oleksandr Pavlyk, Per A. Brodtkorb, Perry Lee, Robert T. McGibbon, Roman Feldbauer, Sam Lewis, Sam Tygier, Scott Sievert, Sebastiano Vigna, Stefan Peterson, Surhud More, Tadeusz Pudlik, Takuya Oshima, Thomas J. Pingel, Thomas P. Robitaille, Thomas Spura, Thouis R. Jones, Tim Cera, Tim Leslie, Tiziano Zito, Tom Krauss, Utkarsh Upadhyay, Yaroslav O. Halchenko, Yoshiki Vázquez-Baeza, and SciPy 1.0 Contributors. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17, 3 (2020), 261–272.

[66] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. Fast Parallel Algorithms for Euclidean Minimum Spanning Tree and Hierarchical Spatial Clustering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1982–1995.

[67] Joe H. Ward Jr. 1963. Hierarchical grouping to optimize an objective function. *J. Amer. Statist. Assoc.* 58, 301 (1963), 236–244.

[68] Dongkuan Xu and Yingjie Tian. 2015. A Comprehensive Survey of Clustering Algorithms. *Annals of Data Science* 2, 2 (2015), 165–193.

[69] Shangdi Yu, Yiqiu Wang, Yan Gu, Laxman Dhulipala, and Julian Shun. 2021. ParChain: A Framework for Parallel Hierarchical Agglomerative Clustering using Nearest-Neighbor Chain. *arXiv preprint arXiv:2106.04727* (2021).

[70] Wei Zhang, Gongxuan Zhang, Xiaohui Chen, Yueqi Liu, Xiumin Zhou, and Junlong Zhou. 2019. DHC: A distributed hierarchical clustering algorithm for large datasets. *Journal of Circuits, Systems and Computers* 28, 04 (2019), 1950065.

[71] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning Transportation Mode from Raw GPS Data for Geographic Applications on the Web. In *International Conference on World Wide Web*. 247–256.