Parallel HDBSCAN* algorithm

1 Introduction

Clustering is a data mining technique that groups a set of objects into disjoint classes, called *clusters*, each containing similar objects. DBSCAN [Ester et al. 1996] is an algorithm that introduce the concept of density-based approach to clustering. In this approach, the clusters can be thought of as the areas of high density, while being separated by low density areas containing noise. Such an approach was a significant improvement over classical methods, e.g., k-means, in situations where the number of clusters is not known *a priori*, or where clusters may be of irregular shapes.

One of the significant drawbacks of the original DBSCAN algorithm is its inability to deal with clusters having variable densities. The OPTICS algorithm addresses part of it, constructing a dendrogram. But it still leaves up to a user to postprocess it. need better description of what OPTICS does

DENCLUE?

A new HDBSCAN* method, recently introduced in [Campello et al. 2013, 2015] approaches the problem how?. It was further improved in [McInnes and Healy 2017], showing that it is theoretically possible to run it in sub-quadratic $O(n \log n)$ time. However, its implementation [McInnes et al. 2017] was done in Python in serial, and was thus can only be considered to small to medium data sets. Running it for large data sets, such as those used in cosmology, is out of question.

In this paper, we propose a fully parallel algorithm for HDBSCAN*. Instead of using dual-trees as proposed in [McInnes and Healy 2017], we show that one only needs a single spatial index structure with slightly modified kNN computation. Our approach can be used with any spatial index. This is especially important for GPU, which imposes additional restrictions on the available data structures. While in our work we focused on bounding volume hierarchy algorithm for low-dimensional (spatial) data, the algorithm can be used with any dimensional data as long as the corresponding spatial index is available.

The remainder of the paper is organized as follows. Section 2 introduces the HDBSCAN* algorithm and the related work. Section 3 describes our proposed approach, with further implementation and optimization techniques detailed in Section 5. The algorithm's performance is demonstrated in Section 6. Finally, Section 7 concludes the paper and proposed future work directions.

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a nonexclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

2 Background

2.1 HDBSCAN* algorithm

For any point $x \in X$, the *core-distance* of x, denoted by $\kappa(x)$, is defined to be the distance to its k-th nearest neighbor (including x). Then, given points x and y, the *mutual reachibility distance* between x and y is defined as

(1)
$$d_{mreach}(x,y) = \begin{cases} \max\{\kappa(x), \kappa(y), d(x,y)\}, & \text{if } x \neq y, \\ 0, & \text{otherwise.} \end{cases}$$

2.2 Related work

3 Parallel HDBSCAN* algorithm

HDBSCAN* algorithm, as described in [Campello et al. 2015], consists three stages: computation of core-distances, construction of a minimum spanning tree (MST), and tree condensation. In this Section, we explore the parallel algorithms addressing each stage.

For now, we will use $k = minpts$ in our notation.

3.1 Parallel computation of core-distances

As defined in Section 2.1, a core-distance of a point is the distance to its k-th closest neighbor. Thus, it corresponds to a well-known k-nearest neighbors (kNN) search problem, with an additional post-search step of

sorting the found neighbors of each point based on their proximity, and selecting the furthest one.

A necessary step for an efficient kNN search is using a parallel spatial index. Many such indexes exist, and the choice often depends on the dimensionality of the problem, on the hardware used for parallelization, and on the data size.

It is important to note that this step places severe restrictions on a combination of n (size of the problem) and k. All kNN algorithms require storing distances to found candidates. This means that the memory storage requirement of the algorithm is at least $O(nk)$, and can easily exceed the size of the available GPU memory. For example, $n = 10,000,000$ with $k = 100$ requires at least 4 GB of memory, not including the data itself or the search index. This is a critical distinction from the DBSCAN algoritm where exist algorithms that only need $O(n)$ memory.

ArborX implementation

Current ArborX implementation uses bounding volume hierarchy (BVH) as a spatial index, as it demonstrated good performance for 3D data on GPU. The parallel computation of the core-distance algorithm should be done with a callback. Since we are only interested in a single distance and not in the neighbors themselves, storing the computed neighbors is not necessary.

The query should be constructed with an attached index of a point, and should provide access to the points storage. Inside the callback, one would have an index of the query and an index of a neighbor, allowing one to compute the distance between these two points[1]. Then, the max distance (a separate array) should be updated. As a side note, the list of nearest neighbors would include the point itself.

In ArborX, we are actually doing really badly in terms of memory usage, as we also store *indices*, which cuts memory availability by half. It means that we will be unable to run HACC problem for $k > 15$ or so.

3.2 Parallel construction of the MST for HDBSCAN*

The *minimum spanning tree* (MST) problem is a common graph optimization problem. It is stated as follows: given a connected undirected graph with real-valued weights assigned to each edge, find a spanning tree of the graph with a minimum total edge cost. Almost all algorithms to compute MST are a variation of the following generic algorithm. Starting from a forest of one-vertex trees, the generic algorithm connects the trees by choosing a suitable edge, maintaining an acyclic subgraph of the input graph. At any step, this subraph is a part of the final MST. At the end of the procedure, a single spanning tree is constructed.

The algorithms vary in their choice of the trees to connect. If only two trees can be connected at a given step without affecting the rest of the algorithm, this necessarily leads to a serial algorithm. For an algorithm to be suitable for parallel computations, multiple trees must be connected at the same time without affecting the overall result.

Borůvka's algorithm [Borůvka 1926] (and its variants explore literature on different variants of parallel Borůvka's algorithms) exhibit desired parallel characteristics. The vertices in each tree can be seen as disjoint sets component. Each such component defines a *cut*, a set of all edges with one vertex inside the component, and one outside. At each step, the algorithm selects the edge with the smallest weight among all edges of a cut of each component, and connects the trees that the vertices of that edge belong to. The algorithm converges in at most $\log_2(n)$ steps, as on each step each component is connected to at least one other.

One of the caveats of Borůvka's algorithm is that it is only correct if the edge weights are all distinct. Otherwise, cycles may form. If a graph has non-distinct edge weights, the edge comparison operation requires a consistent resolution. A simple approach is to then compare the edge vertices, e.g., their indices.

Borůvka's algorithm establishes the iterative structure of constructing MST. The challenge now is how to efficiently determine the edge with the smallest edge in d_{mreach} metric for each component. The problem can be reformulated as: given a set of points (of a component), find the closest point in the mutual reachability metric that does not belong to this component. In the [McInnes and Healy 2017] paper, the authors suggest using dual tree algorithms, as proposed in [March 2013; Curtin 2015] and other works. In this work, we instead use a single tree algorithm for two reasons. First, a parallel implementation of dual-tree algorithms, especially on accelerator, is an open research problem, with high-performance implementation posing a significant challenge. Second, it is possible to design a single tree implementation and avoid the complexity associated with dual-trees completely.

The posed problem can be reinterpreted as a kNN search problem but with a constraint of ignoring the nearest neighbors that are in the same component. It is, however, is complicated by two issues. First, the mutual reachability distance used in the search proce-

[1]The work of computing these distances is duplicated, as it is also performed during the hierarchy traversal. However, the cost is negligible, and should not be a cause of worry.

dure is different from the distance metric used for the index tree construction (e.g., Euclidian). Second, it is expected that for the components of large size, such as those at the later iterations of the Borůvka's algorithm, the number of nearest points to examine may be infeasible, and therefore, a fast procedure to avoid examining the points of the same components is required.

The first issue of using the mutual reachability distance for KNN search does not pose a significant challenge. kNN search algorithms usually maintain a priority queue of the nearest points, simultaneously maintaining an upper bound on the distance to the kth closest neighbor found so far. The purpose of this upper bound is to trim the traversal procedure by avoiding the nodes that are too far and have no chance to be included in the final result. If instead of an using the Euclidian distance one uses mutual reachibility distance for this upper bound, it can serve the same purpose. This is due to the fact that the reachibility distance between two points is not smaller than the Euclidean distance between them. Thus, if a distance from a point to a node exceeds the maximum of mutual reachibility distances of (at least) k points found so far, it is safe to ignore such a node.

The second issue can be solved in a manner proposed in [McInnes and Healy 2017]. Speicifically, one could associate a value with an internal node indicating whether the descendant points of that node belong to the same component, and if they are, which one. If a query point belongs to the same component, it may completely skip it as it would not contain any edges in the cut. These values will have to be updated after each step of the Borůvka's algorithm. For tree-based index this can be done in a bottom-up traversal algorithm.

Possible optimizations

Borůvka.

An interesting question can be posed: is it possible to avoid performing a kNN search for each iteration of the Borůvka's algorithm? It seems feasible that if, for example, each point of the component finds two closest edges leading to distinct components, then this information can be used to avoid the search on the next step, as the closest edge would already be known. However, the drawback of such approach is that there may be only one component nearby, with much further distance to others. An alternative approach would be for each point to find M closest neighbors. Once the components are merged, the points of the combined component are examined for whether there exists an edge in a new cut. The smallest edge of these could be the solution to the next step of the Borůvka's algorithm in certain scenarios (for example, if all points have an edge in the new cut, or if this minimum edge in the cut is shorter than the distance of each point in the new cut to its kth nearest neighbor). This is worth thinking more about, as avoiding kNN search can significantly speed up the algorithm

Minimum edge computation.

Another intriguing possibility is sharing a single upper bound among all threads of the same component. Since we are interested in the minimum edge for the whole component, we could ignore more nodes during the traversal if some points in the component are significantly close to another component. The drawback, of course, is that for large components this would result in the frequent access of the bound.

ArborX implementation

Borůvka's algorithm is external to the core search functionality of ArborX. Maintaning the indices of the components can be done through a single array of labels. Two components are merged by simply assigning the labels of one to the labels of the other. Borůvka's iterations are performed in loop with four parts: launching kNN search, choosing the minimum edge, merging components, updating the internal nodes metadata (component indices).

It is not currently possible to use kNN to find the minimum edge of the cut. The reason is that the process of finding nearest neighbors is automatic, and callbacks are being called only on the results of such search. Thus, it will not be able to find the closest neighbor from a different component if it is not in the specified number of the nearest neighbors. Thus, a modification of the kNN traversal algorithm is required.

The easiest way to hack our way through the traversal would be to copy the kNN traversal procedure and modify it to our needs. The modifications required are:

- Instead of using a pair of (index, distance) for the priority queue, we should use a pair of (index, mutual reachibility distance)
- If a point belongs to the same component, it should be skipped.

This is the bare minimum requirement to compute things correctly. An additional desired feature is to the ability to skip internal nodes based on the components its descendant indices belong to. This will require more implementation hacking.

As a side note, it may be worthwhile to investigate using `Kokkos::Graph` for the algorithm, as we have multiple kernel launches. For each Borůvka's iteration, we launch a) kNN, b) tree merge, c) tree data update. The

last two seem to be inexpensive, and may benefit from such an approach.

3.3 Tree condensation

In [McInnes and Healy 2017], the authors propose several steps to generated a condensed tree. The edges of the MST are processed in the increasing order to produce a single linkage tree through the use of a union-find data structure. This is then followed by a conversion of a single linkage tree into a condensed tree through a breadth-first traversal.

Several steps of the proposed algorithms are serial, e.g., processing the edges in increasing order or breadth-first traversal. While one could expect that the runtime of such algorithm may be small compared to that of the MST construction algorithm described in Section 3.2, it is possible to improve this approach.

When constructing MST, the following formation is readily avaiable: the components that are being merged, their sizes, and the weight of the edge connecting them. A component becomes a cluster that will be accepted as soon as its size exceeds the specified threshold. A *spurious* split corresponds to the situation where a component of sufficient size is merged with a components of an insufficient size. In [McInnes and Healy 2017] this is called as "falling out of the parent cluster". If both merged components contains sufficient number of points, this is a "true" split.

Following this logic, the value $\lambda_{max,c_i}(x_j)$, which is defined to the value λ at which the point x_j falls out of the cluster c_i (either as an individual point, or as a cluster split in a condensed tree) is precisely the value of λ at which the component containing x_j was merged into the component c_i. Similarly, the $\lambda_{min,c_i}(x_j)$ which is the minimum lambda value for which x_j is present in c_i is the closest larger value of an edge in MST.

While $\lambda_{max,c_i}(x_j)$ can be computed based fully on the available information of the components that are merged, the computation of the $\lambda_{min,c_i}(x_j)$ must be postponed until an ordering of all the edges of an MST may be achieved. The problem, however, then corresponds to a parallel upper bound algorithm.

One caveat of this approach is that if multiple components are merged together in a single iteration of the Borůvka's algorithm, the merging should be done in the order of increasing edges, requiring a (small) sort of the min edges of the merged components.

4 α-Tree

4.1 Notation

- $\mathcal{M}$: Mininum spanning tree
- $\mathcal{A}$: Alpha tree
- $\mathsf{P}_\alpha(v)$: Parent of v in $\mathcal{A}$

Algorithm 1: Dandrogram using α-tree

Input: α-tree : $\mathcal{A}$
Output: Dandrogram as Edge-tree

Function computeDendrogram(*some args*):
  /* Find branch σ for each edge e_i and store it in $\mathcal{B}$ */
  for $i \in 1, 2 \ldots, n_e$ **do**
    $\alpha_i \leftarrow \alpha-$insert$(e_i)$
    $\sigma_i \leftarrow 2\alpha_i$
    if $e_i \vdash \mathcal{A}[\alpha_i]$ **then**
      $\sigma_i = \sigma_i + 1$
    $\mathcal{B}[i] \leftarrow \binom{\sigma_i}{i}$
  Sort $\mathcal{B}$ by branch σ and index i
  for $i \in 2, 3 \ldots, n_e$ **do**
    $\binom{\sigma}{k} \leftarrow \mathcal{B}[i]$
    $\binom{\sigma'}{k'} \leftarrow \mathcal{B}[i-1]$
    /* k and k' are on same branch */
    if $\sigma = \sigma'$ **then**
      $\mathsf{P}(k) \leftarrow k'$
    /* k is start of a branch */
    else
      $k_\alpha \leftarrow \sigma/2$ // branch to α-edge map
      $\mathsf{P}(k) \leftarrow \mathcal{A}[k_\alpha]$
  return P

- $\mathcal{N}_\alpha(e)$: Edges in neighbourhood of e in the tree $\mathcal{A} \cup e$

Algorithm 2: Inserting an edge e_i in α-tree

Function $\alpha-\mathsf{insert}(\mathcal{A}, e_i)$**:**
  if $e_i \in \mathcal{A}$ **then**
    $j \leftarrow$ Index of e_i *in* $\mathcal{A}$
    return $\mathsf{P}_\alpha(j)$
  Find Neighbours of e_i in $\mathcal{A} \cup e_i$: $\mathcal{N}_\alpha(e_i)$
  Let

$$j \leftarrow \arg\max_k \mathcal{A}[k] \; | e_k \in \mathcal{N}_\alpha(e_i), \quad i < \mathcal{A}[j]$$

  /* Its a leaf edge */
  if $j = \emptyset$ **then**
    $j \leftarrow \arg\max_k k | e_k \in \mathcal{N}_\alpha(e_i)$
    return j
  /* k is in a chain $\mathcal{A}$, j */
  while $\mathcal{A}[j] > i$ **do**
    $j \leftarrow \mathsf{P}_\alpha(j)$
  return j

5 Implementation

6 Numerical results

7 Conclusion

Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

Borůvka, O. (1926). O jistém problému minimálním. *Práce Mor. Prırodved. Spol. v Brne (Acta Societ. Scienc. Natur. Moravicae)*, 3(3):37–58.

Campello, R. J. G. B., Moulavi, D., and Sander, J. (2013). Density-Based Clustering Based on Hierarchical Density Estimates. In Pei, J., Tseng, V. S., Cao, L., Motoda, H., and Xu, G., editors, *Advances in Knowledge Discovery and Data Mining*, Lecture Notes in Computer Science, pages 160–172, Berlin, Heidelberg. Springer.

Campello, R. J. G. B., Moulavi, D., Zimek, A., and Sander, J. (2015). Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection. *ACM Transactions on Knowledge Discovery from Data*, 10(1):5:1–5:51.

Curtin, R. R. (2015). *Improving dual-tree algorithms*. PhD thesis, Georgia Tech.

Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231, Portland, Oregon. AAAI Press.

March, W. B. (2013). *Multi-tree algorithms for computational statistics and phyiscs*. PhD thesis.

McInnes, L. and Healy, J. (2017). Accelerated Hierarchical Density Based Clustering. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 33–42. ISSN: 2375-9259.

McInnes, L., Healy, J., and Astels, S. (2017). hdbscan: Hierarchical density based clustering. *Journal of Open Source Software*, 2(11):205.