Appendix of HiClus: Highly Scalable Density-based Clustering with Heterogeneous Cloud

Chun-Chieh Chen\textsuperscript{a}, and Ming-Syan Chen\textsuperscript{b,c}
\textsuperscript{a}Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan
\textsuperscript{b}Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
\textsuperscript{c}Institute of Information Science, Academia Sinica, Taipei, Taiwan

1. Definition of DBSCAN

We describe the definitions of density-based clustering in this subsection. For more details of DBSCAN, please see [6].

\textbf{Definition 1.} (directly density-reachable): A point \( p \) is directly density-reachable from a point \( q \) wrt. \( \varepsilon \), \( \text{MinPts} \) if
1. \( p \in N_{\varepsilon}(q) \) and
2. \( |N_{\varepsilon}(q)| \geq \text{MinPts} \)

\( N_{\varepsilon}(q) \) represents the \( \varepsilon \)-neighborhood of the point \( q \) which includes all points whose distance to the point \( q \) is less than or equal to \( \varepsilon \). From the statement (2), it can be understood that there is a cluster if and only if there is a core point.

\textbf{Definition 2.} (density-reachable): A point \( p \) is density-reachable from a point \( q \) wrt. \( \varepsilon \), \( \text{MinPts} \) if there is a chain of points \( p_1, \ldots, p_n \), such that \( p_{i+1} \) is directly density-reachable from \( p_i \).

\textbf{Definition 3.} (density-connected): A point \( p \) is density-connected to a point \( q \) wrt. \( \varepsilon \), \( \text{MinPts} \) if there is a point \( o \) such that both \( p \) and \( q \) are density-reachable from \( o \).

\textbf{Definition 4.} (cluster): Let \( D \) be a set of points. A cluster \( C \) wrt. \( \varepsilon \) and \( \text{MinPts} \) is a non-empty subset of \( D \) satisfying the following conditions:
1. \( \forall p, q: \text{if } p \in C \text{ and } q \text{ is density-reachable from } p \text{ wrt. } \varepsilon \text{ and } \text{MinPts}, \text{ then } q \in C. \) (Maximality)
2. \( \forall p, q \in C: p \text{ is density-connected to } q \text{ wrt. } \varepsilon \text{ and } \text{MinPts}. \) (Connectivity)

\textbf{Definition 5.} (noise): A point \( p \) is not belong to definition (1) to (4).

2. Complexity Analysis and Proofs

2.1. Proof of HiClus Correctness

\textbf{Theorem 1.} Consider a data space \( S \). For any two partitions \( X \) and \( Y \) \( \in S \), the interval size of the overlapped area \( \geq \varepsilon \), where \( \varepsilon \) is a radius in DBSCAN. Let \( C_1 \) and \( C_2 \) be clusters residing in partitions \( X \) and \( Y \), respectively. If \( \exists p \in C_1 \text{ and } q \in C_2 \text{ are density reachable, where } p \text{ and } q \text{ are core points in clusters } C_1 \text{ and } C_2, \text{ respectively, and } p \neq q \text{ and } \exists \text{ a point } r \in C_3 \text{ in the overlap of partitions } X \text{ and } Y \), which is direct density reachable to points \( p \) and \( q \), then the interval size of the partition must be \( \geq \varepsilon \).

\textbf{Proof.} As shown in Figure 1 (a), we assume the size of overlap interval is \( \leq \varepsilon \). There are a point \( r \) which resides in an overlap area of partition \( X \), but \( p \) is in a non-overlap area of partition \( Y \). Since \( r \) is directly density-reachable to \( p \). The distance between \( p \) and \( r \) is less or equal to \( \varepsilon \). Based on this observation, point \( r \) only checked in one partition \( X \). We can see that \( d(p, r) \leq \varepsilon \) in partition \( X \) and \( d(q, r) = \infty \) in partition \( Y \), where \( d() \) is the distance of points. This contradicts the hypothesis that the distance between \( r \) and \( q \) is less or equal to \( \varepsilon \). So we reject our assumption and find that the interval size of the partition must be \( \geq \varepsilon \). We can ensure the correctness of HiClus. \( \square \)
2.2. Time Complexity

**Theorem 2.** The time complexity of HiClus is $O\left(\frac{n}{m} \log \frac{n}{m}\right) \ast \log m$.

**Proof.** Given $m$ machines in the cloud and the number of point in input dataset is $n$. Intuitively, the complexity of distributed partition phase is $\text{logm}\ast[O\left(\frac{n}{m}\right)+O\left(\frac{n}{m} \log \frac{n}{m}\right)]$, where $\text{logm}$ is the number of mapper rounds to split dataset into $m$ partitions. After data partition, the sum of the points in all partitions will increase to $n \ast (1 + p)$. Then, we apply distributed clustering on $m$ machines with complexity $O\left(\frac{n\ast(1+p)}{m}\right) + O\left(\frac{n\ast(1+p)}{m} \log \frac{n\ast(1+p)}{m}\right)$, which includes tree construction and clustering processing. Finally, we merge the sub-clusters in merging phase, the reducer can update the cluster identity in linear time. Heuristically, sub-cluster merging can be processed in $O(1)$, the complexity of merging is $O(n \ast (1 + p))$. Therefore, the worst case scenario of the HiClus complexity easily follows.

3. Cases of Merging Phase

In merging phase, we consider the effect of overlapping partition which incurs incorrect clustering results. Obviously, the clusters can easily merged in the case 1 and case 2 via the resident point. In case 3, the point is border point which belongs to two clusters. However, to merge two clusters need a common core point in both cluster $C_1$ and $C_2$. Thus, we do not perform merging in this case. In case 4, the noise point is removed in overlap analysis, we can skip merging process. We list the detailed rules for how we merge the sub-clusters in the following cases.

**Case 1:** Point $p$ is a core point which belongs to both cluster $C_1$ and $C_2$. Then we merge $C_2$ to $C_1$.

**Case 2:** Point $p$ is a core point in cluster $C_1$ and a border point in cluster $C_2$. It is valid to merge cluster $C_2$ with $C_1$.

**Case 3:** Point $p$ is a border point with different cluster identity $C_1$ and $C_2$ in different sub-trees. We cannot merge such clusters. This is because we do not promise that $C_1$ and $C_2$ have common core points.

**Case 4:** Point $p$ is a border point in cluster $C_1$ in a sub-tree and is noise in another sub-tree. We do not need to perform merging process.

4. Experiment Settings

We implement the HiClus algorithm on CUDA 6.5 with jCUDA library, Hadoop 1.2.1, and Spark 1.0 in the cloud environment. There are five machines, each machine is installed with a Nvidia GPU. One machine serves as master, and the other four machines are solely slaves. Here, we launch four virtual machines on each slave, each with CPU 2.4 GHz, 4 GB main memory. For running GPU computation, we execute the HiClus clustering on four machines, each with NVIDIA 9800 GTX. The 9800 GTX includes 32 SM (Streaming Multiprocessor) and each SM with eight MPs (Multiprocessor) in our GPU that comprises 256 stream processors. The processors with SM can access up to 16 KB shared memory and 512 GB global memory in a GPU.
5. The HiClus Algorithm

This section presents the details of our algorithm. First we define basic parameters for HiClus. Given input dataset $D$ with $Eps$ and $MinPts$. In practice, we need to input number of machines and GPU threads depending on our system settings. Algorithm 1 shows the overview of HiClus. First, the input dataset is divided into $m$ partitions in line 1. Then we assign mappers to construct sub-tree of DistKD-tree and to execute clustering via GPU in line 3 to 6. Finally, sub-results are gathered from machines by reducers, then merging is executed in line 7. The final results are written to HDFS and are presented as point and corresponding cluster identity.

**Algorithm 1:** The proposed HiClus algorithm.

<table>
<thead>
<tr>
<th>Input</th>
<th>1. The mappers split dataset $D$ into disjoint partitions $P_1, ..., P_m$, each into size $\frac{n}{m}$, where $m$ is the number of machines.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>$C$, the set of clustering results in $&lt;\text{point, cid}&gt;$, where $\text{point}$ is data and $\text{cid}$ is cluster identity;</td>
</tr>
<tr>
<td>1</td>
<td>2 for $i \in [1, m]$ do</td>
</tr>
<tr>
<td>2</td>
<td>;i access the cached partition $P_i$ and then constructs the sub-tree of DistKD-tree $T_i$ in local memory.</td>
</tr>
<tr>
<td>3</td>
<td>Mapper loads the sub-tree of $T_i$ to GPU shared memory and $\text{points}$ in partition $P_i$ to global memory of GPU.</td>
</tr>
<tr>
<td>4</td>
<td>$C_i \leftarrow \text{GPU_Clustering}(T_i, gThread, gBlocks, Eps, MinPts)$;</td>
</tr>
<tr>
<td>5</td>
<td>The mapper obtains immediate clusters $C_i \in P_i$ then copies the immediate clusters $C_i$ from GPU to the mapper with $&lt;\text{point, type, cid}&gt;$.</td>
</tr>
<tr>
<td>6</td>
<td>Reducer collects immediate results and performs merging of each cluster $C_i$ depending on the information of overlapped analysis.</td>
</tr>
<tr>
<td>7</td>
<td>return Outputs the merged clusters $C$ in $&lt;\text{point, cid}&gt;$, where $\text{cid}$ is cluster identity.</td>
</tr>
</tbody>
</table>

**Details of Distributed Partition**

Detailed pseudo-code of distributed partition is shown in Algorithm 2. $P_i$ is a partition which is stored in machine memory. In line 2 to 7, we adopt the KD-partition concept in distributed machines to find median value in each partition. Then split partitions into smaller ones. The distributed partition phase is a routine which, given number of machines $m$, decides if number of partitions is larger or equal than $m$.

**Details of Distributed Clustering**

Algorithm 3 shows details of our clustering process on GPU. Whenever a cached partition $P_i$ is read by machine $M_i$, it will be presented as sub-tree of DistKD-tree $T_i$ in line 1 to 4. We assume that total number of GPU threads is $g\text{Threads} \times g\text{Blocks}$. At line 6 to 13, we perform DBSCAN algorithm on GPU. The cached data is copied to GPU then we find the close neighbors of points on GPU, expand sub-clusters, and check that the processed points meet the definition of DBSCAN. Line 14 shows that a mapper outputs cluster results in key-value pairs.
Algorithm 2: The distributed partition phase in HiClus.

**Input:**
- data: input point data;
- Eps: the distance threshold;
- m: the number of machines;

**Output:**
- <key, values>: the output key is partition identity, pid and values are points;

1. **Mapper**(Key=line number, Value=data)
2. The mapper arbitrarily reads data to local memory. Such that there are m data partitions P1 to Pm assigned to different machines M1 to Mm.
3. \( d \leftarrow 1 \);
4. while \(|P| < m\) do
5.   for \( i \in [1, m] \) do
6.     The mapper, finds the median on d-dimension space and then outputs that value to a shared space.
7.     Depending on median value, the \( P_i \) in a mapper is split into two smaller partitions \( P_{2i} \) and \( P_{2i+1} \).
8.     Then mapper, i caches the immediate partitions \( P_{2i} \) and \( P_{2i+1} \) in local memory.
9. \( d \leftarrow d + 1 \)
10. Mappers emit the partition results to \(<\text{pid}, (\text{point}_1, ..., \text{point}_{n\cdot(1+p\%)}))>\) as output. // Bear in mind that the p is the ratio of points in overlapped area.
11. **Reduce**(Key=pid, Value=points)
12. Reducers gather the point data with the same pid;
13. while \((\text{points} \neq \text{null})\) do
14.   Each Reducer caches the assigned partition data in local memory;

**Details of Merging**

The merging process is shown in Algorithm 4. To decide if clusters \( C_1 \) and \( C_2 \) can be merged with point \( p \) is to check every point with case 1 to 4 in line 2 to 6. If the clusters are merged into one another, we reset the cluster identity. Finally, we write the final results to HDFS.
Algorithm 3: The details of distributed clustering phase via GPU

**Input:**
- $P_i$: the cached data partition with pid is $i$ in local memory;
- $Eps$: the distance threshold;
- $MinPts$: the minimum point number per cluster;
- $gThreads$: the number of threads per block on GPU;
- $gBlocks$: the number of blocks that we used on GPU;
- $P_i$: the $i$-th cached data partition in local memory of machine $i$;
- $T_i$: the index sub-tree of the DistKD-tree;
- $m$: the number of machines;

**Output:**
- $C_i$: the sub-results in form of $<point, (pid, type, cid)>$;

1. **Mapper** ($\text{Key} = \text{pid}$, $\text{Value} = \text{points}$)
2. Mapper at a host machine which accesses $<\text{pid}, \text{points}>$ in cached partition $P_i$, constructs sub-tree $T_i$ and initializes GPU shared memory.
3. if (size of ($T_i$) > shared memory size) then
   4. Mapper partition $T_i$ into smaller slices $T_{s1}, ..., T_{sgBlocks}$, where each slice is partial sub-tree.
   5. else
   6. $T_{s1} \leftarrow T_i$
7. The points in $P_i$ are copied to global memory of GPU.
8. _syncthreads();
9. while ($\exists T_{s_j}$ is not processed, where $j \in [1, gBlocks]$) do
10. neighbors $\leftarrow \text{Find Nearest Points} <<<< gBlocks, gThreads >>> (Eps, MinPts, P_i, T_{s_j})$
11. $C_i \leftarrow \text{ClusterExpand} <<<< 1, gThreads >>> (Eps, MinPts, P_i, neighbors)$
12. _syncthreads();
13. Mapper copies the clustering result $C_i$ from GPU to host memory.
14. Mapper emits $C_i$ in form of $<point, (pid, type, cid)>$;

Algorithm 4: The merging phase in HiClus.

**Input:** $<\text{key, values}>$: the immediate cluster results with $<\text{point, (pid, type, cid)>}$ as input;

**Output:** $C$: the final clustering results;

1. **Reduce** ($\text{Key} = \text{point}$, $\text{Value} = \text{pid, type, cid}$)
2. Master machine allocates a distributed mapping table $\text{Merge.Pool}$ which can store cluster information in distributed memory.
3. Reduce obtains the immediate cluster results.
4. $\text{Merge.Pool} \leftarrow (\text{point, type, cid})$
5. while ($\exists$ point not traversed) do
6. if (point $\in$ different pids) then
7. $\text{Overlap.Analysis(Merge.Pool, C)}$
8. Based on four cases that we show in Section 3.4, we decide to merge the clusters into one cluster or not.
9. if (two clusters need to be merged) then
10. Reducers change the cid in $\text{Merge.Pool}$.
11. $\text{Barrier.Synchronisation()}; // for multiple reducers$
12. $C \leftarrow \text{Merge.Pool}(\text{point, cid})$
13. Write cluster results $C$ to HDFS;