7-1-2015

Scalable Euclidean Embedding for Big Data

Zohreh S. Alavi
alavi.3@wright.edu

Sagar Sharma
Wright State University - Main Campus, sharma.74@wright.edu

Lu Zhou
Wright State University - Main Campus, zhou.34@wright.edu

Keke Chen
Wright State University - Main Campus, keke.chen@wright.edu

Scalable Euclidean Embedding for Big Data

Zohreh Alavi, Sagar Sharma, Lu Zhou, Keke Chen
Data Intensive Analysis and Computing (DIAC) Lab, Kno.e.sis Center
Department of Computer Science and Engineering
Wright State University, Dayton, OH
{*alavi.3, sharma.74, zhou.34, keke.chen*}*@wright.edu*

***Abstract*—Euclidean embedding algorithms transform data defined in an arbitrary metric space to the Euclidean space, which is critical to many visualization techniques. At big-data scale, these algorithms need to be scalable to massive data-parallel infrastructures. Designing such scalable algorithms and understanding the factors affecting the algorithms are important research problems for visually analyzing big data. We propose a framework that extends the existing Euclidean embedding algorithms to scalable ones. Specifically, it decomposes an existing algorithm into naturally parallel components and non-parallelizable components. Then, data parallel implementations such as MapReduce and data reduction techniques are applied to the two categories of components, respectively. We show that this can be possibly done for a collection of embedding algorithms. Extensive experiments are conducted to understand the important factors in these scalable algorithms: scalability, time cost, and the effect of data reduction to result quality. The result on sample algorithms: FastMap-MR and LMDS-MR shows that with the proposed approach the derived algorithms can preserve result quality well, while achieving desirable scalability.**

I. Introduction.

Thanks to a wide range of high-throughput sensors and the advancement in communication technology, there has been an explosive growth in the data generated from online applications, cyber-physical applications, and simulations. It is well perceived that the value of big data lies in the understanding of the data, for which many have believed that visual analytics is one of the most effective approaches [10]. Visual analytics is particularly useful at the beginning stage of data analysis when analysts have few clues about the data. Combined with automated algorithms, visual exploration can help users filter data, organize data, capture unique patterns, and understand patterns in big data.

However, the scale of data adds several unique challenges to visual exploratory data analysis, one of which is Euclidean embedding [5] that maps the data defined with a certain non-Euclidean metric to the Euclidean space while preserving the relative distance values defined with the original metric. This step is necessary for visualization as human visual perception is more sensitive to Euclidean distance. For example, we can visually detect dense point clouds and understand how they form clusters, where similar objects are placed closer to each other in the Euclidean space than non-similar ones. Well-known examples of non-Euclidean metrics include angular distance derived from cosine similarity for information retrieval, and Hamming distance (and edit distance) for strings or DNA sequence data. Note that Euclidean embedding should not be misunderstood as dimensionality reduction [1], although it can be used for dimensionality reduction if the original metric space is also Euclidean.

Most existing techniques for Euclidean embedding (e.g., multidimensional scaling (MDS) [5]) were designed without big data in mind, often in non-linear time (e.g., $O(N^3)$) and space (e.g., $O(N^2)$) complexity for single-machine processing. However, at the big data scale, scalable algorithms have to be exploited to utilize massive data-parallel infrastructures such as Hadoop/MapReduce [7]. Meanwhile, when the new algorithms are capable to handle data at scale, it is also critical to preserve the quality of embedding. To our knowledge, Euclidean embedding at the big data scale has not been fully studied yet.

Scope of Research and Our Contributions. Our strategy is to combine data reduction and parallel processing techniques to extend existing algorithms to scalable ones. Specifically, this approach examines an existing embedding algorithm to identify the naturally parallel components and non-parallelizable components. The naturally parallel components can be cast to any existing parallel processing platform such as MapReduce [7], while we apply the data reduction techniques (e.g., sampling) to handle the non-parallelizable components. In studying this approach, we try to answer several questions. (1) What kind of algorithms can be efficiently transformed with the proposed approach? (2) While the scalability is the major target, how is the quality of embedding preserved, especially when data reduction is applied?

Our study has several major contributions.

- We use a divide-and-conquer approach to decomposing Euclidean embedding algorithms and scaling up them with a combination of data reduction and parallel processing techniques.
- We show that it can be effectively applied to several algorithms such as FastMap [8] and Landmark MDS [6]. The scalable solutions can be conveniently implemented with popular platforms such as MapReduce.
- We have thoroughly studied two intricately related

factors: scalability and result quality with experiments on real datasets.

The whole paper is organized as follows. Section II gives the notations and definitions of the problem. Section III presents the approach to decomposing an embedding algorithm and extending it with data reduction and parallel processing techniques. This approach is instantiated with the FastMap-MR and LMDS-MR algorithms. Section IV studies the scalability and the effect of data reduction on the sample algorithms with extensive experiments.

II. Preliminary.

A. notations and defintions

We assume that a dataset contains N objects, denoted as the set V. With a given metric definition (Euclidean or non-Euclidean) on the N objects, the task is to find another set of N points in the Euclidean space, $V'_{k\times N}$, whose pairwise Euclidean distances approximately match the distances defined by the original metric. This problem is called multidimensional scaling (MDS) or Euclidean embedding. Note that V may or may not be a multidimensional-vector set. If the original metric is Euclidean distance on multidimensional space, V consists of a set of multidimensional vectors and Euclidean embedding can be also used to reduce the dimensionality of the dataset V. As dimensionality reduction is another big topic itself, our study applies mainly to non-Euclidean original metrics.

B. classical MDS

In this section, we briefly review classical MDS(cMDS) and some other related approaches. Algorithm 1 gives the details of the cMDS algorithm, which is a two-stage procedure [2]. Assume the pairwise distance matrix D from the N objects in the original metric space is given. The first stage converts the matrix D to the Gram matrix B, which is the multiplication of some matrix $W_{k\times N}$ and its transpose, i.e., $B = W^TW$. The idea is simple: once the matrix B is found, the columns of W are the vectors in the Euclidean space, whose pairwise Euclidean distances reconstructs the D matrix. It is easy to find W from B by using eigen-decomposition. The transformation $B = -\frac{1}{2}HDH$ is called double-centering, which transforms the distance matrix to a Gram matrix. This original MDS algorithm is not appropriate for large scale applications because they require the entire $N \times N$ distance matrix to be stored in memory and has $O(N^2)$ complexity in generating B and $O(N^3)$ in conducting eigen-decomposition.

Algorithm 1 cMDS

Input: $D, k(k < N)$
Set $B = -\frac{1}{2}HDH$, where $H = I - \frac{1}{n}11^T$ is the centering matrix
Compute the decomposition of $B : B = U\Lambda U^T$
Form Λ_+ by keeping the top k eigenvalues of Λ and setting other eigenvalues to 0.
Return $X = \Lambda_+^{1/2}U^T$

Several approximations to classical MDS have been proposed for larger scale data, such as FastMap [8] and Landmark MDS (LMDS) [6]. Since these algorithms were designed for "large data" defined more than a decade ago, it remains unclear whether they can be cast to massive-scale parallel processing infrastructures and whether the result accuracy can be preserved at the big-data scale.

C. Massive data parallel processing

Since the size of big data is evolving, it demands that the corresponding algorithms effectively utilize massively parallel processing infrastructures to adapt to the future growth of data scale. Recently, a few approaches have been developed for parallel processing of very large datasets with many shared-nothing commodity servers, such as Hadoop/MapReduce (hadoop.apache.org), and Spark (spark.apache.org). These approaches allow developers to develop efficient parallel and distributed applications that can easily scale up to many machines.

In this paper, for convenience we will use MapReduce to formulate the parallelized algorithms. However, any parallel processing platform can be applied to implement the derived scalable algorithms. A MapReduce program has two major functions: Map and Reduce. Map is used for naturally parallel tasks, which in our context applies to every record in the dataset. Reduce functions are often aggregate functions summarizing the Map functions' outputs.

III. Scalable Euclidean Embedding for Big Data

In this section we propose a simple approach to decompose and extend an existing Euclidean embedding algorithm to generate a scalable version. We choose FastMap and Landmark MDS to show how we can apply this approach to convert an existing Euclidean embedding algorithm to a big-data-ready one. The discussion will also be extended to the major classes of existing Euclidean embedding algorithms to show the broader applicability of this approach.

A. A Framework Extending Embedding Algorithms for Big Data.

A traditional method to develop parallel matrix-based algorithms for the high-performance computing (HPC) environment is to replace the matrix operations, such as matrix addition and multiplication, with the parallel versions. However, big data raises special challenges in parallel processing. Specifically, it's impractical to have a dense matrix of $O(N^2)$ physically stored and processed with parallel matrix operations that still have overall computation and node-node communication complexity non-linear to N.

In this study, we employ a simpler decomposition approach to transform the existing algorithms for big data. The basic idea is to decompose the algorithm into two

types of components: the naturally parallel ones and those that data reduction applies, which will be easy to redesign and implement. Specifically, we will identify the naturally parallel (also known as *embarrassingly parallel*) components in the algorithms that apply to *each record* of the big dataset. These components can be easily implemented with the popular data-parallel processing platforms such as MapReduce. Second, for those operations that approximate results on reduced data are also acceptable, we will investigate whether the data reduction approaches (e.g., sampling) can apply. The advantage of this approach is its simplicity. However, there are several problems to be further studied: (1) how to design the components for parallel processing and data reduction, respectively, after we finish the decomposition, and (2) how the data reduction step affects the quality of final results.

Note that this approach may not work in a few situations: there is no naturally parallel components in the program, naturally parallel components are expensive to implement, or data reduction cannot guarantee the quality of the non-linear complexity components. However, in our initial study on the embedding problem, we have identified that several algorithms can possibly use this approach, and we would like to understand the key factors: the scalability, the time complexity, and the effect of data reduction to the result quality, for the extended algorithms. In the following, we show how to transform two representative algorithms: FastMap and Landmark MDS. The key factors will be evaluated and understood with experiments.

B. Scaling up FastMap for big data

The FastMap algorithm is an iterative process. In each iteration, it tries to find an approximate eigenvector, maps the whole data to that eigenvector to get one dimension of values in the transformed space, and computes the "residue" of distance fitting.

The approximate eigenvector is defined with two furthest points with the distance relationship in the original metric space, named "pivot points", assuming that the actual eigenvector is more likely to be on the line connecting the pivot points. To reduce the complexity of find the furthest pairs, it employs a simple heuristic to find a pair of approximate furthest points: for any randomly selected object O_r, find its furthest point O_a and then find the O_a's furthest point O_b. O_a and O_b are the pivot points.

At the second step, it iteratively finds one-dimensional embedding by mapping all points to this approximate eigenvector. Assume the original metric is defined by a distance function $d_{i,j} = dist(O_i, O_j)$, where O_i and O_j represents any two objects. Let the transformed k-dimensional record be $Y_i = (y_{i1}, \ldots, y_{ik})$. The mapping to the first eigenvector is done with the following formula based on the cosine mapping law.

$$y_{i1} = \frac{d_{i,a}^2 + d_{a,b}^2 - d_{i,b}^2}{2d_{a,b}} \tag{1}$$

where y_{i1} is the first coordinate of object O_i, $d_{i,a}$ and $d_{i,b}$ are the distances between object O_i and pivot points O_a and O_b, respectively, and $d_{a,b}$ is the distance between the two pivot points. Finally, the residue of distance fitting for any pair of points O_u and O_v is computed with the following formula for j-th iteration

$$d_{u,v}^2 \leftarrow d_{u,v}^2 - \sum_{i=1}^{j}(y_{u,i} - y_{v,i})^2, \tag{2}$$

which updates $d_{u,v}^2$ for the next iteration. The iterative procedure continues until the desired number of dimensions, k, is reached.

1) Decompose and Design: Each iteration of the algorithm consists of three major steps: "finding the approximate eigenvector" $\rightarrow$ "mapping to the eigenvector" $\rightarrow$ "computing distance residues". Since the later two steps (i.e., the formulas 2 and 1) are individually applied to each record, they can be clearly cast to naturally parallel components. However, it is still not trivial to design the parallel components to efficiently generate the dimensional values and computing the residual distances, which demands a careful design of data structures and algorithms that will be described in detail.

The first step, finding approximate eigenvector, has two alternative scalable solutions: either finding the pair of pivot points in a parallel algorithm, or use data reduction methods to find the pivot points in reduced datasets. The design of parallel algorithm will be described in more detail. Here, we briefly justify the data reduction method. The original algorithm for finding the pivot points is already an approximation algorithm because it is prohibitively expensive to find the exact results. Thus, further relaxing the approximation level with data reduction might be still acceptable. The use of data reduction is quite straightforward - applying the original pivot point algorithm on a random sample set. This can dramatically reduce the computational cost - in a sense that one machine can be sufficient to hold the sample set and find the pivot points.

In the following, we will mainly describe the design of data structure suitable for data-parallel processing and cast the parallel algorithms to the MapReduce model.

2) FastMap-MR: All the three steps of FastMap can be cast to the MapReduce framework, although the first step can be replaced with the algorithm based on data reduction. First of all, to maximize the locality in data-parallel processing, the data structure needs to be carefully designed so that the major steps can be conducted locally with all required information available locally. For simplicity, we assume the original dataset is stored in a file, where each line corresponds to one data object. Note that all the steps require the distance residues to be used as the current distance in computation. However, updating all distances will be impractical as it will cost $O(N^2)$ in both time and storage.

Since not all of these distances are used, the key idea here is to avoid the computation of distance residue for each pair of points, and only compute it on demand, in a "lazy" way. Our storage design is to have the generated new dimensions appended to the same line of the record, so that when needed the distance residue can be computed on the fly.

For simplicity, we assume the objects in the original space are also represented as vectors of m dimensions, with a known distance function $d(i, j)$ defined on the vectors. Let x_{ij} be the j-th dimensional original value of the i-th object and y_{ik} be the k-th dimensional value in the mapped Euclidean space. Each line in step k of the algorithm (before getting y_{ik}) will look like

$$i : x_{i1}, \ldots, x_{im} | y_{i1}, \ldots, y_{i,k-1}, i = 1..N \qquad (3)$$

With this organization, we can conveniently cast the original algorithms to MapReduce programs.

Finding Pivot Points with MapReduce. The algorithm to find the two approximately furthest points consists of two steps. First, for a randomly selected point in the dataset, a point furthest to this point is found, denoted as O_a. This step simply computes the distances to the randomly selected point and find the maximum. Next, the furthest point to O_a, denoted as O_b, is found with the same procedure. The core operation is computing the distance between a selected point and each of the remaining ones in the dataset. Note that this distance should be the "residual distance", the efficient computation of which is enabled by the data structure. The residual distance is only computed when it is needed.

Algorithm 2 formulates the distance computation and comparison in one MapReduce program. Specifically, this algorithm takes the dataset file and a fixed point U from the dataset as input parameters, and output the point V that is the furthest to U. The Map function calculates the distance,d_i , between each object i (i.e., the line L_i in the input file) and input parameter U, and emits a key-value pair of $\langle 1, L_i : d_i \rangle$, where 1 is used as the key as we want all Map outputs going to the same combiner or reducer. Before a Map output going to the Reduce, the Combine function will elect the local furthest point, which is then sent to the Reduce function. Clearly this local reduction is valid as finding the maximum distance is a normal aggregate procedure. The Combine function is essentially the same as the Reduce function.

Mapping All Records to the Euclidean space with MapReduce In each iteration, this step takes the dataset file and two pivot points as input and finds one embedding coordinate. In iteration j, the Map function finds the y_{ij} for the point X_i, and output a key-value pair $\langle i, x_{i1} \ldots x_{im} : y_{i1} \ldots y_{ij} \rangle$. As no further computation is needed, the Reduce stage is ignored. In the end of each iteration, the algorithm will generate an updated dataset in the same format as Eq. 3 shows, which will be used as the input to the next iteration.

As described earlier, the step of computing distance residues is done on the fly in the two MapReduce algorithms when it is needed. There is no individual step to update all distance residues.

Algorithm 2 FastMap-MR: finding the furthest point to a given point.

Input: dataset D, each line L_i in the form of i : $x_{i1}, \ldots, x_{im} | y_{i1}, \ldots, y_{i,k-1}, i = 1..N$
a given point $U = (u_1, \ldots, u_m)$
Output: vector $V = (v_1, \ldots, v_m)$

`Map` (L_i)
▷ key is the line number, i and value is the content of the line
begin
  $X_i, Y_i \leftarrow$ parse L_i
  $d_i \leftarrow \sqrt{d_{i,u}^2 - \sum (y_{ij} - y_{uj})^2}$
  emit $\langle 1, L_i : d_i \rangle$
end

`Combine/Reduce` (key k, value-set $\{L_i : d_i\}$)
begin
  $V \leftarrow \emptyset$
  $maxD \leftarrow 0$
  for each $L_i : d_i$ **do**
    if $d_i > maxD$ **then**
      $maxD \leftarrow d_i$ $V \leftarrow L_i$
    end if
  end for
  emit $\langle 1, V : maxD \rangle$
end

Algorithm 3 FastMap-MR: generating one dimension in Euclidean space

Input: dataset D, each line L_i in the form of i : $x_{i1}, \ldots, x_{im} | y_{i1}, \ldots, y_{i,k-1}, i = 1..N$
vector $O_a = (a_1, \ldots, a_m)$ vector $O_b = (b_1, \ldots, b_m)$
Output: dataset D', each line L_i' in the form of $x_{i1} \ldots x_{im}$: $y_{i,1} \ldots y_{i,j}, y_{i,j+1}$

`Map (key k, value v);`
▷ key is the line number, i, and value is the content of the line
begin
  $y_i = \dfrac{d_{i,a}^2 + d_{a,b}^2 - d_{i,b}^2}{2d_{a,b}}$
  emit $\langle i, append(L_i, y_i) \rangle$
end

C. Scaling up LMDS for big data.

The bottleneck for applying the classical MDS to big data is the expensive matrix computation, where finding the eigenvectors is the core operation. FastMap uses a pair of furthest points to compute an eigenvector, while Landmark MDS uses a sample set to compute all approximate eigenvectors in one step. These sample points are called "landmarks". Specifically, the LMDS algorithm is divided into two stages. At the first stage, the classical MDS is applied on landmarks to transform the sample points to the Euclidean space, the eigenvectors of which can be obtained afterwards. This step has used data reduction and can be possibly done with a single machine. At the second stage,

the remaining data points are embedded using a linear transformation based on the eigenvalues of the Gram matrix found by the embedding procedure. We consider D_L as the squared-distance matrix for landmark points, and after applying cMDS we get the matrix $M = U\Lambda^{-1/2}$, where U is the eigenvectors and Λ is the diagonal eigenvalue matrix. Let μ be the column mean of D_L, and d_i be a vector having the squared distances between the point O_i and the n landmark points. The embedding vector of O_i is related linearly to d_i by the formula:

$$y_i = \frac{1}{2}M^{\#}(\mu - d_i), \tag{4}$$

where $M^{\#}$ is the pseudoinverse transpose of M defined as: $M^{\#} = \Lambda^{-1/2}U^T$.

Clearly, once the eigenvectors are obtained, $M^{\#}$ can be computed and passed to all records. The follow-up operation (Eq. 4) is applied to *each* record. Since the core algorithm has already used data reduction and the follow-up operation is naturally parallel, casting the algorithm to our approach is straightforward.

1) LMDS-MR: As the first stage is already done with data reduction, we only need to design the parallel algorithm for the second stage. Algorithm 4 uses the landmark points, the precomputed μ and $M^{\#}$ as the input to process the dataset file. The Map function computes the embedding vector in $\mathbb{R}^k$ for each object and emits the pairs $\langle i, (y_{i1}, y_{i2}, ..., y_{ik})\rangle$ for each line.

Algorithm 4 LMDS-MR: embed all objects into $\mathbb{R}^k$

Input: dataset D, each line L_i in the form of i : $x_{i1}, \ldots, x_{im}|y_{i1}, \ldots, y_{i,k-1}, i = 1..N$; the *Landmark* set, each line of which contains coordinates of one landmark; matrix $M^{\#}$; and the vector μ
Output: dataset D', each line of which contains y_i.

```
Map (L_i)
begin
  O_i ← parse L_i
  d_i ← compute distances from O_i to Landmark points
  y_i = 1/2 M#(μ − d_i)
  emit ⟨i, y_i⟩
end
```

D. Extending Other Methods with the Proposed Approach.

In this section, we briefly examine several other Euclidean embedding algorithms to see whether they can be revised to scalable versions, based on our proposed approach.

MetricMap [12] has a similar 2-step structure to LMDS. First, MetricMap uses a sample of dataset (i.e., the reference objects) to build the target space. Then, it maps each object in the dataset to a point in the target space by comparing the object with the reference objects. The first step uses data reduction; the second step applies to each point and thus is naturally parallel.

Another set of MDS algorithms are based on the spring models [3], [13], which are well adopted by the information visualization community. The basic spring model iteratively calculates a low-dimensional displacement vector for each point to minimize the difference between the low-dimensional and high-dimensional distances. Since every iteration requires each point to be computed with all other points in the dataset, the iteration complexity is $O(N^2)$, prohibitively expensive for big data. There are two possible approaches can be applied to address the scalability. Chalmers [3] uses a fixed set of neighboring points and a randomly sampled set for each point as the tuning points. This can be possibly cast to a random *shuffling* step in MapReduce. Williams et al. [13] improves the algorithm further with dataset partitioning.

The MDS problem is also modeled with the semi-definite programming (SDP) for some special settings such as non-metric or noisy distance computation in the original space [2], [11]. Typical SDP solvers are very expensive, with a $O(N^3)$ cost, not scaling to big datasets. Cayton et al. [2] has mentioned using the Nyström method to reduce the complexity of each SDP iteration. However, there is no efficient study on naturally-parallel solutions for SDP yet.

Table I summarizes the application of the proposed method to the major Euclidean embedding algorithms.

Table I
DECOMPOSE AND SCALE UP OTHER EMBEDDING METHODS.

Methods	Data reduction	Data parallel
FastMap[8], LMDS[6], MetricMap[12]	sampling at the first step	second step is data parallel. Discussed in this paper.
Spring methods [3], [13]	record sampling [3] and partitioning [13]	To be explored.
SDP-based methods [2], [11]	matrix sampling	To be explored.

IV. EXPERIMENTS

In previous section we showed how to apply the proposed approach to scale up existing Euclidean embedding algorithms, with FastMap and Landmark MDS as the detailed examples. In this section, we conduct extensive experimental evaluation on the derived sample algorithms Fastmap-MR and Landmark-MR to understand scalability and the impact of data reduction to the embedding quality. Specifically, there are three goals of this study. First, we want to see how our approach scales up with computing resources. Second, we want to see how the data reduction step affects the quality of final results. Finally, we compare these methods to see which of the scalable solutions provides better performance in both quality and computational cost.

A. Experiment setup

System setup. The algorithms are implemented and tested with an in-house Hadoop cluster. This Hadoop cluster has

16 nodes: 15 worker nodes and 1 master node. The master node also serves as the application server. Each node has two quad-core AMD CPUs, 16 GB memory, and two 500GB hard drives. These nodes are connected with a gigabit Ethernet switch. Each worker node is configured with eight map slots and six reduce slots, approximately one map slot and one reduce slot per core as recommended in the literature.

Datasets. To evaluate the ability of processing large datasets, we extend two existing large scale datasets to larger scales for experiments. The two extended datasets are (1) the Census 1990 data (http://goo.gl/AGkszE) with 68 attributes and (2) The Buzz in tweets (http://goo.gl/jPLto3) with 77 attributes. Both can use Euclidean distances for clustering as shown in previous studies. The following data extension method is used to preserve the clustering structure for any extension size. The dataset is first normalized for easier handling. Then, for a randomly selected record from the normalized dataset, we add a random noise (e.g., with normal distribution $N(0, 0.01)$) to each dimensional value to generate a new record and this process repeats for sufficient times to get the desired number of records. In this way the basic clustering structure is preserved in the extended datasets. With this method, the Census data is extended to 10 GB, and the Buzz data to 8.5 GB.

Evaluation Measures. We use two measures to evaluate the embedding result, the stress function [8] and clustering accuracy. The stress function is defined as follows:

$$stress = \sqrt{\frac{\sum_{i,j}(d'_{ij} - d_{ij})^2}{\sum_{i,j} d_{ij}^2}} \quad (5)$$

where d_{ij} is the dissimilarity measure between objects O_i and O_j, and d'_{ij} is the Euclidean distance between their images O'_i and O'_j. In the experiments, we also use Euclidean distance as the original metric. The stress function gives the relative error in distance transformation for each pair of objects. Because of its high complexity, it is impractical to compute the stress for the entire big dataset. In the following we will use a random sample set to compute the stress function.

Clustering accuracy represents the high-level utility of the transformed data. As clustering is a major method for data analytics, it is important to understand how well Euclidean embedding preserves clustering structures. We adopt the classical purity definition as the clustering accuracy that is based on confusion matrix. Let the element M_{ij} of the confusion matrix be the number of items in the cluster C_i of the first clustering labeled as C'_j of the second clustering. The purity measure is defined on the maximum consensus between the two clustering results with the confusion matrix. To construct the confusion matrix we need to know the standard clustering result. The Buzz dataset has the class labels, which are often used for approximate clustering evaluation. The Census dataset does not have the standard clustering result. However, our previous studies [4] show that the Census dataset contains three major clusters, which can be clearly visualized with the VISTA tool [4] (Figure 1 shows about 12,000 sample records). We define labels based on the visual clustering results. Again, due to the limitation of visual labeling system, we use a random sample set to evaluate the overall clustering accuracy.

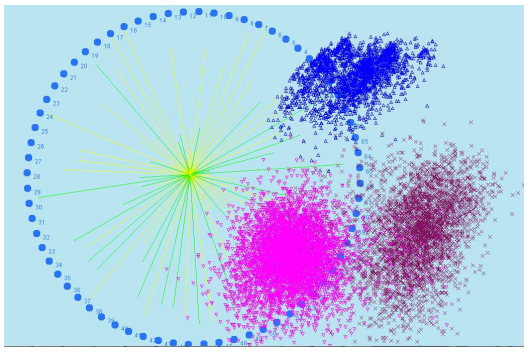

Figure 1. Visualization of Census data with the VISTA system.

B. Results

1) Scalability: We first show how the new methods scale with computing resources. Figure 2 shows the speedup for the two methods, FastMap-MR and LMDS-MR according to the available computing resources. The mapping result uses 3 dimensions in the Euclidean space (k=3) and sample rate = 0.00001 for the Census dataset. The x-axis is the number of map slots of the Hadoop system, which represents the available resources. This number is controlled by using the Hadoop's fair scheduler. The y-axis shows the speedup, which is defined by the following formula: $S = \frac{T_{base}}{T_{new}}$, T_{base} is the base execution time. Here we use the running time on 5 map slots as the base execution time. T_{new} is the new execution time with more computing resources. Linear speedup or ideal speedup is obtained when $S = p$, where p is the number of map slots divided by 5 map slots in the base case. Figure 2 shows that FastMap-MR with the sampling step is close to ideal speedup, while LMDS-MR's speedup reduces because its first-step cannot be sped up with more resources.

Next, we want to see how data reduction saves the time cost. Figure 3 shows the relationship of data sample size and time cost for the first step of LMDS-MR and FastMap-MR. Figure 3 shows that the sampling rate has significant impact on the time cost of the first step of LMDS-MR, as this step has a non-linear complexity. However, with a very low sampling rate (e.g., 0.000001) LMDS-MR can still return meaningful results as we will show later. When FastMap-MR algorithm uses sampling in the first step, the overall cost is significantly reduced and it scales well with higher sampling rates. We also give the time cost of FastMap-MR without

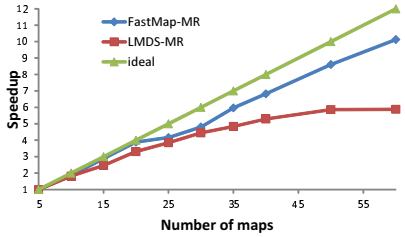

Figure 2. Speedup with more resources, k = 3, sampling rate = 0.00001 for Census dataset.

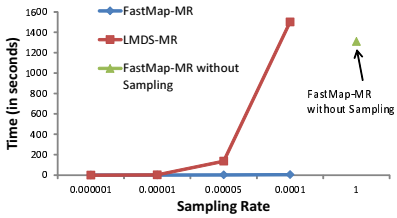

Figure 3. Running time vs. sampling rate with k=5 for Census dataset.

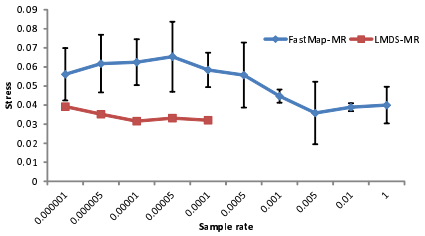

Figure 4. Stress vs. sampling rate, for Census dataset with k=5.

sampling (the right most point) for comparison, which uses MapReduce for the first step.

2) Data Reduction and Result Quality: Data reduction is useful only when the result quality is guaranteed. In this set of experiments, we investigate the impact of data reduction on the result quality. To observe the relationship between sample size and result quality, we choose a certain number of output dimensionality that gives satisfactory results. As we have mentioned, the quality is measured by the stress function and the clustering accuracy. In the experiments, we take a random sample set with sampling rate 0.0001 for computing these measures.

Figure 4 and 5 shows the two measures for the Census dataset, respectively. The result of LMDS-MR only includes sampling rates less than 0.0001 because higher rates will result in datasets too big to be processed for the first step of LMDS-MR. Overall, increasing sample size improves the quality for both methods and LMDS-MR slightly beats FastMap-MR.

Figure 6 and 7 for the Buzz dataset shows that raising sample size helps but does not affect the quality very much after certain sampling rate (e.g., 0.00001). However, the stress at such low sampling rate is already very good - less than 0.1 is considered as good quality in the literature [5].

3) Comparing two methods: One may wonder which of the two approaches: FastMap-MR and LMDS-MR is better for processing big datasets if all of the three aspects: scalability, time cost, and result quality, are considered. In this set of experiments, we will show for each testing dataset under the same setting (the fixed first-stage sampling rate and the output dimensionality), how these two methods perform.

We choose the dimensionality 6 and 9 for the Census dataset and the Buzz dataset, respectively, as our experiments have shown both methods give good result quality with acceptable costs. The sampling rate is set to 0.0001, with which both methods can possibly handle the sample dataset at the first stage. We also include the FastMap-MR without sampling for comparison as its overall cost is still lower than LMDS-MR for larger datasets.

We first show the time costs and the scalability for the two methods. Figure 8 shows all methods scale almost linearly to the dataset size. However, the increasing rate of LMDS-MR is much higher than other methods. At 10 GB, LMDS-MR takes about 2 times more than FastMap-MR without sampling and 8 times more than FastMap-MR.

Figure 9 and 10 show the result quality for the data size of 10 GB. LMDS-MR gives the best stress measure and clustering accuracy, although the differences on clustering accuracy is small. FastMap-MR without sampling is also better than with sampling. But overall, the stress measures are good, even for the worst case (all stress values < 0.07).

The experiment result shows a possible trade-off on selecting the methods: LMDS-MR is slower and less scalable due to the first stage processing, but it gives better results; the FastMap with sampling is more scalable, but gives slightly worse result quality.

V. Related Work

Although Euclidean embedding can also be used to reduce dimensionality of high-dimensional data in Euclidean space, we are more concerned about that the original metric space is non-Euclidean. Therefore, we will not specifically discuss another major body of work in dimensionality reduction [1].

So far, our study has been focused on FastMap [8] and Landmark MDS [6], which are also related to MetricMap [12]. In addition to this category of methods, there are several other types of related work as we have mentioned

The spring-mass models [13] are popularly used in information visualization. These methods calculate lower-dimensional coordinates by iteratively minimizing a cost, or stress, function that is proportional to the distance between the current coordinates and the given dissimilarities. A disadvantage of spring-based models, in general, is that they are subject to local minima.

Some approaches are proposed to address the special problems in Euclidean embedding. Cayton et al. [2] studied the Euclidean embedding of noisy distance values with semidefinite programming (SDP). Co-occurrence Data Embedding (CODE) [9] tries to embed objects of different types into a single Euclidean space, based on their co-occurrence statistics. It also uses SDP to model the MDS problem. It is well known that existing SDP solvers have high complexity, not ready for big data.

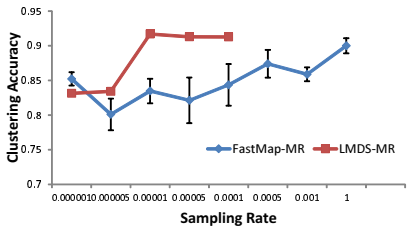

Figure 5. Clustering accuracy vs. sampling rate, for Census dataset, k=5.

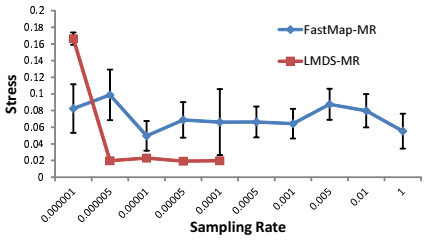

Figure 6. Stress vs. sampling rate, for Buzz dataset, k=9.

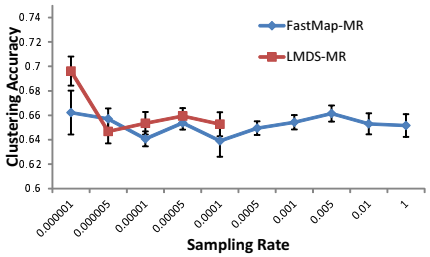

Figure 7. Clustering accuracy vs. sample rate, for Buzz dataset, k=9.

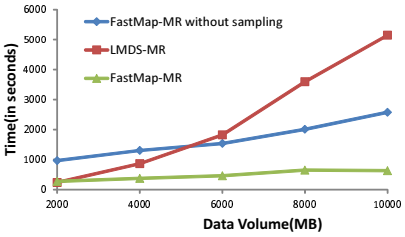

Figure 8. Running time vs. data volume, k =4, sample rate= 0.0001, Census dataset.

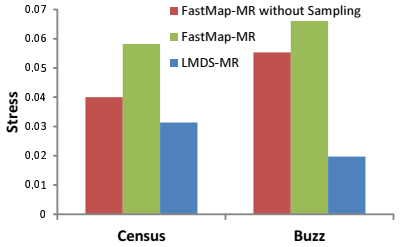

Figure 9. Stress for Census dataset with $k = 6$, and Buzz with $k = 9$, sample rate=0.0001.

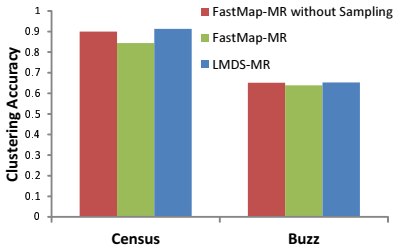

Figure 10. Clustering accuracy for Census and Buzz datasets.

VI. Conclusion

Euclidean embedding algorithms are important for visual data analytics. They are typically non-linear in both time and space complexity. There are a few methods have been developed aiming at handling large datasets, but only for running on a single machine. There is no study on big data that is stored in a distributed massively parallel infrastructure. In this paper we study the decomposition approach to scale up the existing Euclidean embedding algorithms. It tries to decompose an Euclidean embedding algorithm to steps that can be extended with either data reduction or parallel processing. We also show how to apply our approach with two algorithms: FastMap and Landmark MDS. We have done extensive experiments to study the scalability, the time cost, and the result quality of these algorithms on two real datasets. Experimental results on real datasets show that the extended algorithms have good scalability with well-preserved result quality. The outcome of this work will have major applications on visual exploration of big data that defined on non-Euclidean metrics.

References

[1] C. J. C. Burges, *Dimension Reduction: A Guided Tour*. Now Publishers Inc, 2010.

[2] L. Cayton and S. Dasgupta, "Robust euclidean embedding," in *Proceedings of the 23rd international conference on machine learning*. ACM, 2006, pp. 169–176.

[3] M. Chalmers, "A linear iteration time layout algorithm for visualising high-dimensional data," in *Visualization'96. Proceedings*. IEEE, 1996, pp. 127–131.

[4] K. Chen and L. Liu, "iVIBRATE: Interactive visualization based framework for clustering large datasets," *ACM Transactions on Information Systems*, vol. 24, no. 2, pp. 245–292, 2006.

[5] T. F. Cox and M. A. Cox, *Multidimensional scaling*. CRC Press, 2010.

[6] V. De Silva and J. B. Tenenbaum, "Sparse multidimensional scaling using landmark points," Technical report, Stanford University, Tech. Rep., 2004.

[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[8] C. Faloutsos and K.-I. Lin, *FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets*. ACM, 1995, vol. 24, no. 2.

[9] A. Globerson, G. Chechik, F. Pereira *et al.*, "Euclidean embedding of co-occurrence data," *Journal of Machine Learning Research*, vol. 8, 2007.

[10] D. Keim, "Visual exploration of large data sets," *ACM Communication*, vol. 44, no. 8, pp. 38–44, 2001.

[11] N. Linial, E. London, and Y. Rabinovich, "The geometry of graphs and some of its algorithmic applications," *Combinatorica*, vol. 15, no. 2, pp. 215–245, 1995.

[12] J. T.-L. Wang, X. Wang, K.-I. Lin, D. Shasha, B. A. Shapiro, and K. Zhang, "Evaluating a class of distance-mapping algorithms for data mining and clustering," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1999, pp. 307–311.

[13] M. Williams and T. Munzner, "Steerable, progressive multidimensional scaling," in *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*. IEEE, 2004, pp. 57–64.