Information Sciences 000 (2018) 1–21

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

An optimized unsupervised manifold learning algorithm for manycore architectures

Alexandro Baldassin [a], Ying Weng [b,*], Daniel Carlos Guimarães Pedronette [a], Jurandy Almeida [c]

[a] Department of Statistics, Applied Mathematics and Computing, São Paulo State University – UNESP, Rio Claro/SP, 13506-900, Brazil
[b] School of Computer Science, Bangor University, Bangor, LL57 1UT, United Kingdom
[c] Instituto de Ciência e Tecnologia, Universidade Federal de São Paulo – UNIFESP, São José dos Campos/SP, 12247-014, Brazil

ARTICLE INFO

Article history:
Received 30 January 2018
Revised 27 May 2018
Accepted 8 June 2018
Available online xxx

Keywords:
Multimedia retrieval
Unsupervised learning
Efficiency
Scalability
Parallelism

ABSTRACT

Multimedia data, such as images and videos, has become very popular in people's daily life as a result of the widespread use of mobile devices. The ever-increasing amount of such data, along with the necessity for real-time retrieval, has lead to the development of new methods that can process them in a timely fashion with acceptable accuracy. In this paper, we study the performance of ReckNN, an unsupervised manifold learning algorithm based on the reciprocal neighbourhood and the authority of ranked lists. Most of the related work in this field do not fully investigate optimization strategies, an aspect that is becoming more important with the high availability of manycore machines. In order to address that issue, we fully investigate optimization opportunities in this article and make the following three main contributions. Firstly, we develop an efficient and scalable method for storing and accessing the distances between objects (e.g., video or image) based on dictionaries. Secondly, we employ memoization to speed up the computation of authority scores, leading to a significant performance gain even on single-core architectures. Lastly, we devise and implement several parallelization strategies and show that they are scalable on a 72-core Intel machine. The experimental results with MPEG-7, Corel5k and MediaEval benchmarks show that the optimized ReckNN delivers both efficiency and scalability, highlighting the importance of the proposed optimizations for manycore machines.

© 2018 Published by Elsevier Inc.

1. Introduction

Multimedia data have become the mean of communication for a growing number of people sharing social content via mobile devices, social networks, and cloud environments. This change in human lifestyle and behavior has increased the availability of information in form of multimedia, enabling the creation of large multimedia collections. In this scenario, there is a growing demand for effective and efficient systems capable of managing such repositories and reduce the work and information overload when seeking a given content of interest.

In the last years, several research efforts have been focused on the investigation of approaches to exploit visual or audio properties and a similarity model aiming at supporting users to perform search on multimedia collections in a content-based fashion [6,48]. Usually, these approaches require a user to provide a given multimedia object as input, named as query.

* Corresponding author.
E-mail addresses: alex@rc.unesp.br (A. Baldassin), yingweng@gmail.com (Y. Weng).

https://doi.org/10.1016/j.ins.2018.06.023
0020-0255/© 2018 Published by Elsevier Inc.

Then, it is computed the similarity between the query and each object in the collection based on the distance between their corresponding feature representations. Many different features have been used to represent multimedia content. The first approaches were mostly based on low-level features, such as color, shape, and texture [24,39]. Motion patterns [1] and spatio-temporal properties [13] are some examples of other features used for encoding video content. Recently, mid- and high-level features based on local descriptors [3] and deep learning [46] have emerged as promising alternative solutions.

Regardless of the features used for encoding multimedia content, a key issue for content-based multimedia retrieval systems is to determine the similarity among multimedia objects. In general, it relies on distance functions used for comparing feature vectors [7]. However, pairwise distance measures, like the Manhattan or Euclidean distance, often fail to produce effective results. In fact, discovering and understanding the similarity structure of multidimensional patterns, especially in unsupervised scenarios, is of crucial relevance in data mining, pattern recognition, and machine learning applications [22]. Many data mining and pattern recognition employ high-dimensional representations, in which the *curse of dimensionality* phenomena is often an obstacle. Dimensionality reduction and manifold learning techniques are commonly used in such cases [20].

In retrieval scenarios, this has motivated the use of unsupervised post-processing methods for computing a more effective distance measure among multimedia objects. The basic idea of such methods is to replace pairwise similarities by more global affinity measures [5,31,50]. Diffusion process [18,50], graph-based learning methods [47], and iterative re-ranking techniques [36,37] are some approaches proposed in the literature. The intrinsic structure of the datasets has been also exploited by other recent manifold learning methods [2,25,33,34], yielding significant effectiveness gains in different scenarios.

In spite of the importance of unsupervised distance learning methods for improving the effectiveness of content-based multimedia retrieval systems, little attention has been given to efficiency and scalability issues [31]. Apart from the quality of the retrieval process, the time spent to obtain the results and the capability of handling growing multimedia collections are also indispensable [45]. Usually, distance learning methods need to be recomputed whenever significant changes are made in the datasets. Additionally, with the use of unsupervised algorithms as steps of semi-supervised interactive approaches [35], efficiency aspects became even more relevant. Semi-supervised learning methods exploit both the dataset manifold and user interaction, therefore requiring online performance of unsupervised stages.

Efficient and scalable implementations of such methods may be very challenging, demanding parallel and high-performance techniques and trade-off analysis. Some of these challenges are discussed by Valem et al. [45] in the context of unsupervised distance learning. Other works provide solutions based on General Purpose Graphics Processing Unit (GPGPU) [44,49]. There are also some works in other fields that propose optimizations such as artificial neural networks [9,15,42,43,51]. In most of the existing works, however, a detailed analysis on efficiency and scalability issues is missing.

This article aims to fill the gap by proposing several optimizations concerning the efficient use of memory and parallelism. Here, we focus on the Unsupervised Manifold Reciprocal k-Nearest Neighbors Graph [34] algorithm (*ReckNN*), which is based on the reciprocal neighborhood and a graph-based analysis of ranking references. The algorithm estimates the authority of ranked lists, spreading the similarity information throughout the dataset by a collaborative score. We opted for *ReckNN* because it represents a class of unsupervised ranking algorithms. Nonetheless, the optimizations discussed in this work should be equally applied to other ranking algorithms in the literature [32,36] with the same degree of effectiveness. Moreover, the proposed optimizations are aimed at the wide group of manycore processors, virtually present in very computer today. In addition, the ReckNN algorithm admit an implementation of linear complexity and storage requirements. However, such implementation depends directly on optimizations aspects discussed in this paper.

The major contributions of this paper are:

- Starting from a baseline implementation of *ReckNN*, it proposes a series of optimizations devised for large scale settings, such as the use of dictionaries to store the distances and the use of memoization to compute the authority scores (Section 4);
- It presents a shared-memory parallelization strategy for *ReckNN*, providing a solution using Open Multi-Processing (OpenMP) [8] and different synchronization methods (Section 5);
- It provides a thorough analysis of the performance with the different optimizations and synchronization methods using a modern 72-core Intel machine (Section 6). Our results show the feasibility of the proposed methods in a large scale learning setting, with speedups of up to 42x.

The remainder of this paper is organized as follows. Section 2 presents the problem formulation. Section 3 describes the Manifold Reciprocal kNN Graph algorithm. Section 4 discusses the proposed optimizations. Section 5 describes the parallelization and tradeoffs involved. Section 6 presents the experimental evaluation and discusses the results. Finally, Section 7 draws our conclusions.

2. Problem formulation

A formal definition of the rank model considered along the paper is described in this section. General multimedia retrieval tasks are considered, in which the goal is to retrieve the most similar objects to a given query contained in a collection of multimedia objects. A distance is computed among objects and the system response is often defined in terms of ranked lists, which provide an unified representation for distinct multimedia content (e.g., images, videos).

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

Formally, let $\mathcal{C} = \{obj_1, obj_2, \ldots, obj_N\}$ be a multimedia collection, where N is its number of objects. Let $\mathcal{D}$ be a descriptor which defines a distance function between two objects obj_i and obj_j as $\rho(obj_i, obj_j)$. A common data structure used to store the distances among objects in a collection is a squared distance matrix A, such that $A_{ij} = \rho(i, j)$.

Based on the distance function ρ, a ranked list τ_q can be computed in response to a multimedia query obj_q. The ranked lists can encode information from the entire collection, although the top positions of ranked lists contain the most relevant information. In this way, it is desirable that the ranked list τ_q considers only a subset of the top-n_s positions, such that $n_s \ll N$ and n_s is a constant value. In order to achieve efficiency and scalability requirements, such limitation is crucial, special for large-scale collections, where the full ranked lists are expensive to compute.

The ranked list $\tau_q = (obj_1, obj_2, \ldots, obj_{n_s})$ can be defined as a permutation of the subset $\mathcal{C}_S \subset \mathcal{C}$, which contains the most similar multimedia objects to the query obj_q, such that $|\mathcal{C}_S| = n_s$. A permutation τ_q is a bijection from the set $\mathcal{C}_S$ onto the set $[n_s] = \{1, 2, \ldots, n_s\}$. For a permutation τ_q, we interpret $\tau_q(i)$ as the position (or rank) of the object obj_i in the ranked list τ_q. We can say that, if obj_i is ranked before obj_j in the ranked list of obj_q, that is, $\tau_q(i) < \tau_q(j)$, then $\rho(q, i) \leq \rho(q, j)$. Note that, if the position of obj_i in the ranked list of obj_q is higher than the constant n_s, then $\tau_q(i) = n_s$.

Every object $obj_i \in \mathcal{C}$ can be taken as a query obj_q, producing a set of ranked lists $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$. The unsupervised learning algorithm aims at computing a new and more effective distance function $\hat{\rho}$ taking as input the set $\mathcal{T}$. The objective is to exploit the information encoded in the ranked lists in order to improve the effectiveness of the multimedia retrieval tasks.

Formally, the Manifold Learning Reciprocal kNN Graph [34] algorithm can be defined as a function f_r which takes a set of ranked lists $\mathcal{R}$ as the input and computes a new and more effective set of ranked lists $\hat{\mathcal{R}} = f_r(\mathcal{R})$. For readability purposes, we refer to this algorithm as simply *ReckNN* in future sections.

3. Unsupervised Manifold Reciprocal kNN Graph

This section presents the Unsupervised Manifold Reciprocal kNN Graph [34] algorithm. An overview and its main motivations are discussed in Section 3.1. Section 3.2 presents a formal definition and Section 3.3 provides an algorithmically view for computing the method.

3.1. Overview and motivation

The Manifold Reciprocal kNN Graph [34] is an unsupervised manifold learning algorithm which improves the effectiveness of retrieval tasks without the need of any user intervention. The algorithm exploits information about the dataset structure based on the information given by top positions of the ranked lists. In this way, the algorithm is able to propagate the similarity among neighbors taking into account the geometry of the dataset manifold.

The ranked lists represent a relevant source of similarity information, since they define relationships not only between pairs of multimedia objects (as distance functions), but also among all multimedia objects represented in a ranked list. The manifold learning approach analyzes the reciprocal references among ranked lists using a graph approach. The distances among multimedia objects are recomputed based on the graph structure, considering all references among multimedia objects at top positions of ranked lists.

3.2. Formal definition

The algorithm is mainly defined in terms of three rank-based scores, which are used for computing a new distance measure. The scores and the distance measure are briefly discussed and formally defined below.

3.2.1. Reciprocal kNN score

Once the nearest neighbor relationships are not symmetric [17], the reciprocal nearest neighborhood is a much stronger indicator of similarity than the unidirectional nearest neighborhood [40], reducing the risk of false positives at top positions of ranked lists.

The Reciprocal kNN Score R_s considers the position from which objects become reciprocal neighbors, as follows:

$$R_s(q, i) = \frac{max(\tau_q(i), \tau_i(q))}{n_s}. \quad (1)$$

3.2.2. Authority score

The Authority Score gives an unsupervised estimation of the effectiveness (quality) of each ranked list. The score is computed by analysing the Reciprocal kNN Graph's density, which represents the quantity of reciprocal references among objects at top positions of ranked lists. The approach is analogous to the PageRank [30] algorithm, considering that an effective ranked list has the best returned objects for a query referencing each other at the top positions of their ranked lists.

The ranked lists encode similarity information of the whole dataset, which can be exploited for improving effectiveness of other ranked lists. In this scenario, it is important to estimate the quality of the information provided. The Authority

Score is used for this purpose, representing an estimation of the authority of a given ranked list for contributing with other ranked lists.

Let $\mathcal{N}(q,k)$ denote a neighborhood set which contains the k most similar multimedia objects to obj_q. The score is computed in terms of reciprocal references among objects at top-k positions, and is formally defined as follows:

$$A_s(q,k) = \frac{\sum_{i\in\mathcal{N}(q,k)} \sum_{j\in\mathcal{N}(i,k)} f_{in}(j,q)}{k^2}, \quad (2)$$

where f_{in} returns 1 if $obj_j \in \mathcal{N}(q,k)$ and 0 otherwise. The score A_s is defined in the interval [0,1], where 1 represents a perfect score.

3.2.3. Collaborative score

A collaborative analysis is conducted, aiming at discovering similarity information encoded in the dataset manifold in order to compute a more global affinity measure. In this way, the Reciprocal kNN Collaborative Score exploits the information encoded in the ranked list according to their corresponding authority.

The score considers the co-occurrence of objects in ranked lists as an indication of similarity. For example, if two objects appear at the top positions of other ranked lists, they are probably similar to each other. The increase of similarity, however, is defined proportionally to the authority of ranked list which present the co-occurrence.

With the aim of giving higher weights to references at top positions, the collaborative score is computed at different values of k (varying from $1,2,\ldots,k_{max}$). The collaborative score C_s between two objects obj_q and obj_i is defined as:

$$C_s(q,i,k_{max}) = \sum_{k=1}^{k_{max}} \sum_{j\in\mathcal{C}} A_s(j,k)^2 \times f_{in}(q,i,j), \quad (3)$$

where f_{in} returns 1 if $obj_q, obj_i \in \mathcal{N}(j,k_{max})$ and 0 otherwise.

3.2.4. Reciprocal kNN distance measure

While the Reciprocal kNN Score (R_s) provides a locally similarity information, the Collaborative Score (C_s) gives a more globally measure considering the whole dataset. In order to compute a new and more effective distance measure ρ_r, both information are combined. All objects obj_q, $obj_i \in \mathcal{C}$ that present collaborative score $C_s(q, i, k_{max}) > 0$ have the distance between them updated as:

$$\rho_r(q,i) = \frac{R_s(q,i)}{1 + C_s(q,i,k_{max})}. \quad (4)$$

The multimedia objects with $C_s = 0$ keep their current ranking information. Formally, the distance is defined as $\rho_r(q,i) = \tau_q(i)/n_s$, if $\tau_q(i) \leq n_s$ or 1, otherwise.

The set of ranked lists is updated based on the distance ρ_r and the process can be repeated iteratively. Additionally, the value of k_{max} grows along iterations, since it is expected that non-relevant objects are moved out from the first positions of the ranked lists and, hence, k_{max} can be increased for considering more objects.

3.2.5. Convergence

The iterative redefinition of distances based on rankings are repeated until convergence. For the *ReckNN* method, a convergence criterion was defined in order to determine the number of iterations. Once the Authority Score gives an estimation of the quality of ranked lists, the convergence criterion exploits such information.

The process is computed while the authority of ranked lists is increasing more than a threshold ϵ per iteration. Aiming at verifying the convergence criterion, an average authority score between all objects in the collection is computed as follows:

$$G_s = \frac{\sum_{k=1}^{k_{max}} \sum_{j\in\mathcal{C}} A_s(j,k)}{k_{max} \times n} \quad (5)$$

Then, the difference between iterations is compared with the threshold ϵ in order to define the convergence criterion: $(G_s^{(t+1)} - G_s^{(t)}) > \epsilon$.

3.3. The algorithm

As discussed in previous work [34], the constrained size of ranked lists (to top-n_s positions) allows the development of an algorithm which assumes computational and storage requirements of only $O(n)$. However, for ensuring the complexity of $O(n)$, various efficiency aspects, such as data structure, should be properly considered. Since the focus of [34] was the effectiveness evaluation, only a direct and non-optimized implementation of the method was initially devised. This section describes the algorithm used in [34] for computing the *ReckNN*method, while next sections present the proposed optimizations, a detailed discussion about efficiency aspects and the evaluation on manycore architectures.

Algorithm 1: Baseline *ReckNN* Algorithm.

```
Input: R: set of ranked lists; C: Collection of all objects
Output: R̂: set of improved ranked lists
1  ConvScore ← 0;
2  R̂ ← R;
3  repeat
4      Initialize distance matrix A;
5      PrevConvScore ← ConvScore;
6      AcumScore ← 0;
7      foreach τ_q ∈ R̂ do
8          obj_q ← τ_q(1);
9          for k ← 1 to k_max do
10             AuthScore ← 0;
11             foreach obj_i ∈ N(q, k) do                          // A_s(q, k)
12                 foreach obj_j ∈ N(i, k) do
13                     if obj_j ∈ N(q, k) then AuthScore++;
14                 end
15             end
16             AuthScore ← AuthScore/k^2;
17             foreach obj_i ∈ N(q, k) do                          // C_s(q, i, k), j = q
18                 foreach obj_j ∈ N(q, k) do
19                     A[i, j] ← A[i, j] + AuthScore^2;
20                 end
21             end
22             AcumScore ← AcumScore + AuthScore;
23         end
24     end
25     foreach obj_q ∈ C do                                        // ρ_r(q, i)
26         foreach obj_i ∈ C do
27             if A[q, i] > 0 then
28                 R_s(q, i) ← max(τ_q(i), τ_i(q))/n_s;
29                 A[q, i] ← R_s(q, i)/(1 + A[q, i]);
30             end
31         end
32     end
33     foreach τ_q ∈ R̂ do
34         SortRankedList(τ_q);
35     end
36     ConvScore ← AcumScore /(k_max * n)                          // G_s
37     k_max++;
38 until (ConvScore − PrevConvScore) ≤ ϵ;
```

Algorithm 1 outlines a direct solution for computing the *ReckNN*method. The algorithm takes a set of ranked lists $\mathcal{R}$ as input and produces $\hat{\mathcal{R}}$, a new set of improved ranked lists using the Reciprocal kNN Graph context information. A distance matrix A, initialized in line 4, is used as the data structure for storing the distances and scores used throughout the algorithm.

The main goal of each iteration is to compute a new distance among multimedia objects, resulting in a new set of ranked lists. The distances are computed based on the scores defined in Section 3. Therefore, the first step consists in computing the Authority Score (Eq. (2)), given by lines 11–22. The Authority Score is computed for each ranked list (loop of line 7) and considering different values of k (loop of line 9).

The Authority Score is used for computing the Collaborative Score, defined in lines 17–22. The Reciprocal kNN Score is computed in line 28 and the new distance in line 29. The steps from line 4 to line 37 are repeated until a convergence criterion is reached, calculated by subtracting the previous convergence score from the current one and comparing the result against the threshold ϵ.

4. Optimizations

In this section, we introduce two optimizations over the baseline version of *ReckNN* presented in Section 3.3. Firstly, the use of a dictionary is suggested in Section 4.1, followed by a technique based on memoization to speedup the computation of the authority scores in Section 4.2. While the former optimization aims at decreasing space complexity, the latter is designed to decrease execution time, thus improving performance. Both optimizations were guided by an earlier profiling stage.

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

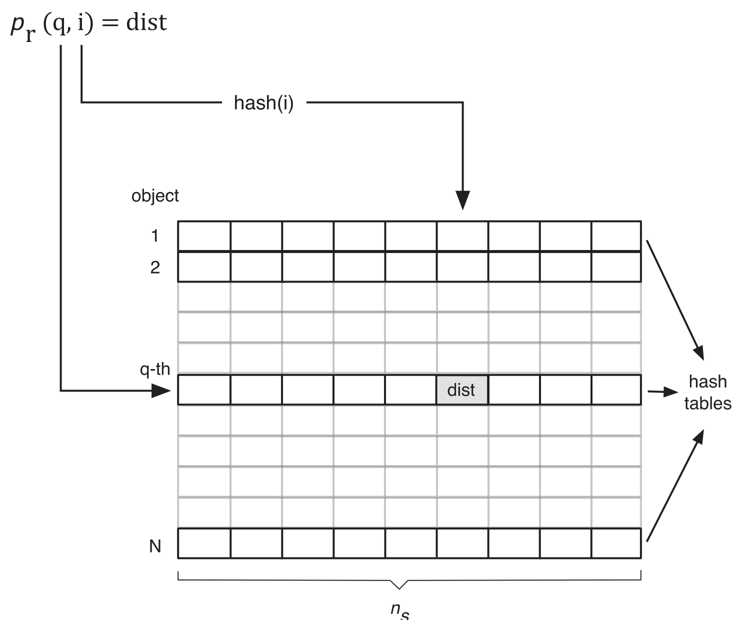


Fig. 1. A dictionary is implemented as an array of hash tables. Given a pair of objects (obj_q, obj_i), the index of the first object is used to select the qth hash table, while the second object is hashed and gives the position of the slot of the selected hash table in which the distance is to be stored.

4.1. Storing the distances

As discussed previously, the *ReckNN* algorithm computes the distance $\rho_r(q, i)$ between any pair of objects, obj_q, $obj_i \in C$, that appear in the set of ranked lists $\mathcal{R}$. Based on the distance ρ_r, the set $\mathcal{R}$ of ranked lists is updated by ensuring that, if $\rho_r(q, i) \leq \rho_r(q, j)$ then $\tau_q(i) < \tau_q(j)$, thus generating a new set $\hat{\mathcal{R}}$ of ranked lists.

The baseline *ReckNN*algorithm makes use of a $n \times n$ matrix to store the distances, as also shown in Algorithm 1 (lines 19 and 28). Although this seems reasonable for small collections, it clearly does not scale to larger sets as its space complexity is $O(n^2)$. Besides the space complexity, there are other unsatisfactory side effects resulting from a matrix representation. Firstly, notice that all pairs of objects that present a collaborative score greater than zero need to have their distances updated, as shown in lines 25–32 of Algorithm 1. Since the same matrix A is also used to store the collaborative score in the first part of the algorithm (line 19), one cannot predict in advance which pairs of objects have a nonzero distance, and thus all pairs must be checked, resulting in a time complexity of $O(n^2)$.

Secondly, the sorting procedure (line 34) may promote objects not initially in a given ranked list if a new computed distance is significative (not shown in the algorithm). When a matrix is used, for each ranked list τ_q, it is necessary to check the distances $\rho_r(q, i)$ for every $obj_i \in C$, whose time complexity is also $O(n^2)$. Note also that most entries in the matrix A will be empty since the number of objects in a ranked list is n_s, which tends to be much smaller than n. The matrix design clearly trades off easy of implementation and fast accesses for space and increased time complexity.

Therefore, a better strategy is needed in order for the *ReckNN*algorithm to be usable in a large scale learning setting. Our optimization employs a *dictionary* data type for that purpose, also known in the literature by *map* or *associative array* [26]. A dictionary is composed of a collection of (*key, value*) pairs, such that a *key* appears at most once in the collection. For the *ReckNN* algorithm, a *key* is a pair (obj_q, obj_i) of objects, and the *value* represents the distance between them. For good performance, it is necessary to have fast operations for searching a key and also updating it with a new value. Typical implementations of a dictionary make use of either a hash table or a search tree [11].

Considering the requirements for good performance and space limitations, our solution implements a dictionary based on an array of hash tables, with one hash table for each object, as illustrated in Fig. 1. Given a pair of objects (obj_q, obj_i), its respective distance is located within the dictionary in two steps: (i) the qth hash table is selected using the index of the first object in the pair (obj_q); (ii) a hash function is applied to the index of the second object ($hash(i)$), yielding a slot within the selected hash table with the distance (notice that the hashed key is also stored in the hash table, but it is not shown in Fig. 1).

For performance reasons, each hash table is implemented as an array with open addressing. When a collision happens (i.e., the key of the located slot does not match the hashed key), a linear probing with a interval of 1 is used to locate a next spot. For this kind of structure, it is very important to have a good hash function, since open addressing has the tendency of creating clustering of hash values that are consecutive in the probe order. Our implementation makes use of Knuth's multiplicative hash function since it is known to be simple and fast [19], as required by our optimization.

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

Another important issue in the design of hash tables with open addressing is the load factor: the ratio between the number of entries and the total number of slots in the hash table. A small value is desired, since a value closer to 1 means that the hash table is full. The number of slots in each hash table in our implementation is proportional to n_s, as indicated by Fig. 1. Recall that n_s is the number of objects in a ranked list and therefore is a good approximation for the size of each hash table.

Compared to the $O(n^2)$ space complexity of the matrix implementation, the dictionary approach is only $O(n \times n_s)$. Since n_s is independent and smaller than n, the dictionary space complexity is practically $O(n)$. Also, the time complexity to update the distances and promote new objects during the sorting procedure is now proportional to $O(n \times n_s)$, since it is only required to traverse the hash table of size $O(n_s)$ for a given ranked list.

As an example, Algorithm 2 is equivalent to lines 25–32 of Algorithm 1 when a dictionary is used. Notice that for each

Algorithm 2: Computation of Collaborative Authority Scores Using a Dictionary

```
1  ...
   foreach obj_q ∈ C do
2      foreach (key, value) ∈ HT[q] do
3          R_s(q, key) ← max(τ_q(key), τ_key(q))/n_s;
4          value ← R_s(q, key)/(1 + value);
5      end
6  end
7  ...
```

object obj_q (line 2), we traverse each pair (*key, value*) in its respectivehash table, $HT[q]$ (line 3), computing and storing the distance back in the dictionary (lines 4 and 5).

We give more details about the parameters of our implementation, as well as comment on different values for n and n_s, when discussing the performance results in Section 6.

4.2. Memoization of authority scores

Lines 11–22 of Algorithm 1 are responsible for computing the authority score $A_s(q, k)$ of ranked list τ_q (lines 11–16) for a given k, and then using this value to update the collaborative score C_s for each object pair $(obj_i, obj_j) \in \mathcal{N}(q,k)$ (lines 17–22). This process is repeated for increasing values of k according to the outer loop of line 9. In particular, the computation of the authority score $A_s(q, k)$ requires going through the first k objects of the ranked list of q ($\mathcal{N}(q,k)$) and, for each object obj_i, to check if the first k objects of its ranked list ($\mathcal{N}(i,k)$) is present in $\mathcal{N}(q,k)$. Notice that, for increasing values of k, much of the computation already performed previously (e.g., for $k-1$, $k-2$, ...) is computed again. In other words, when computing $C_s(q, k)$, if we could remember previous values for $C_s(q, k-1)$, the time efficiency of $C_s(q, k)$ could be improved.

The goal of this optimization is to *memoize* (i.e., remember) previously computed values of $C_s(q, l), l \in \{0, 1, 2, \ldots, k-1\}$, in order to compute $C_s(q, k)$. Lines 4–12 of Algorithm 3 presents the memoized version of $A_s(q, k)$ and replaces lines 11–16

Algorithm 3: Memoization Optimization

```
1  ...
   LastK ← 0;
2  for k ← 1 to k_max do
3      obj_l ← τ_q(k);
4      foreach obj_i ∈ N(q, k − 1) do
5          if obj_l ∈ N(i, k − 1) then LastK + +;
6          if τ_i(k) ∈ N(q, k) then LastK + +;
7      end
8      foreach obj_i ∈ N(l, k) do
9          if obj_i ∈ N(q, k) then LastK + +;
10     end
11     AuthScore ← LastK/k^2;
12     ...
13 end
14 ...
```

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

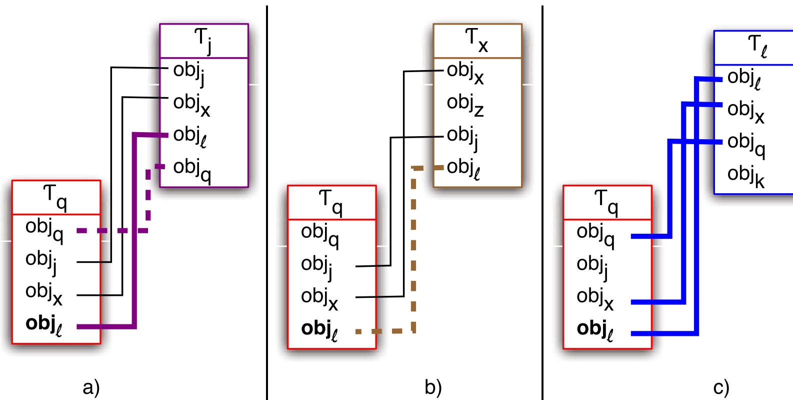


Fig. 2. Computation of $A_s(q, 4)$ using memoized values of $A_s(q, 3)$. Each of the parts represent the new (thick lines) and memoized (regular lines) connections for each object in $\mathcal{N}(q, 4)$: a) ranked list of obj_j; b) ranked list of obj_x; and c) ranked list of obj_l.

of Algorithm 1. The memoized values are accumulated incrementally and stored in a new variable named *LastK*, reset before entering the loop that increments k (line 2).

To better understand Algorithm 3, Fig. 2 illustrates the extra work (thick lines) necessary to calculate $A_s(q, 4)$, assuming the value of $A_s(q, 3)$ is already memoized (regular lines). Initially, notice that the extra computation refers to the new kth object (line 4), which is obj_l in the given example (last object of τ_q). Two main steps are required. Firstly, lines 5–8, it is necessary to check in the ranked lists of the first $k-1$ objects if: (i) obj_l is now present in the top $k-1$ positions (line 6) – this is the case for $\tau_j(3)$ in Fig. 2a (represented with a full thick line); (ii) the last object ($\tau_i(k)$) is in the ranked list of obj_q ($N(q, k)$) (line 7) – this is the case for obj_q and obj_l in Fig. 2a and Fig. 2b, respectively (represented with a dashed thick line). Finally, the algorithm checks whether each object in $\mathcal{N}(l, k)$ is present in $\mathcal{N}(q, k)$ (lines 9–11), incrementing the value of *LastK* if the condition is true. This is the case for the first three objects of τ_l, as shown in Fig. 2c. Line 12 then updates the authority score by dividing the accumulated values by k^2.

It should be noticed that Fig. 2 does not show the trivial connections of τ_q with itself (which is always k, from which $k-1$ has been already memoized), only with τ_j (Fig. 2a), τ_x (Fig. 2b), and τ_l (Fig. 2c). Therefore, for the given example, the authority score $A_s(q, 4)$ is equal to $14/4^2$. The time complexity for the computation of an authority score in Algorithm 1 (lines 11–16) is $O({k_{max}}^3)$, since the membership test in line 13 also needs to iterate through every object in $\mathcal{N}(q, k)$. On the other hand, the memoized version of the authority score computation given by Algorithm 3 is only $O({k_{max}}^2)$. Although k_{max} can be considered a constant, the authority score is computed for every object in the collection, with different values for k, resulting in a significant portion of the execution time. As such, this optimization provides an important performance boost, as discussed in Section 6.

5. Parallelization

In this section a parallel version of *ReckNN* aimed at multicore processors is devised. Most of current processors provide at least two execution cores, and thus exploiting parallelism is an important characteristic for large scale learning algorithms. In particular, the parallelization techniques we present in this section are orthogonal to the optimizations described in the previous section (Section 4). We start by giving a brief background on parallel computing in Section 5.1, followed by the parallelization strategy in Section 5.2, thread management in Section 5.3, and synchronization techniques in Section 5.4.

5.1. Background

Parallelization involves splitting the work evenly among different *threads* such that they can execute concurrently and independently of each other. Ideally, if N threads are used to do the same job that required T units of time sequentially, one would expect a parallel execution time of T/N. In this case, we say that the *speedup* achieved by the parallelization is $\frac{T}{T/N}$, or simply N. Notice that the speedup is calculated by dividing the original sequential time by the new parallel time. As a consequence, speedup numbers are expected to be higher than one, for otherwise the parallel version is slower than the sequential one, and the term *slowdown* is used instead.

Achieving a perfect speedup is, however, not achievable in practice due to a host of different factors. Firstly, it is not always easy to split the work evenly across the working threads, in which case it is said that the parallelization suffers from *load imbalance*. Secondly, not all parts of a serial program may be amenable to parallelization. The impact on the speedup due to serial fragments is given by Amdahl's Law [4]: $S(N) = \frac{1}{(1-p)+(p/N)}$, where p is the fraction of the program that was parallelized, and N is the number of threads or processors. Amdahl's law states that the speedup is limited by the serial fragment $(1-p)$ so that, if 10% of the program is inherently serial, the maximum speedup is 10 regardless of the number of processors N employed.

Finally, speedup can be affected by the amount of *communication* performed by the working threads. For instance, a given part of the code may only be executed after all threads communicate their partial results to others. Communication is particularly important in distributed systems as the link latencies are more notable. However, current shared memory architectures are adopting non-uniform memory access (NUMA) interconnect interfaces and communication is starting to become a bottleneck. This is particularly important for the parallelization strategies presented in this paper, since the focus of the work is on multicore architectures. These architectures use the shared memory paradigm, wherein a thread may communicate to others by reading/writing to the same address space. When multiple threads are acessing the same position in this address space, and at least one of them is writing, *consistency* errors may appear. One typical consistency error is known as the *lost update*, manifested when two or more threads update the same memory position and some values are overwritten by other threads. These types of consistency errors do not always manifest themselves, as they are timing-dependent, a scenario known in the literature as *race condition*. Consistency errors are usually avoided by means of *synchronization*. A popular form of synchronization is to acquire a *lock* before attempting to update the data, releasing the lock afterwards. The region of code protected by the lock is commonly referred to as the *critical section*.

5.2. Parallelization strategy

A brief look at Algorithm 1 reveals a lot of potential for parallelism. For instance, there are three big loops whose bodies could be executed concurrently with almost no interference: (i) authority score loop (lines 7–24); (ii) distance computation loop (lines 25–32); and (iii) sorting loop (lines 33–35). Notice, however, that these loops need to be executed in order: it is not possible to start the distance computation loop before completing the authority score computation loop.

Since the focus of our optimizations are on manycore architectures, our parallelization strategy assumes a shared-memory model. A natural way of devising a parallelization of *ReckNN* is to evenly split the data (i.e., the ranked lists or collection of objects) among the threads that perform the same task which, in our case, is simply the actions contained in the body of the three main loops identified previously. Such strategy is known as *data parallelism* in the literature. Another alternative would be to use *task parallelism*, with threads performing different tasks simultaneously. However, as can be seen from the description of Algorithm 1, there is not a large number of different tasks to justify using task parallelism.

One way to perform the required parallel computation is known as *fork-join*: before entering each of the loops, N threads are forked and the data is split evenly among each one – each of the threads perform the computation on $\mathcal{R}/N$ distinct ranked lists or $\mathcal{C}/N$ distinct objects; when the loop is over, the threads are joined back together before continuing to the execution of the next step. This particular type of parallelism is also known as *loop-level parallelism*, since what happens in practice is that a particular thread is responsible to execute a given number of iterations of each loop. As long as iterations are independent from each other, they can be executed by the threads concurrently.

It is easy to see that the iterations of the second (distance computation) and third (sorting) loops are completely independent: the former because the computation performed in lines 28–29 only affects $A[q, i]$, which can only be written by the thread assigned to object q; the latter because the sorting of one ranked list in line 34 does not depend on any other ranked list. Unfortunately, the same cannot be said about the authority score loop. Notice that the update performed to $A[i, j]$ in line 19 may be executed simultaneously by multiple threads. For instance, there might be objects (i, j) in multiple ranked lists assigned to different threads, in which case the threads will attempt to update $A[i, j]$ concurrently, given rise to the lost update problem discussed earlier in this section. Moreover, the update of `AcumScore` in line 22 also exhibits the lost update problem.

To summarize the preceding discussion, a concurrent implementation of *ReckNN* requires a parallel infrastructure that allows: (i) the fork and join of threads so that the iterations of the three main loops can be evenly assigned to different threads; (ii) a way to force thread ordering between the loops, so that no thread is allowed to enter the next loop until all of them have finished the previous one; (iii) a synchronization strategy to avoid the lost update problem presented by the authority score loop. OpenMP (Open Multi-Processing) [8] is an Application Programming Interface (API) for shared memory multiprocessing that meets all these requirements. It consists of a set of compiler directives, library procedures and environment variables, and is supported by a group of major software and hardware vendors (e.g., Intel, AMD, Nvidia). Pthreads [28] could also be used in the parallelization process, but OpenMP has a clear advantage given its high-level API which further translates into higher productivity and portability.

5.3. Thread management and affinity

One of the main advantages of using OpenMP is that one can start from a working sequential version of the code and incrementally add compiler directives so as to build a parallel version. In order to show how OpenMP can be employed to implement a parallel version of *ReckNN*, we will use a syntax close to the one used by the C language. In this language, OpenMP uses *pragmas* do specify parallel constructs. All OpenMP directives must start with `#pragma omp`.

Of particular interest here is the `#pragma omp parallel for` directive, since it specifies that the iterations of the loop following the directive must be executed in parallel. Algorithm 4 highlights how this directive can be applied to the sequential version of Algorithm 1. Notice that the pragmas are used just before the loops (lines 7, 10, 13). Recall from Algorithm 1 that the variable `AcumScore` is summed up in the first loop and then used to calculate the convergence score. By default, the scope of all variables entering a parallel region in OpenMP is global. As a result, `AcumScore` may also suffer

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

Algorithm 4: Parallelization of *ReckNN* through OpenMP directives.

Input: R: set of ranked lists; C: Collection of all objects
Output: R': set of improved ranked lists

```
1  ConvScore ← 0;
2  R' ← R;
3  repeat
4  |  Initialize distance matrix A;
5  |  PrevConvScore ←ConvScore;
6  |  AcumScore ← 0;
7  |  #pragma omp parallel for reduction(+:AcumScore )
8  |  foreach τ_q ∈ R' do
   |  |  // calculation of authority and collaborative authority scores
9  |  end
10 |  #pragma omp parallel for
11 |  foreach obj_q ∈ C do                                    // ρ_r(q, i)
   |  |  // calculation of the distance
12 |  end
13 |  #pragma omp parallel for
14 |  foreach τ_q ∈ R' do
15 |  |  SortRankedList(τ_q);
16 |  end
17 |  ConvScore ←AcumScore /(k_max*n)                          // G_s
18 |  k_max ++
19 until (ConvScore − PrevConvScore) ≤ ε;
```

from the lost update problem. This can easily be solved by adding the OpenMP clause `reduction`. It specifies the type of the reduction (e.g., addition, multiplication) and the variable (of scalar type) to be reduced. In line 7 of Algorithm 4, OpenMP will create a private copy of `AcumScore` for each thread and add their values at the end of the parallel loop. The `parallel for` directive also implicitly forces the threads that finished their computation to wait for the others before proceeding, thus solving the thread ordering problem.

Another important aspect of the `parallel for` construct is how the iterations of the loops are divided among the threads. OpenMP provides a clause, `schedule` (not shown), that allows different forms of scheduling. In our case, this division is done statically since the number of ranked lists $\mathcal{R}$ and objects $\mathcal{C}$ are known for each dataset. Roughly, with a total of N threads, each one will process a total of $|\mathcal{R}|/N$ or $|\mathcal{C}|/N$ items. Notice that the amount of work performed by each thread is virtually the same, thus providing a good load balancing.

One final important detail about thread management involves the assignment of threads to cores at runtime. Usually the operating system is responsible for binding threads to cores but, as setting the right *thread affinity* can have a dramatic effect on performance depending on the machine topology, parallel runtimes (such as OpenMP) also provide ways to pin threads to physical processing units in order to make the most out of multicore machines. Fig. 3 exhibits a machine with two *sockets* or processors, each providing two cores with two *thread contexts* (TC) each. A thread context is a processing element as seen by the operating system. In the referred figure, threads can be bound to eight different TCs. The figure also shows 3 different thread affinity settings: (i) `Compact` (C), assigns a thread to a free TC as close as possible to a previous one (same core); (ii) `Scatter` (S), distributes the threads as evenly as possible across the entire machine; and (iii) `Balanced` (B), binds threads to at most one TC per core, with cores as close as possible to a previous one (same socket) – a mix of `Compact` and `Scatter`.

While `Compact` provides better locality, `Scatter` benefits threads which do not need to share data (each one can have the entire cache for itself, for instance). The reasoning behind `Balanced` is that TCs belonging to the same core are usually implemented as simultaneous multithreading (e.g., Intel's Hyperthreading), which allows some functional units to be shared among TCs. Therefore, allocating only one TC per core tends to maximize its performance. The best affinity is application-dependent and we discuss the case of *ReckNN* when analyzing its performance in Section 6.1.

5.4. Synchronization

Recall that the main synchronization issue with the parallelization of *ReckNN* concerns the update of the distance matrix (line 19 of Algorithm 1). Recall also that the update of `AcumScore` (line 22 of Algorithm 1) was solved by using a reduction mechanism, wherein private copies of `AcumScore` are created for each thread, and their values summed up at the end. While in theory it would be possible to apply the reduction strategy to solve the problem with the update of the distance

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

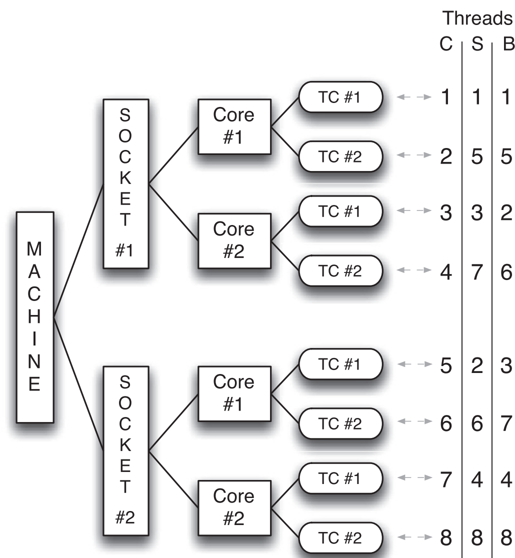


Fig. 3. Different thread affinities for a dual socket, 4-core machine, each with 2 thread contexts (TC) per core: `Compact` (C), `Scatter` (S), and `Balanced` (B).

matrix, notice that it would require the creation of multiples copies of the matrix (or the dictionaries in the case of the optimization discussed in Section 4.1), which is really not an option in a scalable setting. The creation of multiple copies of large data structures also tends to pollute the processor cache, thus decreasing performance.

One possible approach to synchronize the access to the distance matrix is to use critical sections. OpenMP provides the `#pragma omp critical` directive for that purpose. One downside of using critical sections is that there is an overhead to acquire and release the lock. Moreover, critical sections tend to serialize the code with, according to Amdahl's Law discussed earlier, negatively impact the performance.

Current processors provide atomic instructions that can read, modify, and write a single datum atomically, that is, without the interference of any other instructions. One such popular instruction is called *Compare-And-Swap* (CAS), virtually present in all existing processors. As long as the data to be modified fits the word size of the processor architecture (i.e., 64 bits), a CAS instruction can be used. Each element of the distance matrix is a floating point, single-precision, data, which on most 64-bit architectures are 4 bytes, or 32 bits. Therefore, the update could be implemented using a CAS instruction. OpenMP provides the `#pragma omp atomic` for that purpose and was the preferred method employed to synchronize the access to the distance matrix. We did run some experiments using the first two approaches (reduction and critical sections), but their performance were subpar.

The synchronization for the update of the distances using the dictionary-based optimization is slightly more complicated. While it is possible to protect each update using a critical section, this solution does not scale because it serializes the entire operation. Thus, our solution uses a *non-blocking* algorithm for updates in the dictionary. Non-blocking algorithms usually make use of CAS operations briefly introduced earlier. A CAS instruction takes three arguments: a memory location, an expected value, and a new value to be written. It then compares the contents of location with the expected value and, if they are equal, it atomically assigns the new value to location. Otherwise, location is left untouched. A boolean value is returned indicating whether the replacement occurred.

The pseudo-code for the *lock-free* dictionary update operation is presented in Algorithm 5. It is a simplification of similar approaches in the literature [27,29], particularly tailored to our problem. In particular, we make the following simplifications: (i) the value zero (0) is used to represent empty slots (keys must be non-zero); (ii) each hash table contains a fixed amount of entries; and (iii) there is no delete operation. Also, notice that the only time that updating the distances require synchronization is when the authority score is calculated in the first loop. The search and insert operations performed as part of the second and third loops are private to a specific hash table, and thus they do not need to be thread-safe.

Algorithm 5 takes as input a key, comprised of an object pair (obj_q, obj_i), and a value to be added (`val`). Initially, all entries (keys and values) are zero. In the first step (line 1), the correct hash table is selected using obj_q as index. Next, the position inside the selected hash table (`Index`) is calculated according to a hash function (line 2). The loop from lines 4 to 18 is responsible for searching a key in the selected hash table (line 6) such that its value is either zero (empty) or matches obj_i (line 7). If that is not the case, it probes the next position until all slots are eventually inspected (lines 17 and 18). If an empty slot is found (line 8), it attempts to atomically update the position using obj_i as the key (line 9). If it fails, it retries by returning to the beginning of the loop (line 10). Otherwise, the CAS operation succeeded and the new key obj_i was correctly written in the hash table. It then proceeds to atomically update the value (line 14).

ARTICLE IN PRESS

Algorithm 5: Lock-free dictionary update.

Input: (obj_q, obj_i): key comprised of the object pair;
val: distance between the pair (obj_q, obj_i);
Output: **true** if success; **false** if dictionary is full;

```
1  HTable ← HT[obj_q];
2  Index ← hash(obj_i);
3  Start ← Index;
4  repeat
5      Index ← Index%NK;
6      Key ← HTable[Index].key;
7      if (Key == 0)||(Key == obj_i) then
8          if Key == 0 then
9              if !CAS(HTable[Index].key, 0, obj_i) then
10                 continue;
11             end
12         end
13         #pragma omp atomic
14         HTable[Index].value = HTable[Index].value + val;
15         return true;
16     end
17     Index ← Index + 1;
18 until Index! = Start;
19 return false;
```

To understand how Algorithm 5 provides correct synchronization, consider two different scenarios. Firstly, assume that two different threads are trying to update the same entry in the hash table with different keys. Because of the CAS operation, at most one will succeed. The other will retry and both comparisons at line 7 will fail. It will then try to update another slot. Now consider two threads updating the same entry with the same key. Again, only one will succeed, but when the other thread retries, the second comparison at line 7 will now evaluate to true, and the thread will correct update the value at line 14. Notice also that the update in line 14 needs to be atomic, for there is a chance of two or more threads with the same key to update the value at the same time.

The last synchronization alternative we employed in the parallelization of the *ReckNN* algorithm is *Hardware Lock Elision* (HLE). This synchronization approach was proposed about 15 years ago [41], but only recently appeared on Intel processors as an instruction set extension [12]. From a programmer's point of view, the code is synchronized using lock and unlock operations. At runtime, the hardware first elides the acquisition of the lock and execute the critical section optimistically. If no other thread changes any of the memory locations accessed by the thread executing the critical section, the unlock operation is also elided and the changes are made permanent. If one or more threads also enter the critical section and their accesses conflict with other threads, the hardware will abort the execution of the conflicting threads and retry pessimistically by acquiring the respective lock. With HLE, as long as there are no conflicts at runtime, threads can execute the critical section concurrently.

6. Experimental evaluation

In this section, we provide a quantitative analysis of the optimizations described in previous sections on a modern multicore machine. Section 6.1 describes the methodology, platform and datasets employed in the experiments, followed by the analysis of the impact of the optimizations in Section 6.2. The performance achieved by the proposed parallelization strategy and different synchronization methods is discussed in Section 6.3. Finally, Section 6.4 discusses the impact of not using any synchronization.

6.1. Setup

For the experiments, four specific versions of the *ReckNN* algorithm were implemented using the C language, according to the different optimizations proposed in this paper:

- `Naive`: the baseline implementation using a distance matrix (Algorithm 1). Notice that this is the original version as proposed in [34];
- `Dictionary`: replaces the distance matrix with a dictionary, as discussed in Section 4.1. After some tests, we opted to use Knuth's multiplicative hash function [19] since it provided the best results. The size for each hash table was set to $n_s/1.5$, providing a load factor of about 50%;

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

- `Memoiz`: makes use of the memoization optimization described in Section 4.2, while still using the distance matrix;
- `Optimized`: both dictionary and memoization are present.

To help in the analysis, the results present execution time for three different phases of *ReckNN*:

- `Initialization`: responsible for initializing the main data structures – roughly the distance matrix in `Naive` and `Memoiz`, and the dictionary in `Dictionary` and `Optimized`;
- `AuthScore`: calculates the collaborative authority score – the first loop of Algorithm 1 (lines 7–24);
- `DistSort`: updates the distances and sorts the ranked lists – second and third loops of Algorithm 1 (lines 25–36).

The experimental evaluation was conducted on three different datasets, with different characteristics. Table 1 presents a general description of the datasets used. Diverse multimedia content is also considered, involving two image datasets and one video dataset. The MPEG-7 and Corel5k image datasets became *de facto* benchmarks and are commonly used in the literature for evaluation and comparison of image retrieval methods. The use of such common benchmarks allow the comparison and reproducibility of experiments. Since the video retrieval literature is more recent, the MediaEval dataset was chosen due to its use in a relevant competition in the area. Three different visual descriptors are considered, defined according to the dataset, including color, shape, and motion features.

Since the evaluation is focused on efficiency aspects, the same parameters settings are used for all datasets. The ReckNN algorithm consider three parameters: k_{max}, ϵ, and n_s. The parameters k_{max} and ϵ are related with the neighborhood size and convergence, which have direct impact on the effectiveness results. In order to evaluate the optimizations and efficiency aspects, without affecting the effectiveness of the method, the values suggested by the original paper [34] are used ($k_{max} = 15$ and $\epsilon = 0.0125$). Regarding the n_s parameter, which defines a trade-off between effectiveness and efficiency, different values were used according to the dataset (Table 1) with the aim of evaluating distinct scenarios and the impact on the efficiency results.

The machine and system configurations used in the experiments are described in Table 2. The machine topology is similar to that in Fig. 3, but each of the 2 sockets has 18 cores, each with 2 thread contexts. As a result, there are 72 different cores from the operating system point of view. Like most Intel processors, there are 2 private levels of cache (L1 and L2), with the last level (L3) shared between all cores in the socket. There is also a total of 128 GB of DRAM available. The access to memory is not uniform (NUMA), that is, if a memory position accessed by a core in the first socket is present in the DRAM slot assigned to the second socket, the latency will be higher than if the memory was present in the first socket. The Linux kernel version used (3.10.0) automatically deals with distributing and migrating memory pages around in order to reduce the NUMA impact (this feature is commonly known as AutoNUMA [10]). Finally, the GNU C Compiler was used to compile all versions of *ReckNN* with optimization flag `-O3`.

As can be seen, our evaluation makes use of a high-end machine with an Intel 72-core processor. This is very important in order to assess the effectiveness of the proposed optimizations since they were devised for manycore architectures. We expect to see the same performance trends for lower configurations but with reduced levels of parallelism.

Each experiment was executed 10 times and the results discussed in the next subsections represent the averaged execution time. In all cases, the confidence interval (95%) was calculated and the results are all consistent. The scalability evaluation performed in Section 6.3 uses the following increasing thread configurations: 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, and 72. This large variation is enough to appropriately assess the scalability of the system given the chosen machine. We also considered the thread affinity configurations discussed in Section 5.3 in our experiments. Overall, the execution time we measured with all of them were similar when the confidence interval is considered. Therefore, the results presented here use the `Balanced` policy as default. The only exception we found concerning the choice of thread affinity was for the synchronization method based on HLE, which is further explained in Section 6.3.

6.2. Impact of optimizations

In this section, we quantity the impact of the several optimizations proposed in this paper. Fig. 4 shows the sequential execution time for the three datasets: `MPEG-7` (Fig. 4a), `Corel5k` (Fig. 4b), and `MediaEval` (Fig. 4c). Firstly, notice that the execution time for each dataset is proportional to their collection size. Therefore, `MPEG-7` exhibited the fast execution time, followed by `Corel5k` and `MediaEval`.

Another important behavior displayed by Fig. 4 is that the impact of the `Initialization` phase is negligible for `MPEG-7` and `Corel5k`, but it starts to become relevant for `MediaEval` when the distance matrix is used (`Naive` and `Memoiz`) due to its larger size. On the other hand, the dictionary used by `Dictionary` and `Optimized` does not exhibit any significant overhead since the time required to initialize this data structure is $\mathrm{O}(n)$ instead of $\mathrm{O}(n^2)$.

For all the datasets, the reduced execution time provided by the memoization optimization (`Memoiz` and `Optimized`) in the `AuthScore` phase is very noticeable. Consider, for instance, `Corel5k`: the `AuthScore` phase of `Memoiz` is 4.9x faster than the same phase of `Naive` (1.36s versus 6.63s). If we compare the `AuthScore` phase of `Memoiz` against `Optimized`, it can be seen that the latter is slower in all datasets, because the time required to access the dictionary is higher than accessing the distance matrix. However, the contrary is true if we consider the `DistSort` phase: the use of the dictionary in `Dictionary` and `Optimized` makes this phase faster as the time required to update the distance and sort the rank lists is proportional to $O(n \times n_s)$ instead of $O(n^2)$. Since values of n and n_s are similar in `Corel5k`, it does not benefit from this feature provided by the dictionary.

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

Table 1
Datasets, descriptors and parameters used in the experimental evaluation.

Dataset	Content Type	General Description	Descriptor	Dataset Size (n)	Rk. List Size (n_s)	# Classes	Iterations
MPEG-7	Images of shapes	A well-known dataset [21] composed of 1400 shapes divided in 70 classes. Commonly used for evaluation of unsupervised learning approaches.	Aspect Shape Context (ASC) [23]	1400	200	70	6
Corel5K	Color scenes	A heterogeneous sub-set [14] of Corel image dataset, consisting of similar semantic content (such as beach, bird, mountain, jewelry, sunset, etc).	Auto Color Correlograms (ACC) [16]	5000	5000	50	12
MediaEval	Videos	Provided by the MediaEval 2012 organizers, the dataset contains 3288 h of video collected from the blip.tv, distributed among 26 genre categories.	Histogram of Motion Patterns (HMP) [1]	14,838	1000	26	12

ARTICLE IN PRESS

Table 2
Machine/System configuration used in the experiments.

Component	Configuration
Processor	Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30 GHz
Sockets	2
Cores per socket	18
Threads per core	2
L1 instruction cache	32 KB, 8-way set associative (private)
L1 data cache	32 KB, 8-way set associative (private)
L2 (unified)	256 KB, 8-way set associative (private)
L3 (unified)	45 MB, 20-way set associative (shared)
RAM	128 GB DDR4-2400, 8*16 GB DIMMS
Max bandwidth	71.53 GB/s
Operating system	Linux kernel 3.10.0 (64-bit) – Oracle Server 7.2
Compiler	gcc (GCC) 4.8.5 (Red Hat 4.8.5-4)
Optimization flag	-O3

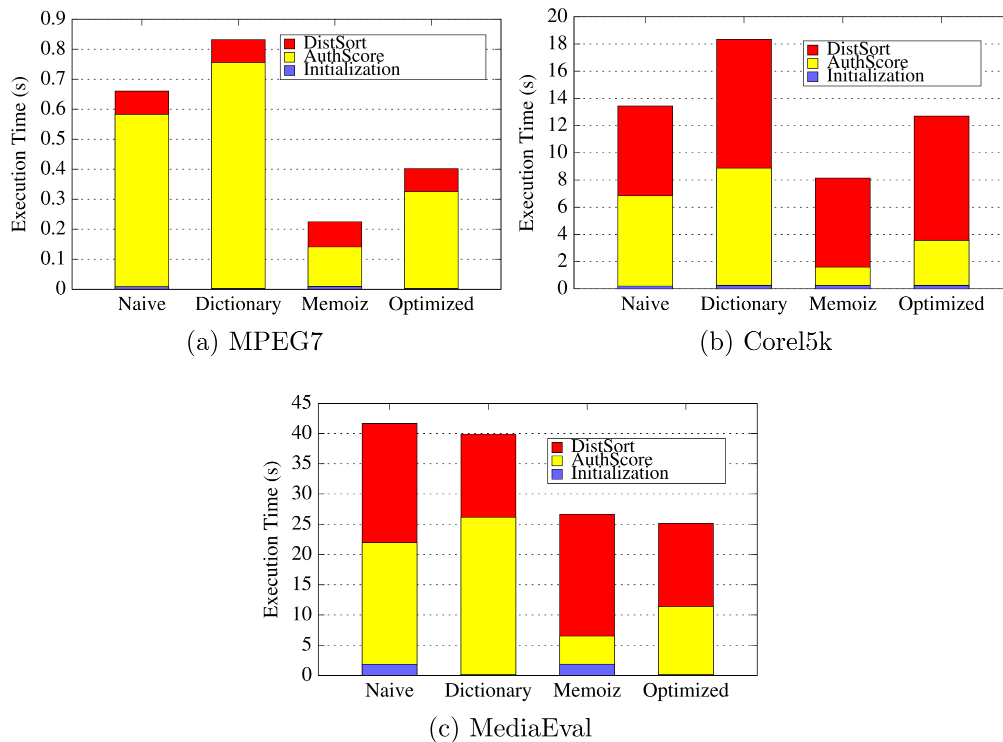


Fig. 4. Broadwell Optimization Results (sequential version).

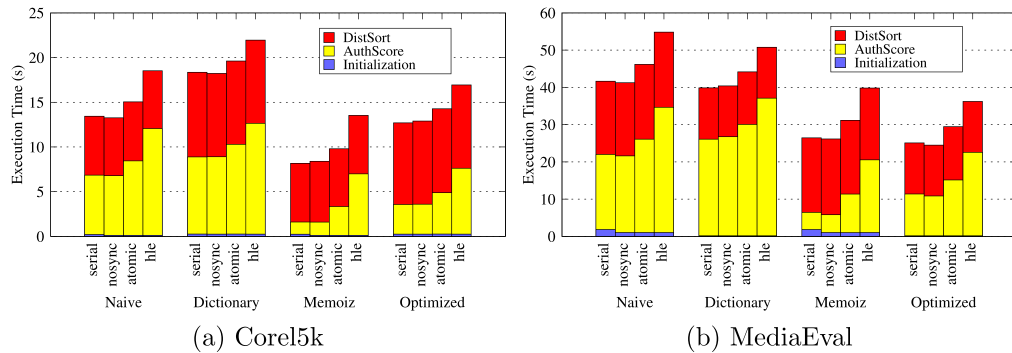
Overall, the results indicate that, for smaller collections, the use of the distance matrix and memoization is the best alternative. However, as the collection size grows, replacing the distance matrix with a dictionary is the best choice. Since our goal is to provide an efficient implementation of *ReckNN* for large-scale distance learning, the results strongly indicate that the proposed optimizations (i.e., memoization and dictionary) are effective. For instance, with `MediaEval`, the `Optimized` version is already faster than `Memoiz` (25.1s versus 26.6s, respectively)

6.3. Parallelization results

In this section, the parallelization strategy and different synchronization methods presented in Section 5 are analysed. In particular, the following configurations are considered:

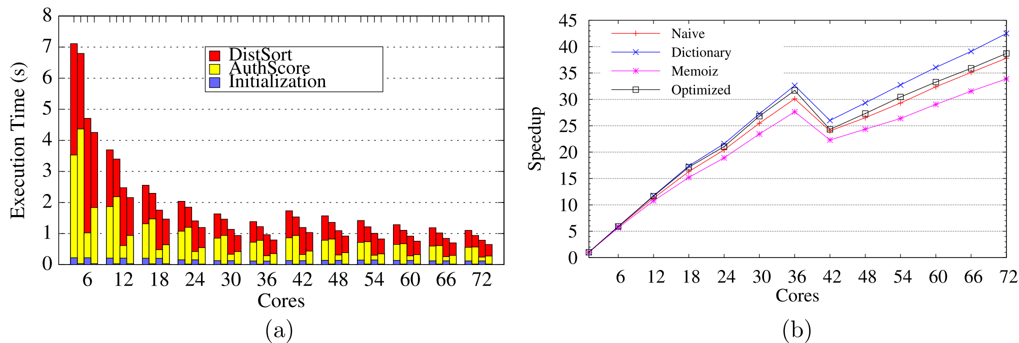
- `serial`: the serial version as presented in the previous section;
- `nosync`: in this version we execute the code in parallel, but do not make any effort to avoid the lost update problem. As a result, the accuracy of ReckNN maybe compromised. We analyze the impact of this version on the accuracy in Section 6.4;
- `atomic`: this version refers to the atomic instructions used to avoid the lost update problem in `Naive` and `Memoiz`, and to the lock-free insertion in `Dictionary` and `Optimized`;

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023



(a) Corel5k (b) MediaEval

Fig. 5. Overhead introduced by the different synchronization methods for each optimization.



(a) (b)

Fig. 6. Results for `MediaEval` with the `nosync` method: (a) Execution time, (b) Speedup. For each cluster, from left to right: `Naive`, `Dictionary`, `Memoiz`, and `Optimized` .

- `hle`: it uses Intel's hardware lock elision when updating the authority scores (both in the distance matrix and the dictionary versions).

In the following paragraphs, we discuss the overhead introduced by the various synchronization mechanisms, as well as their scalability. Due to space limitations, we focus on the `Corel5k` and, in particular, the `MediaEval` datasets. The results for `MPEG-7` follow the same trends.

6.3.1. Single core overhead

We start by analysing the overhead introduced by the different synchronization methods in the single core case compared with the sequential execution. The results for `Corel5k` and `MediaEval` under different optimizations are shown in Fig. 5. Notice that the overhead comes from the `AuthScore` phase, since this is the phase where the update is performed. As expected, `nosync` has virtually no overhead since no synchronization is being performed. This is not the case for `atomic` and `hle`, however. In particular, the overhead is more prominent in the `Memoiz` optimization, as the updates are executed more frequently. For the `atomic` method, the overhead can reach 20% in `Corel5k` and 17% in `MediaEval`. The situation is even worse for `hle`, with an overhead up to 66% in `Corel5k` and 50% in `MediaEval`. This initial analysis shows that there is an important price to be paid for the consistency of the updates.

6.3.2. No synchronization scalability

In this configuration, the lost update problem is allowed to happen. Fig. 6 shows the execution time (left) and speedup (right) for `MediaEval` as the number of cores is increased up to 72. For each cluster in Fig. 6a, the bars (from left to right) correspond to `Naive`, `Dictionary`, `Memoiz`, and `Optimized`. As can be seen, the execution time decreases as more cores are added. With 36 cores, the running time with `Optimized` is about 0.79s. It is important to notice that, with 36 cores and the `Balanced` policy, there is at most one thread context being used per core (hyperthreading is not used). Initially, as the number of cores is increased to 42, a slowdown is observed because of the extra overhead caused by the additional thread contexts. However, as the number of cores keeps increasing, the execution time starts to decrease again. This behavior can be easily seen in the speedup graph shown in Fig. 6b. With 72 cores, when all thread contexts are being fully utilized, the `Optimized` execution time is 0.65s, a gain of 21% when compared with the same configuration with 36 cores.

The speedup curves of Fig. 6b also show that the improvements are much more significant when going from 6 to 36 cores. As mentioned before, when going from 36 to 72 cores, more thread contexts per core are used and the performance

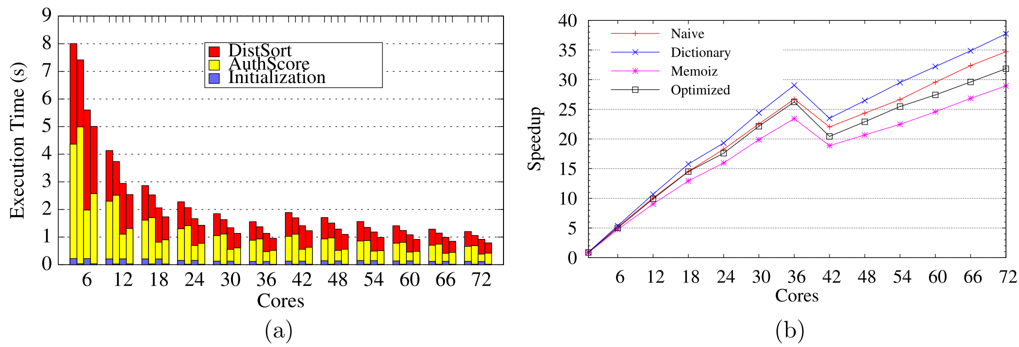


Fig. 7. Results for `MediaEval` with the `atomic` method: (a) Execution time, (b) Speedup. For each cluster, from left to right: `Naive`, `Dictionary`, `Memoiz`, and `Optimized` .

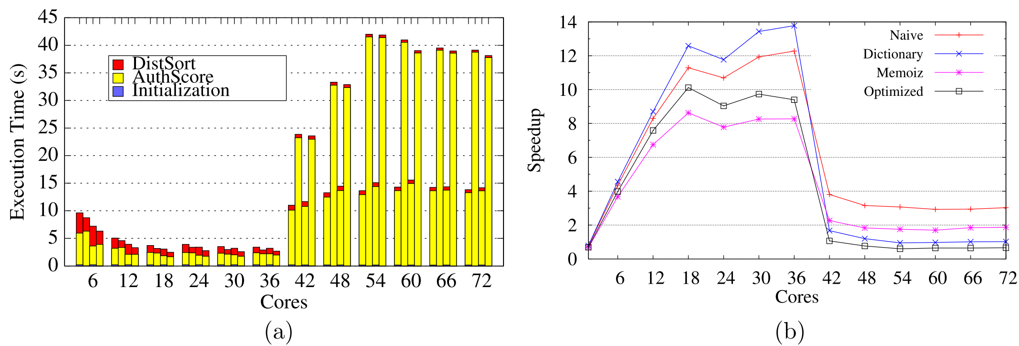


Fig. 8. Results for `MediaEval` with the `hle` method: (a) Execution time, (b) Speedup. For each cluster, from left to right: `Naive`, `Dictionary`, `Memoiz`, and `Optimized` .

does not improve at the same pace. This is because the Intel processor utilized in our experiments use a form of simultaneous multithreading (called hyperthreading in Intel's terminology) that forces multiple thread contexts to share hardware resources. For instance, the `Dictionary` optimization achieves a speedup of 32.6x with 36 cores and 42.5x with 72 cores. It is also important to notice that the speedup numbers are calculated individually for each optimization (they have different sequential execution time). Therefore, an optimization with a speedup higher than others does not imply that its respective execution time is also faster. As an example, `Dictionary` achieves a 42.5x speedup with 72 cores while `Optimized`'s speedup is 38.7x, although `Optimized` execution time is 0.65s compared to `Dictionary`'s 0.93s.

Overall, the results with inconsistent updates reveal a good scalability and performance, with the `Optimized` version providing the best execution time with 72 cores (0.65s). Also, `Optimized` was 1.69x faster than `Naive` with that core configuration (0.65s and 1.10s, respectively).

6.3.3. Atomic scalability

As with the previous figure for `nosync`, Fig. 7 shows execution time and speedup results for `atomic`. Both Fig. 7a and Fig. 7b look very similar structurally to the ones showed previously for `nosync`, but the absolute values are somewhat lower due to the increased overhead added by the atomic instructions. `Optimized` is still the best version with a running time of 0.79s with 72 cores. Compared to `nosync`'s 0.65s with the same configuration, the slowdown is 17.7%. The speedups are also less accentuated, with `Optimized` achieving 31.8x with 72 cores, against `nosync`'s 38.7x. Nonetheless, the performance still scales and the lost updated problem is solved with the atomic synchronization. Finally, `Optimized` was 1.52x faster than `Naive` with 72 cores (0.79s versus 1.20s), showing the effectiveness of the proposed optimizations.

6.3.4. HLE scalability

The last synchronization strategy considered makes use of Intel's hardware lock elision feature. For this mechanism, the best thread affinity strategy was `Compact`, as both `Balanced` and `Scatter` tend to worsen the performance very early. As illustrated by Fig. 8, Intel's HLE is very sensitive to locality: the closest the thread contexts are, the less the overhead to elide the locks. With 36 cores and the `Compact` policy, all thread contexts of a single socket are fully utilized. Therefore, the system performs reasonably well from 1 to 36 cores but, after that amount, cores from the other socket start to be utilized and the performance degrades considerably. With a high thread count, the situation is worse for `Dictionary` and `Optimized` because the region of code that is covered by the lock elision mechanism is larger (recall that a dictionary update is required).

Looking at Fig. 8b, it is possible to notice that Memoiz and Optimized do not scale after 18 cores. These versions execute update operations more frequently due to the memoization optimization. Although Naive and Dictionary still scale from 24 to 36 cores, their running time do not improve by a large margin. The best execution time for HLE was achieved with the Optimized version and 18 cores, totalizing 2.48s.

6.3.5. Summing up

For all the synchronization methods described, the Optimized version provided the best execution time with MediaEval, showing the effectiveness of the proposed parallelization technique and optimizations for datasets with large collections. Although we did not show the results for the MPEG-7 and Corel5k datasets due to space limitations, the trends are similar but with Memoiz providing the best performance since those datasets are smaller.

As expected, the best results were achieved when no synchronization is performed (nosync), although the price is a possible loss of accuracy of the classifier (quantified in Section 6.4). When atomic operations are used to synchronize the access to the distance matrix/dictionary, a slowdown of about 17.7% compared to nosync was observed with 72 cores. Unfortunately, the hardware lock elision mechanism did not scale after 36 cores and, even when only one socket is used, the slowdown is about 73.8% and 68.1% compared to nosync and atomic, respectively.

6.4. Impact of no synchronization

Previous subsections have shown that the best performance is achieved when no synchronization is used to avoid the lost update problem. Earlier works [38,45] have argued that, in similar scenarios, the problem does not affect the quality of the distance learning method and thus the solution should be accepted. However, no quantitative analysis has been performed on machines with a large number of cores to investigate whether that is really the case. In order to shed some light into that issue, Table 3 provides, for each dataset, the mean average precision (MAP) and lost updates percentage as the number of cores are increased. Except for the single core case (when all values collected by the 10 executions are exactly the same), the table shows the MAP values with the largest difference to the base MAP (the one collected with 1 core) and the maximum number of lost updates among the 10 executions, for each core configuration.

Consider, initially, the quality loss measured by MAP. The base value is given when the number of cores is one, since the lost update problem cannot occur in this case. For each dataset, MAP scores for each optimization are presented: N for Naive, D for Dictionary, M for Memoiz, and O for Optimized. Notice that, for MPEG-7, the MAP for Naive and Memoiz (93.10%) slightly differs from that of Dictionary and Optimized (93.11%), even when only 1 core is used. The reason for this mismatch is due to the different positions objects with the same weight can occupy in the ranked list after the sorting procedure. While in the distance matrix (Naive and Memoiz) this position is given by the order they appear in the matrix, the position when using a dictionary (Dictionary and Optimized) is determined by the hash function. It is important to note, however, that both solutions are semantically equivalent and the resulting value should not diverge by a large margin (which is the case here).

Table 3 also shows that the MAP values for increasing number of cores tend not to differ a lot from the base value (single core). This is particularly true for MediaEval, when the difference is only 0.01%. For MPEG-7 (93.14%) and Corel5k (35.47%), the difference is never greater than 0.04%. The results indicate that the version with no synchronization does not seem to lose accuracy compared to the base value, even with a large number of cores. In order to get further insight into this behavior, we instrumented the code so as to measure the number of times a core failed to update the distance due to another core also updating it. These values are shown in the second big column of Table 3 (Lost Updates) for the versions of *ReckNN* that use the distance matrix (Naive and Memoiz). As it can be seen, the amount of lost updates indeed increase with the number of cores, but very slowly. Notice that the amount of lost updates with memoization (M) tends to increase faster due to the fact that the updates are performed more frequently with this optimization. Even then, the number of lost updates was never larger than 1.63% of the total updates performed.

7. Conclusion

The massive volumes of multimedia data have evidenced the necessity of efficient and scalable retrieval methods. With the goal of meeting those demands, this paper proposed and analysed a series of optimizations for the Unsupervised Manifold Reciprocal kNN Graph algorithm (*ReckNN*). In particular, it was investigated the use of dictionaries to store the distances between the objects in *ReckNN*, along with the memoization of authority scores. To effectively tackle scalability, a parallelization of *ReckNN* was presented together with several synchronization methods. A thorough analysis using a modern 72-core Intel machine showed that the proposed optimizations are both efficient and scalable, with speedups in the order of 40x.

One possible limitation of the current study is that most of the optimization strategies were devised and evaluated in shared memory machines. Therefore, in future works we plan to investigate the optimizations for distributed environments (such as clouds). As future work, we also intend to investigate the possibility of applying the proposed strategies to other manifold learning methods.

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

Table 3
MAP values and lost update percentages for different core configurations.

	MAP (%)												Lost updates (%)					
	MPEG-7				Corel5k				MediaEval				MPEG-7		Corel5k		MediaEval	
Cores	N	D	M	O	N	D	M	O	N	D	M	O	N	M	N	M	N	M
1	93.10	93.11	93.10	93.11	35.51	35.51	35.51	35.51	3.89	3.89	3.89	3.89	0.00	0.00	0.00	0.00	0.00	0.00
6	93.10	93.11	93.11	93.11	35.51	35.50	35.49	35.51	3.89	3.89	3.90	3.89	0.03	0.05	0.03	0.08	0.05	0.09
12	93.12	93.11	93.12	93.11	35.50	35.52	35.48	35.49	3.90	3.89	3.90	3.89	0.08	0.18	0.07	0.17	0.11	0.21
18	93.11	93.11	93.12	93.12	35.52	35.49	35.52	35.51	3.89	3.89	3.89	3.90	0.10	0.20	0.11	0.26	0.18	0.33
24	93.11	93.12	93.10	93.10	35.48	35.49	35.52	35.48	3.89	3.89	3.90	3.89	0.28	0.45	0.12	0.29	0.22	0.43
30	93.10	93.11	93.10	93.13	35.48	35.51	35.47	35.51	3.89	3.89	3.89	3.89	0.19	0.38	0.17	0.38	0.31	0.58
36	93.11	93.12	93.12	93.10	35.50	35.49	35.48	35.49	3.89	3.89	3.89	3.89	0.18	0.35	0.21	0.47	0.37	0.72
42	93.11	93.11	93.10	93.11	35.49	35.49	35.51	35.49	3.90	3.89	3.90	3.90	0.24	0.48	0.19	0.46	0.36	0.71
48	93.10	93.12	93.14	93.11	35.48	35.49	35.51	35.52	3.90	3.89	3.90	3.90	0.36	0.74	0.20	0.54	0.40	0.76
54	93.10	93.13	93.11	93.13	35.51	35.49	35.47	35.52	3.90	3.89	3.90	3.90	0.43	0.84	0.25	0.69	0.43	0.86
60	93.12	93.12	93.13	93.12	35.51	35.49	35.51	35.51	3.89	3.89	3.90	3.89	0.41	0.90	0.30	0.80	0.48	0.92
66	93.12	93.11	93.10	93.11	35.48	35.50	35.47	35.49	3.90	3.90	3.90	3.89	0.36	0.90	0.34	0.92	0.45	0.90
72	93.10	93.12	93.11	93.10	35.48	35.51	35.50	35.52	3.89	3.89	3.90	3.89	0.74	1.63	0.38	1.01	0.56	1.06

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023

Acknowledgments

We thank the São Paulo Research Foundation- FAPESP (grants 2013/08645-0 and 2016/06441-7), CNPq (grants 446160/2014-8, 423228/2016-1, 313122/2017-2, and 308194/2017-9), and Royal Academy of Engineering(grant NRCP1516/1/33) for funding.

References

[1] J. Almeida, N.J. Leite, R.S. Torres, Comparison of video sequences with histograms of motion patterns, in: Proceedings of the IEEE International Conference on Image Processing (ICIP), 2011, pp. 3673–3676.
[2] J. Almeida, D.C.G. Pedronette, O.A.B. Penatti, Unsupervised manifold learning for video genre retrieval, in: Proceedings of the Iberoamerican Congress on Pattern Recognition (CIARP), 2014, pp. 604–612.
[3] J. Almeida, A. Rocha, R.S. Torres, S. Goldenstein, Making colors worth more than a thousand words, in: Proceedings of the ACM International Symposium on Applied Computing (ACM-SAC), 2008, pp. 1180–1186.
[4] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the AFIPS Spring Joint Computer Conference, 1967, pp. 483–485.
[5] X. Bai, S. Bai, X. Wang, Beyond diffusion process: neighbor set similarity for fast re-ranking, Inf. Sci. 325 (2015) 342–354.
[6] C. Beecks, Distance-based Similarity models for content-based multimedia retrieval, Fakultät für Mathematik, Informatik und Naturwissenschaften, RWTH Aachen University, 2013 Ph.D. thesis.
[7] C. Beecks, M.S. Uysal, T. Seidl, A comparative study of similarity measures for content-based multimedia retrieval, in: Proceedings of the IEEE International Conference on Multimedia and Expo, 2010, pp. 1552–1557.
[8] O.A.R. Board, OpenMP Application Program Interface, Version 4.0, 2013 https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.
[9] K.-w. Chau, Use of meta-heuristic techniques in rainfall-runoff modelling, Water 9 (3) (2017).
[10] J. Corbet, AutoNUMA: The Other Approach to NUMA Scheduling, 2012 https://lwn.net/Articles/488709/.
[11] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, third ed., The MIT Press, 2009.
[12] I. Corporation, Intel Architecture Instruction Set Extensions Programming Reference, Intel Corporation, 2012.
[13] L.A. Duarte, O.A.B. Penatti, J. Almeida, Bag of genres for video retrieval, in: Proceedings of the Conference on Graphics, Patterns and Images (SIBGRAPI), 2016, pp. 257–264.
[14] P. Duygulu, K. Barnard, J.F.G.d. Freitas, D.A. Forsyth, Object recognition as machine translation: learning a lexicon for a fixed image vocabulary, in: Proceedings of the 7th European Conference on Computer Vision-Part IV, ECCV, 2002, pp. 97–112.
[15] V. Gholami, K. Chau, F. Fadaee, J. Torkaman, A. Ghaffari, Modeling of groundwater level fluctuations using dendrochronology in alluvial aquifers, J. Hydrol. 529 (2015) 1060–1069.
[16] J. Huang, S.R. Kumar, M. Mitra, W.-J. Zhu, R. Zabih, Image indexing using color correlograms, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 1997, pp. 762–768.
[17] H. Jegou, C. Schmid, H. Harzallah, J. Verbeek, Accurate image search using the contextual dissimilarity measure, Proceedings of the IEEE Trans. Pattern Anal. Mach. Intell. 32 (1) (2010) 2–11.
[18] J. Jiang, B. Wang, Z. Tu, Unsupervised metric learning by self-smoothing operator, in: Proceedings of the International Conference on Computer Vision (ICCV), 2011, pp. 794–801.
[19] D. Knuth, The art of computer programming, vol. 3, sorting and searching, second ed., Addison-Wesley, 1998.
[20] A. Kuleshov, A. Bernstein, Manifold learning in data mining tasks, in: P. Perner (Ed.), Machine Learning and Data Mining in Pattern Recognition, 2014, pp. 119–133. Cham
[21] L.J. Latecki, R. Lakamper, U. Eckhardt, Shape descriptors for non-rigid shapes with a single closed contour, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2000, pp. 424–429.
[22] M.H.C. Law, A.K. Jain, Incremental nonlinear dimensionality reduction by manifold learning, IEEE Trans. Pattern Anal. Mach. Intell. 28 (3) (2006) 377–391.
[23] H. Ling, X. Yang, L.J. Latecki, Balancing deformability and discriminability for shape matching, in: Proceedings of the European Conference on Computer Vision (ECCV), 3, 2010, pp. 411–424.
[24] P. Liu, J.-M. Guo, K. Chamnongthai, H. Prasetyo, Fusion of color histogram and lbp-based features for texture image retrieval and classification, Inf. Sci. 390 (2017) 95–111.
[25] S. Liu, J. Wu, L. Feng, H. Qiao, Y. Liu, W. Luo, W. Wang, Perceptual uniform descriptor and ranking on manifold for image retrieval, Inf. Sci. 424 (2018) 235–249.
[26] K. Mehlhorn, P. Sanders, Algorithms and Data Structures, Springer, 2008.
[27] M.M. Michael, High performance dynamic lock-free hash tables and list-based sets, in: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, 2002, pp. 73–82.
[28] B. Nichols, D. Buttlar, J.P. Farrell, Pthreads Programming - A POSIX Standard for Better Multiprocessing, O'Reilly, 1996.
[29] J.P. Nielsen, S. Karlsson, A scalable lock-free hash table with open addressing, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2016, pp. 33:1–33:2.
[30] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank citation ranking: bringing order to the web., Technical Report, Stanford InfoLab, 1999. Previous number = SIDL-WP-1999-0120
[31] D.C.G. Pedronette, J. Almeida, R. da S. Torres, A scalable re-ranking method for content-based image retrieval, Inf. Sci. 265 (1) (2014) 91–104.
[32] D.C.G. Pedronette, J. Almeida, R.S. Torres, A graph-based ranked-list model for unsupervised distance learning on shape retrieval, Pattern Recognit. Lett. 83, Part 3 (2016) 357–367.
[33] D.C.G. Pedronette, F.M.F. Gonalves, I.R. Guilherme, Unsupervised manifold learning through reciprocal knn graph and connected components for image retrieval tasks, Pattern Recognit. 75 (2018) 161–174.
[34] D.C.G. Pedronette, O.A. Penatti, R.d.S. Torres, Unsupervised manifold learning using reciprocal knn graphs in image re-ranking and rank aggregation tasks, Image Vis. Comput. 32 (2) (2014) 120–130.
[35] D.C.G. Pedronette, R.T.C.O.A.B. Penatti, R. da S. Torres, A semi-supervised learning algorithm for relevance feedback and collaborative image retrieval, EURASIP J. Image Video Process. 2015 (1) (2015) 27.
[36] D.C.G. Pedronette, R.d.S. Torres, Exploiting pairwise recommendation and clustering strategies for image re-ranking, Inf. Sci. 207 (2012) 19–34.
[37] D.C.G. Pedronette, R.d.S. Torres, Image re-ranking and rank aggregation based on similarity of ranked lists, Pattern Recognit. 46 (8) (2013) 2350–2360.
[38] D.C.G. Pedronette, R.d.S. Torres, E. Borin, M. Breternitz, Efficient image re-ranking computation on GPUs, in: Proceedings of the International Symposium on Parallel and Distributed Processing (ISPA), 2012, pp. 95–102.
[39] O.A.B. Penatti, E. Valle, R.d.S. Torres, Comparative study of global color and texture descriptors for web image retrieval, J. Vis. Commun. Image Represent. 23 (2) (2012) 359–380.
[40] D. Qin, S. Gammeter, L. Bossard, T. Quack, L. van Gool, Hello neighbor: Accurate object retrieval with k-reciprocal nearest neighbors, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2011, pp. 777–784.
[41] R. Rajwar, J.R. Goodman, Speculative lock elision: Enabling highly concurrent multithreaded execution, in: Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture, 2001, pp. 294–305.

ARTICLE IN PRESS

[42] P. Sefeedpari, S. Rafiee, A. Akram, K. wing Chau, S.H. Pishgar-Komleh, Prophesying egg production based on energy consumption using multi-layered adaptive neural fuzzy inference system approach, Comput. Electron. Agric. 131 (2016) 10–19.
[43] R. Taormina, K.-W. Chau, B. Sivakumar, Neural network river forecasting through baseflow separation and binary-coded swarm optimization, J. Hydrol. 529 (2015) 1788–1797.
[44] T. Gunarathne, B. Salpitikorala, A. Chauhan,, G. Fox, Optimizing OpenCL kernels for iterative statistical algorithms on GPUs, in: Proceedings of the Second International Workshop on GPUs and Scientific Applications (GPUScA), PACT, 2011, pp. 33–44.
[45] L.P. Valem, D.C.G. Pedronette, R.S. Torres, E. Borin, J. Almeida, Effective, efficient, and scalable unsupervised distance learning in image retrieval tasks, in: Proceedings of the ACM International Conference on Multimedia Retrieval (ICMR), 2015, pp. 51–58.
[46] J. Wan, D. Wang, S.C.H. Hoi, P. Wu, J. Zhu, Y. Zhang, J. Li, Deep learning for content-based image retrieval: A comprehensive study, in: Proceedings of the ACM International Conference on Multimedia, MM, 2014, pp. 157–166.
[47] J. Wang, Y. Li, X. Bai, Y. Zhang, C. Wang, N. Tang, Learning context-sensitive similarity by shortest path propagation, Pattern Recognit. 44 (10–11) (2011) 2367–2374.
[48] Z. Xia, N.N. Xiong, A.V. Vasilakos, X. Sun, Epcbir: an efficient and privacy-preserving content-based image retrieval scheme in cloud computing, Inf. Sci. 387 (2017) 195–204.
[49] F.Z. Xiao, E. McCreath, C. Webers, Fast on-line statistical learning on a gpgpu, in: Proceedings of the 9th Australasian Symposium on Parallel and Distributed Computing (AusPDC), 2011, pp. 3–13.
[50] X. Yang, L. Prasad, L. Latecki, Affinity learning with diffusion on tensor product graph, IEEE Trans. Pattern Anal. Mach. Intell. 35 (1) (2013) 28–38.
[51] S. Zhang, K.-W. Chau, Dimension reduction using semi-supervised locally linear embedding for plant leaf classification, in: D.-S. Huang, K.-H. Jo, H.-H. Lee, H.-J. Kang, V. Bevilacqua (Eds.), Emerging Intelligent Computing Technology and Applications, Springer Berlin Heidelberg, 2009, pp. 948–955.

Please cite this article as: A. Baldassin et al., An optimized unsupervised manifold learning algorithm for manycore architectures, Information Sciences (2018), https://doi.org/10.1016/j.ins.2018.06.023