Parallel Cosine Nearest Neighbor Graph Construction

David C. Anastasiu[a,*], George Karypis[b]

[a]San José State University, One Washington Square, San José, CA, USA
[b]University of Minnesota, 200 Union Street SE, Minneapolis, MN, USA

Abstract

The nearest neighbor graph is an important structure in many data mining methods for clustering, advertising, recommender systems, and outlier detection. Constructing the graph requires computing up to n^2 similarities for a set of n objects. This high complexity has led researchers to seek approximate methods, which find many but not all of the nearest neighbors. In contrast, we leverage shared memory parallelism and recent advances in similarity joins to solve the problem exactly. Our method considers all pairs of potential neighbors but quickly filters pairs that could not be a part of the nearest neighbor graph, based on similarity upper bound estimates. The filtering is data dependent and not easily predicted, which poses load balance challenges in parallel execution. We evaluated our methods on several real-world datasets and found they work up to two orders of magnitude faster than existing methods, display linear strong scaling characteristics, and incur less than 1% load imbalance during filtering.

Keywords: similarity search, neighborhood graph construction, bounded similarity graph, cosine similarity, all-pairs, nearest neighbors, shared memory parallel

1. Introduction

Computing the nearest neighbor graph (NNG), or similarity graph, for a set of objects is a common task in many data analysis tasks, including clustering [13, 26], online advertising [35], recommender systems [17], data cleaning [7, 45], and query refinement [11, 41]. For example, effective clustering methods [47] have been devised that work by partitioning the nearest neighbor graph of a set of objects. In the recommender systems domain, item-based nearest neighbor collaborative filtering algorithms derive recommendations (e.g., books or movies) from the k most similar items to each of the user's preferred items [30]. Moreover, state-of-the-art online advertising [35] and recommender systems [15, 16, 36] methods rely on an initially computed NNG to guide the discovery of the latent factor models used for recommendation.

Often, real-world objects are depicted as vectors in a high-dimensional feature space, each dimension quantifying a relevant attribute of the object. Similarity between objects is then computed as a function of their feature vectors. In this work, we focus on objects represented as *sparse non-negative vectors* and compute the proximity between two objects as the *cosine similarity* of their vector representations. Sparse non-negative vectors have been successfully used for decades in many mining tasks. As a few examples, they are the standard way to encode document collections in preparation for search [34] or text mining [28], user ratings or purchase history in recommender systems [30], and are often used to depict the structure of chemical compounds [44].

Given a set of n objects $D = \{d_1, d_2, \ldots, d_n\}$, the NNG $G = (V, E)$ is a directed graph which consists of a vertex set V, corresponding to the objects in D, and an edge for each pair (v_i, v_j) when the ith and jth objects are neighbors. In most problems, the neighbors of interest are those with close connections, which has given rise to two important problems that we study in this article. The all-pairs similarity search (APSS) or ϵ-NNG construction problem finds, for each object in the set, all other objects with a similarity value above a certain threshold ϵ. On the other hand, the k-NNG construction problem seeks to find the k closest neighbors to each object in the set D, i.e. those objects j, $j \neq i$, with highest similarity $\text{sim}(d_i, d_j)$.

A naïve approach to construct the NNG executes $O(n^2)$ object comparisons. Despite many existing works on the subject, efficient NNG construction algorithms addressing high dimensional sparse data are still being actively researched. In two recent works [3, 4], we introduced `L2AP` and `L2Knng`, two serial methods that efficiently construct the *exact* ϵ-NNG and k-NNG, respectively, by ignoring unimportant object pair comparisons. For each object in D, our methods consider all other objects as potential neighbors. However, most objects that are not one of the desired nearest neighbors are pruned (removed from consideration) without fully computing their similarity.

The two methods share a similar filtering strategy. For a given *query* object, a potential neighbor, which we call the *candidate* object, can be pruned if an upper bound of its similarity with the query object is smaller than ϵ (for `L2AP`) or than the minimum similarity value among any of the k closest currently known neighbors of the query (for `L2Knng`). Given its reliance on minimum neighborhood similarities, as a way to boost its pruning effectiveness, `L2Knng` first identifies, for each object, k similar objects that may not be its nearest neighbors. We pro-

*Corresponding author
Email addresses: david.anastasiu@sjsu.edu (David C. Anastasiu), karypis@cs.umn.edu (George Karypis)

posed L2Knng-a[1] for this task, a fast *approximate graph construction* method that we showed achieves high recall in less time than other state-of-the-art methods [4].

Unlike our two previous works, the focus of this article is on cosine similarity ϵ-NNG and k-NNG construction in the shared memory parallel setting. The filtering performed during the construction is data dependent and not easily predicted, which poses load balance challenges in this context. Furthermore, marshaling neighborhood updates may cause contention in both the initial approximate graph construction and the filtering phases of L2Knng. We design novel shared memory parallel algorithms for the ϵ-NNG and k-NNG construction problems which use a number of cache-tiling optimizations, combined with fine-grained dynamically balanced parallel tasks, to construct graphs up to two orders of magnitude faster than existing methods. Our methods display linear strong scaling characteristics and incur less than 1% load imbalance during filtering. Specifically, our parallel ϵ-NNG construction method, pL2AP, solves the APSS problem, using 24 threads, 5.4x–231.6x faster than the best parallel baseline and 12.3x–33.9x faster than the fastest serial method on datasets with hundreds of millions of non-zeros. Using 16 threads, our approximate k-NNG construction method, pL2Knng-a, is 1.5x – 21.7x more efficient than the best approximate state-of-the-art baseline, and our exact variant, pL2Knng, achieves 3.0x – 12.9x speedup over an efficient exact baseline.

Please note that the current paper is a consolidated and extended version of two previous workshop papers we presented at the Irregular Applications: Architectures and Algorithms workshop in 2015–2016, namely [5] and [6]. The article is self-contained and significantly improves upon the initial published papers. Specifically, we have made the following enhancements: (i) we present a unified description of our two filtering-based algorithms for shared memory nearest neighbor graph construction, pointing out the filtering criteria differences between the two methods; (ii) we extended the presentation of our serial L2AP method, clarifying the filtering differences between it and a competing method, APT; (iii) we improved the presentation of our serial L2Knng method and of the query vector mask-hashing technique in pL2AP; (iv) we greatly expanded our experimental evaluation of both pL2AP and pL2Knng, adding or extending the discussion on nine different experiments; and (v) we added a discussion section that highlights strengths and challenges in our parallel cosine similarity graph construction methods.

The remainder of the paper is organized as follows. Section 2 introduces the problem and notation used throughout the paper. We start our algorithmic presentation by first giving an overview of the serial algorithms in Section 3, which will clarify the presentation of our parallel methods in Section 4. Along the way, in Section 3.2, we will also introduce some enhancements over our initial serial k-NNG construction method that led to 1.5x efficiency improvement. We describe our evaluation methodology and analyze experimental results in Sections 5 and 6. Section 7 summarizes related works, and Section 8 concludes the paper.

[1]The method is called L2KnngApprox in [4].

2. Definition & notations

We adopt a similar notation as in our earlier work [4]. Let d_i denote the ith of n objects in D, $\boldsymbol{d}_i \in \mathbb{R}^m$ denote the feature vector in m-dimensional Euclidean space associated with the ith object, and $d_{i,j}$ the value (or weight) of the jth feature of object d_i. We measure vector similarity via the *cosine function*,

$$\cos(\boldsymbol{d}_i, \boldsymbol{d}_j) = \frac{\sum_{l=1}^{m} d_{i,l} \times d_{j,l}}{||\boldsymbol{d}_i||_2 \times ||\boldsymbol{d}_j||_2}.$$

Since cosine similarity is invariant to changes in the length of vectors, we assume that all vectors have been scaled to be of unit length ($||\boldsymbol{d}_i|| = 1, \forall d_i \in D$). Given that, the cosine between two vectors $\boldsymbol{d}_i$ and $\boldsymbol{d}_j$ is simply their dot-product, which we denote by $\langle \boldsymbol{d}_\mathbf{i}, \boldsymbol{d}_\mathbf{j} \rangle$. This not only simplifies the presentation of the algorithm but also reduces the number of floating point operations needed to solve the problem at hand.

The neighborhood of an object d_i in D, denoted by Γ_{d_i}, is the set of objects in $D \setminus \{d_i\}$ whose similarity with d_i is the highest among all objects in $D \setminus \{d_i\}$. The NNG of D is a directed graph $G = (V, E)$ where vertices correspond to the objects and an edge (v_i, v_j) indicates that the jth object is in the neighborhood of the ith object. We are interested in two specific NNGs. The ϵ-NNG restricts the neighborhood of each object d_i to only those objects d_j with a similarity $\text{sim}(d_i, d_j) \geq \epsilon$. The k-NNG restricts the neighborhood of d_i to the k most similar objects to d_i. An *approximate* k-NNG is one in which the k neighbors of each vertex do not necessarily correspond to the k most similar objects.

We denote by the *minimum (neighborhood) similarity* σ_{d_i} the minimum similarity between object d_i and one of its current k neighbors. We say that a neighborhood is *improved* when its minimum similarity σ_{d_i} increases in value, and it is *complete* once all correct neighbors that belong to a neighborhood have been added to it. Given sparse vectors, it is possible that an object d_j may have less than k possible neighbors, as we ignore all null similarities and d_j may have non-zero features in common with less than k other objects in D. In this case, by convention, the σ_{d_j} value of its neighborhood is the minimum among all similarities in its neighborhood, and its neighborhood is complete.

An *inverted index* representation of D is a set of m lists, $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$, one for each feature, containing pairs $(d_i, d_{i,j})$, where d_i is an indexed object that has a non-zero value for feature j and $d_{i,j}$ is that value. The index may store additional information, such as the position of the feature in the given document or other statistics.

Given a vector $\boldsymbol{d}_q$ and a dimension j, we will denote by $\boldsymbol{d}_q^{\leq j}$ the vector obtained by keeping the j leading dimensions in $\boldsymbol{d}_q$, $(d_{q,1}, \ldots, d_{q,j}, 0, \ldots, 0)$, which we call the (inclusive) *prefix* (vector) of $\boldsymbol{d}_q$. Similarly, we refer to $\boldsymbol{d}_q^{>j} = (0, \ldots, 0, d_{q,j+1}, \ldots, d_{q,m})$ as the (exclusive) *suffix* of $\boldsymbol{d}_q$, obtained by setting the first j dimensions of $\boldsymbol{d}_q$ to 0. The exclusive

prefix $\boldsymbol{d}_q^{<j}$ and inclusive suffix $\boldsymbol{d}_q^{\geq j}$ are analogously defined. One can then verify that

$$\boldsymbol{d}_q = \boldsymbol{d}_q^{\leq j} + \boldsymbol{d}_q^{>j},$$
$$||\boldsymbol{d}_q||^2 = ||\boldsymbol{d}_q^{\leq j}||^2 + ||\boldsymbol{d}_q^{>j}||^2, \text{and}$$
$$\langle \boldsymbol{d_c}, \boldsymbol{d_q} \rangle = \langle \boldsymbol{d_c}, \boldsymbol{d_q^{\leq j}} \rangle + \langle \boldsymbol{d_c}, \boldsymbol{d_q^{>j}} \rangle.$$

Table 1 provides a summary of notation used in this work.

Table 1: Notation used throughout the work.

	Description
D	set of objects
k	size of desired neighborhoods
ϵ	minimum neighbor similarity threshold
$\boldsymbol{d}_i$	vector representing object d_i
$d_{i,j}$	value for jth feature in $\boldsymbol{d}_i$
$\boldsymbol{d}_i^{\leq l}, \boldsymbol{d}_i^{>l}$	prefix and suffix of $\boldsymbol{d}_i$ at dimension l
Γ_{d_i}	neighborhood for object d_i
σ_{d_i}	smallest similarity value in N_{d_i}
$\mathcal{N}$	set of neighborhoods
$\hat{\mathcal{N}}$	set of initial approximate neighborhoods
$\mathcal{I}$	inverted index
μ	candidate list sizes
γ	number of neighborhood enhancement updates
θ	number of objects in an inverted index tile
ζ	number of non-zeros in an inverted index tile
η	number of objects in a query tile

3. Serial algorithms

In this section, we present an overview of our serial ϵ-NNG and k-NNG construction methods, `L2AP` and `L2Knng`. In describing the methods, we will also analyze the flow of computation and data in the algorithms, which will inform our algorithmic choices for the parallel methods described in Section 4.

3.1. L2AP

Most serial APSS solutions follow a similar computation framework, first introduced by Bayardo et al. [11]. The main idea in the framework is to decompose the computation of $\mathbf{DD}^T$, which finds all pairwise similarities for objects in D, into

$$\mathbf{DD}^T = \mathbf{DA}^T + \mathbf{DB}^T,$$

where $\mathbf{D} = \mathbf{A} + \mathbf{B}$, and matrices $\mathbf{A}$ and $\mathbf{B}$ contain disjoint subsets of the non-zero values in $\mathbf{D}$. Specifically, for the ith object, $\mathbf{A}$ contains the prefix vector $\boldsymbol{d}_i^{\leq l}$ and $\mathbf{B}$ contains the suffix vector $\boldsymbol{d}_i^{>l}$ as their respective ith rows. The segmentation point l is chosen individually for each object such that all *correct* neighbor pairs, those that will be part of the exact solution, will have a non-zero dot-product after computing $\mathbf{DB}^T$. The computation of $\mathbf{DA}^T$ is then restricted to only those object pairs with non-zero values in $\mathbf{DB}^T$. Additional object pairs are *pruned* (eliminated from consideration) during both matrix product computations by relying on different similarity upper bounds that are checked against the threshold ϵ, which is an input to the problem. Figure 1 depicts a conceptual decomposition of matrix $\mathbf{D}$ into its prefix and suffix components.

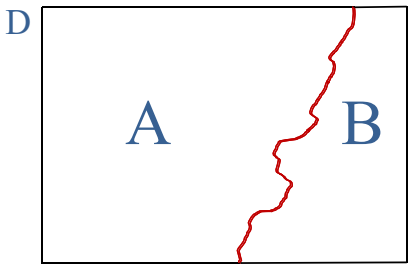


Figure 1: Decomposition of matrix $\mathbf{D}$ into its prefix and suffix sections, denoted as matrices $\mathbf{A}$ and $\mathbf{B}$.

Algorithm 1 The AllPairs Framework

1: **function** ALLPAIRS(D, ϵ)
2: Set processing order for vectors and features
3: $O \leftarrow \varnothing$, $I_j \leftarrow \varnothing$, **for** $j = 1, \ldots, m$
4: **for each** $q = 1, \ldots, n$ **do**
5: $\boldsymbol{c}_q \leftarrow$ GenerateCandidates($\boldsymbol{d}_q, \mathcal{I}, \epsilon$)
6: $O \leftarrow O \cup$ VerifyCandidates($\boldsymbol{d}_q, \boldsymbol{c}_q, \mathcal{I}, \epsilon$)
7: Index($\boldsymbol{d}_q, \mathcal{I}, \epsilon$)
8: **return** O

Algorithm 1 describes the sequential similarity search execution in the AllPairs framework. Given that cosine similarity is commutative, the framework only computes the lower triangular part of $\mathbf{DD}^T$. The algorithm incrementally finds the result by identifying each object's neighbors, one object at a time, in a given processing order. While processing an object d_q, which we call the *query*, a list of potential *candidates* is generated (line 5) by computing $\boldsymbol{c}_q = \boldsymbol{d}_q \mathbf{B}_{<i}^T$, where $\mathbf{B}_{<i}^T$ contains only rows that come before i in the processing order. The inverted index $\mathcal{I}$ is a growing compressed sparse column (CSC) representation of $\mathbf{B}_{<i}^T$. A *candidate* for the ith object is any object with a non-zero value in $\boldsymbol{c}_q$. Some of the values in $\boldsymbol{c}_q$ are expressly set to zero (candidate pruning) if a similarity estimate with that candidate is below ϵ. In the second stage, each candidate is verified (line 6) by computing, for each candidate d_c, $\langle \boldsymbol{d_q}, \boldsymbol{d_c^{\leq}} \rangle + c_{q,c}$, where $\boldsymbol{d}_c^{\leq} = \mathbf{A}(c, :)$ is the prefix of $\boldsymbol{d}_c$, those values of $\boldsymbol{d}_c$ not included in $\mathbf{B}$. Additional candidates are pruned and only those with a similarity of at least ϵ are added to the result. In the final stage (line 7), the query object is analyzed and some of its suffix features and other meta-data are added to the growing inverted index $\mathcal{I}$.

Awekar and Samatova [9] provide the only existing shared memory parallel algorithm to solve the ϵ-NNG construction problem, which we call `pAPT`. Their method is based on an existing serial APSS algorithm they developed, `APT` [8], and uses a similar filtering strategy as described in this section. In the remainder of this section, we highlight the pruning choices in `APT` and `L2AP`. Additionally, we analyze memory access patterns inherent in the computations in each stage of the framework. Table 2 provides a quick reference for these pruning choices.

3.1.1. Indexing

Since lists in the inverted index are traversed each time a search is performed for a query object, it is beneficial to index

Table 2: Similarity estimates in APT/pAPT and L2AP/pL2AP.

bound	stage	estimate	APT / pAPT	L2AP / pL2AP
idx	idx	$\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{>q})$	$\langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{mx}_{\geq \boldsymbol{q}} \rangle$	$\min(\langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{mx}_{\geq \boldsymbol{q}} \rangle, \|\boldsymbol{d}_q^{\leq j}\|_2)$
sz	c.g.	$\min(\|\boldsymbol{d}_c\|_0)$	$(\epsilon / \|\boldsymbol{d}_q\|_\infty)^2$	$(\epsilon / \|\boldsymbol{d}_q\|_\infty)^2$
rs		$\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{<q})$	$\langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{mx} \rangle$	$\min(\langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{mx} \rangle, \|\boldsymbol{d}_q^{\leq j}\|_2)$
$l2cg$		$\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_c^{\leq j})$	–	$\|\boldsymbol{d}_q^{\leq j}\|_2 \times \|\boldsymbol{d}_c^{\leq j}\|_2$
ps	c.v.	$\text{sim}(\boldsymbol{d}_q, \boldsymbol{d}_c^{\leq})$	–	$\min(\langle \boldsymbol{d}_{\mathbf{c}}^{\leq}, \boldsymbol{mx}_{\geq \mathbf{c}} \rangle, \|\boldsymbol{d}_c^{\leq}\|_2)$
dps_1		$\text{sim}(\boldsymbol{d}_q, \boldsymbol{d}_c^{\leq})$	$\min(\|\boldsymbol{d}_q\|_\infty \times \|\boldsymbol{d}_c^{\leq}\|_1, \|\boldsymbol{d}_q\|_1 \times \|\boldsymbol{d}_c^{\leq}\|_\infty)$	$\min(\|\boldsymbol{d}_q\|_0, \|\boldsymbol{d}_c^{\leq}\|_0) \times \|\boldsymbol{d}_q\|_\infty \times \|\boldsymbol{d}_c^{\leq}\|_\infty$
dps_2		$\text{sim}(\boldsymbol{d}_q, \boldsymbol{d}_c^{\leq})$	–	$\min(\|\boldsymbol{d}_q\|_0, \|\boldsymbol{d}_c^{\leq l}\|_0) \times \|\boldsymbol{d}_q^{\leq l}\|_\infty \times \|\boldsymbol{d}_c^{\leq l}\|_\infty$
$l2cv$		$\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_c^{\leq j})$	–	$\|\boldsymbol{d}_q^{\leq j}\|_2 \times \|\boldsymbol{d}_c^{\leq j}\|_2$

The vectors $\boldsymbol{d}_q$ and $\boldsymbol{d}_c$ represent the query and candidate objects, respectively. Prefix and suffix vectors are defined in Section 2. The prefix vector $||\boldsymbol{d}_c^{\leq}||$ is the un-indexed portion of the candidate. The vector $\boldsymbol{mx}$ represents the max vector, containing the maximum value for each feature in the dataset. Features in the max vector $\boldsymbol{mx}_{\geq \mathbf{q}}$ are also upper-bounded by $||\boldsymbol{d}_q||_\infty$. The feature j represents a non-zero feature in the query and/or the candidate. Here, the feature l is the last un-indexed candidate feature in the feature processing order that the query also has in common.

as few values as possible. Indexing is delayed in the framework until the similarity estimate of the query prefix with *any unprocessed object* reaches the threshold ϵ (line 3 in Algorithm 2). Any unprocessed *similar object*, one with a similarity of at least ϵ with the query, is guaranteed in this way to have at least one feature in common with the query object. Then, when that similar object is processed, the query object will be found while traversing the index.

Algorithm 2 Indexing in the AllPairs Framework

```
1: function INDEX(d_q, I, ε)
2:     for each j = 1, ..., m, s.t. d_{q,j} > 0 do
3:         if sim(d_q^{≤j}, d_{>q}) ≥ ε then                ▷ idx bound
4:             I_j ← I_j ∪ {(d_q, d_{q,j})}                 ▷ add suffix to index
```

While improving computation efficiency by limiting the number of non-zeros traversed when identifying neighbors for a query object, the *partial indexing* of only suffix values in each query object is also an effective pruning strategy. Note that some objects may not have any features in common with the query suffix. These objects are automatically removed from consideration, without even starting to compare them to the query. Figure 2 shows an example for indexing a query object d_q in our framework. In the figure, shaded cells represent non-zero values, and the index j is chosen such that the similarity of the prefix $\boldsymbol{d}_q^{\leq j}$ with any unprocessed object is lower than ϵ. Let d_{q+1}, d_{q+2}, and d_{q+3} be three such unprocessed objects. When d_q is considered as a potential neighbor for these objects, our method initially explicitly computes only the suffix dot-product $\langle \boldsymbol{d}_{\mathbf{q}}^{>\mathbf{j}}, \boldsymbol{d}_{\mathbf{c}}^{>\mathbf{j}} \rangle$, and estimates the prefix dot-product $\langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{d}_{\mathbf{c}}^{\leq \mathbf{j}} \rangle$ only for those non-zero suffix dot-products. As such, while d_{q+3} and d_q have feature f_1 in common, d_q will never be considered as a potential neighbor for d_{q+3}.

APT computes the prefix similarity estimate $\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{>q})$, which we call the idx bound, as the dot product between the query vector and the *max vector*, the vector made up of all maximum feature values in the dataset, denoted as $\boldsymbol{mx}$. Note that,

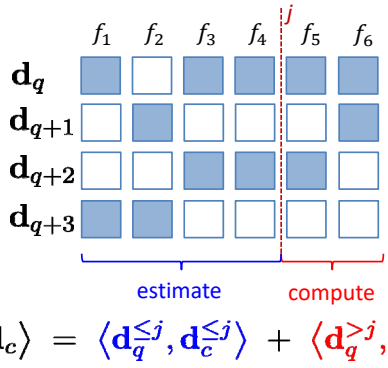


Figure 2: Example partial indexing.

if the dot product between the query prefix and the maximum vector is below the threshold ϵ, the query can only be a neighbor of one of the remaining unprocessed objects if they have at least one feature in common in the query suffix, which has already been indexed. This upper bound similarity estimate is improved by processing objects in non-increasing order of their maximum feature weights ($\|\boldsymbol{d}_i\|_\infty \geq \|\boldsymbol{d}_j\|_\infty, \forall\ i < j$), and then bounding the max vector by the maximum feature weight in the query,

$$\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{>q})_{\texttt{APT}} \leq \langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{mx}_{\geq \boldsymbol{q}} \rangle, \text{ where,}$$

$$\boldsymbol{mx}_{\geq \mathbf{q}} = \langle \min(mx_1, ||\boldsymbol{d}_q||_\infty), \ldots, \min(mx_m, ||\boldsymbol{d}_q||_\infty) \rangle.$$

In addition, L2AP uses the ℓ^2-norm of the query prefix ending at index j, inclusive, $||d_q^{\leq j}||$, as an estimate of the query object similarity with any other object, which includes unprocessed objects,

$$\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{>q})_{\texttt{L2AP}} \leq \min(\langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{mx}_{\geq \boldsymbol{q}} \rangle, ||\boldsymbol{d}_q^{\leq j}||_2).$$

Assuming an unprocessed candidate object does not have non-zero values for any of the query suffix features, then their prefix norm $\|\boldsymbol{d}_c^{\leq j}\|_2 = 1$. However, since by construction $\|\boldsymbol{d}_q^{\leq j}\|_2 <$

ϵ, leveraging the Cauchy-Schwarz inequality, $\langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{d}_{\mathbf{c}}^{\leq \mathbf{j}} \rangle < \epsilon$ and the candidate need not be considered. By constructing a *partial index* (adding only the suffix of each vector to the index), our method automatically ignores these objects.

When indexing each query suffix non-zero value (line 4 in Algorithm 2), L2AP also indexes additional meta-data, such as the ℓ^2-norm of the query prefix and its maximum value, which are used in future pruning. The similarity estimate of the unindexed query prefix with unprocessed objects is also stored, to be used during candidate verification as an effective pruning strategy for false positive candidates.

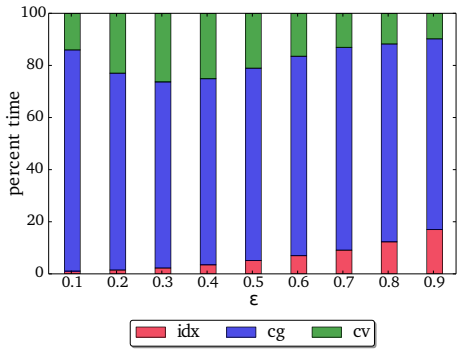


Figure 3: Percent execution times for the WW500 dataset. The stacked bars show the percent of search time taken by the indexing (*idx*), candidate generation (*cg*), and candidate verification (*cv*) phases in L2AP, for similarity thresholds ranging from 0.1 to 0.9.

Data access analysis. Indexing requires traversing the sparse query vector and accessing values in the max vector, which are stored as a dense array. Since this process occurs only once for each object in the set, it takes much less of the overall search time than the other two stages in the framework. As an example, Figure 3 shows the percent of overall search time taken by each of the three stages in L2AP, for ϵ ranging from 0.1 to 0.9 and the WW500 dataset (see Section 5 for dataset details). Furthermore, values in both the query vector and feature maximum values are accessed sequentially, in sorted feature processing order, and can take advantage of software and hardware prefetching to reduce latency. As a result, we will focus on optimizing the other two stages in the framework. It is important to note, however, that the size of the inverted index is highly dependent on the similarity threshold ϵ. As shown in Figure 1 of [3], higher thresholds allow delaying indexing further and lead to a smaller inverted index, which can lead to more potential candidates being automatically pruned.

3.1.2. Candidate generation

During the candidate generation stage of the framework, which is described in Algorithm 3, the lists in the current version of the inverted index associated with non-zero feature values in the query object are scanned, one list at a time. An accumulator (map based data structure that accumulates values

Algorithm 3 Candidate Generation in the AllPairs Framework

1: **function** GENERATECANDIDATES($\boldsymbol{d}_q, \mathcal{I}, \epsilon$)
2: $\quad \boldsymbol{c}_q \leftarrow \varnothing$ ▷ accumulator
3: $\quad$ **for each** $j = 1, \ldots, m$, s.t. $d_{q,j} > 0$ **do**
4: $\quad\quad$ **for each** $(d_c, d_{c,j}) \in I_j$ **do**
5: $\quad\quad\quad$ *check whether to prune* d_c ▷ sz bound
6: $\quad\quad\quad$ **if** $c_{q,c} > 0$ **or** d_c *is a new candidate with* $\text{sim}(d_c, d_q)$ *estimated at least* ϵ **then** ▷ rs bound
7: $\quad\quad\quad\quad c_{q,c} \leftarrow c_{q,c} + d_{q,j} \times d_{c,j}$
8: $\quad\quad\quad\quad$ *check whether to prune* d_c ▷ $l2cg$ bound
9: **return** $\boldsymbol{c}_q$

for given keys) is used to keep track of partial dot-products between the query and encountered objects. Once accumulation has started for an object, it becomes a *candidate*. Figure 4 depicts the use of an accumulator data structure during candidate generation for a query d_3 in an example dataset. In the figure, cells with solid background represent non-zero values. Our algorithm traverses only the inverted index lists for features 1, 2, and 5, which have non-zero values in d_3. Within those lists, which are sorted in increasing object processing ID order, only non-zeros for documents with a smaller ID are accumulated, depicted by the red line bisecting each list. The right-side of the figure shows the multiply-add operations that are executed and the accumulator structure (here depicted as a simple list).

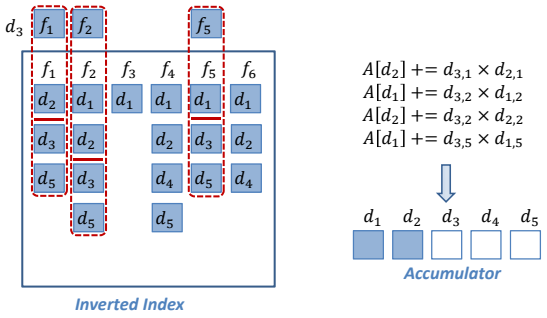


Figure 4: Use of accumulator data structure during candidate generation.

Accumulation is prevented for a new object in two additional ways. First, the size of the candidate vector (number of non-zeros) is checked against a minimum size estimate, which we call the *size* (sz) bound, and candidates with too few non-zeros are ignored. Both APT and L2AP use the same bound in this step[2]. Second, no *new candidates* are accepted if the query prefix does not have enough weight to achieve at least ϵ similarity with an indexed object. Index lists are traversed in inverse feature processing order, and the similarity estimate $\text{sim}(d_c, d_q)$ in line 6 of Algorithm 3 is approximated as the similarity of the query prefix with any indexed object, $\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{<q})$, which we call the *remaining similarity* (rs) bound. In APT, the approximation is based on computing the similarity of the query with

[2]Note that [3] uses a different sz bound, $\epsilon / (||\boldsymbol{d}_q||_\infty \times ||\boldsymbol{d}_c||_\infty)$, and erroneously states it is superior to $(\epsilon / ||\boldsymbol{d}_q||_\infty)^2$. We found both bounds provide limited benefit for different values of ϵ, and chose to use the same bound as APT in this work to simplify comparison.

the max vector, while L2AP additionally bounds it by the prefix ℓ^2-norm of the query,

$$\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{<q})_{\texttt{APT}} \leq \left\langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{mx} \right\rangle,$$
$$\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{<q})_{\texttt{L2AP}} \leq \min(\left\langle \boldsymbol{d}_{\mathbf{q}}^{\leq \mathbf{j}}, \boldsymbol{mx} \right\rangle, ||\boldsymbol{d}_q^{\leq j}||_2).$$

While accumulating partial dot-products with candidates, at each feature they have in common with the query, L2AP also checks an additional bound, $l2cg$, based on estimating the prefix similarity up to that feature, leveraging the Cauchy-Schwarz inequality, as

$$\text{sim}(\boldsymbol{d}_q^{<j}, \boldsymbol{d}_c^{<j}) \leq ||\boldsymbol{d}_q^{<j}||_2 \times ||\boldsymbol{d}_c^{<j}||_2.$$

Data access analysis. The critical memory access portions of the candidate generation stage are updating values in the accumulator data structure, which can be reused for each query, and traversing index lists. If these structures take up more than the available cache memory, the computation will be delayed while data is loaded from main memory.

Due to the predefined object processing order, objects that do not meet the minimum size requirement when traversing the index will also not meet the requirement for future query objects and can be removed from the index. Removing objects from the index is a costly operation, and APT instead updates inverted list start pointers, effectively removing objects from the start of the list until an object of adequate size is found. These objects will not need to be traversed in future iterations and can speed up computation. Experiments in [3] showed this technique had limited benefit and L2AP does not use it.

3.1.3. Candidate verification

Algorithm 4 Candidate Verification in the AllPairs Framework

```
1: function VERIFYCANDIDATES(d_q, c_q, I, ε)
2:     for each d_c s.t. c_{q,c} > 0 do
3:         check whether to prune d_c                    ▷ ps and dps_1 bounds
4:         Find highest j s.t. d^≤_{c,j} > 0 ∧ d_{q,j} > 0
5:         check whether to prune d_c                    ▷ dps_2 bound
6:         for each j s.t. d^≤_{c,j} > 0 ∧ d_{q,j} > 0 do
7:             c_{q,c} ← c_{q,c} + d_{q,j} × d_{c,j}
8:             check whether to prune d_c                ▷ l2cv bound
9:         store similarity if c_{q,c} ≥ ε
```

Candidate verification iterates through the list of candidates and computes the partial similarity between the query vector and the un-indexed portion of each candidate, adding it to the already accumulated similarity (line 7 in Algorithm 4). Each candidate is first vetted based on an upper bound of its un-indexed prefix similarity with any object stored during indexing. APT uses the Hölder inequality to derive this bound, which we name dps_1, as

$$\text{sim}(\boldsymbol{d}_q, \boldsymbol{d}_c^{\leq})_{\texttt{APT}} \leq \min(||\boldsymbol{d}_q||_\infty \times ||\boldsymbol{d}_c^{\leq}||_1, ||\boldsymbol{d}_q||_1 \times ||\boldsymbol{d}_c^{\leq}||_\infty).$$

L2AP uses several different estimates here. First, since the query follows the candidate in processing order, the similarity $\text{sim}(\boldsymbol{d}_q, \boldsymbol{d}_c^{\leq})$ can be approximated as the similarity $\text{sim}(\boldsymbol{d}_c^{\leq}, \boldsymbol{d}_{>c})$, which was computed and stored while indexing $\boldsymbol{d}_c$, and is equivalent to

$$\text{sim}(\boldsymbol{d}_q, \boldsymbol{d}_c^{\leq})_{\texttt{L2AP}} \leq \min(\left\langle \boldsymbol{d}_{\mathbf{c}}^{\leq}, \boldsymbol{mx}_{\geq \mathbf{c}} \right\rangle, ||\boldsymbol{d}_c^{\leq}||_2).$$

We call this bound ps. Second, L2AP uses a different dps_1 bound that, while theoretically inferior to the one in APT with regards to candidate pruning, was slightly more efficient to compute in experiments on a wide range of datasets in [3],

$$\text{sim}(\boldsymbol{d}_q, \boldsymbol{d}_c^{\leq})_{\texttt{L2AP}} \leq \min(||\boldsymbol{d}_q||_0, ||\boldsymbol{d}_c^{\leq}||_0) \times ||\boldsymbol{d}_q||_\infty \times ||\boldsymbol{d}_c^{\leq}||_\infty.$$

Third, after finding the last un-indexed candidate feature l in the feature processing order that the query also has in common, L2AP checks a tighter version of the dps_1 bound, which we call dps_2,

$$\text{sim}(\boldsymbol{d}_q, \boldsymbol{d}_c^{\leq})_{\texttt{L2AP}} \leq \min(||\boldsymbol{d}_q||_0, ||\boldsymbol{d}_c^{\leq l}||_0) \times ||\boldsymbol{d}_q^{\leq l}||_\infty \times ||\boldsymbol{d}_c^{\leq l}||_\infty.$$

Finally, while computing the prefix dot-product, at each common feature, L2AP checks the Cauchy-Schwarz inequality based estimate, which here we call $l2cv$,

$$\text{sim}(\boldsymbol{d}_q^{<j}, \boldsymbol{d}_c^{<j}) \leq ||\boldsymbol{d}_q^{<j}||_2 \times ||\boldsymbol{d}_c^{<j}||_2.$$

Data access analysis. The accumulator is not critical in the candidate verification stage, as processing occurs for one candidate at a time. The partial accumulated similarity of a candidate can be looked up once and further accumulation can occur on the stack. On the other hand, feature values and meta-data associated with those features in the query vector are accessed in a random fashion, based on the features encountered in the candidate object. To facilitate computing dot products between the query and candidate vectors, we have found it beneficial to insert the feature values of the query vector, its prefix ℓ^2-norm values, and its prefix maximum values in a hash table. When iterating through the sparse version of a candidate object's un-indexed prefix, the query feature, prefix maximum and ℓ^2-norm values can then be quickly looked up in $O(1)$ time. The cost of using a hash table can be offset by reusing the structure for verifying many candidates. An alternative to looking up query values in a hash table would be to traverse the candidate and query vectors concurrently, assuming a predefined global feature traversal order. We have found that, in most cases (other than datasets with small number of vector non-zeros), this strategy leads to 2x-3x slower execution times.

3.2. L2Knng

Our k-NNG construction method, L2Knng, relies on similar filtering as discussed in Section 3.1. However, since the method does not have a global minimum similarity threshold ϵ as input, it cannot use the same upper bound similarity estimates and processing order as L2AP. L2Knng instead relies on minimum similarity values in each of the object neighborhoods as bounding thresholds for the filtering framework. Due to this fact, unlike L2AP, which processes each neighborhood independently, L2Knng must keep track of n k-neighborhoods throughout its execution. Updating these neighborhoods can become a source

of thread contention in the shared memory parallel setting. In this section, we will give an overview of L2Knng, paying close attention to data access patterns and potential contention.

L2Knng execution consists of two phases. First, in the *approximate graph construction* phase, L2Knng finds an initial k neighbors for each of the objects in D by calling our approximate graph construction method, L2Knng-a. The minimum neighborhood similarities in each of the neighborhoods of the approximate graph are then used as pruning thresholds in the *filtering* phase, which outputs the *exact* nearest neighbor graph. L2Knng-a constructs the approximate graph in two steps. First, in the *initial graph construction* (IC) step, neighbors that are more likely to be in the exact k-NNG are chosen based on shared features with high weight. Then, a number of *graph enhancement* (GE) steps are executed which attempt to improve the quality of the neighborhoods by finding closer neighbors among the neighbors of the current neighbors. Algorithm 5 gives an overview of this process.

Algorithm 5 The L2Knng Algorithm

```
1: function L2KNNG(D, k, γ, μ)
2:     N̂ ← IC(D, k, μ)                          ▷ Begin L2Knng-a
3:     for each i = 1, 2, ..., γ do
4:         N̂ ← GE(D, k, μ, N̂)                   ▷ End L2Knng-a
5:     N ← Filter(D, k, N̂)
6: return N
```

Our serial improvements in L2Knng focused on the approximate graph construction phase of the method. At a very high level, each of the steps in the L2Knng-a execution is composed of the following tasks, which are shown in Algorithms 6 and 7 and will be detailed later in the discussion. Input data or the current neighborhoods are sorted and indexed to facilitate the selection of neighbor candidates (*srt*). Then, for each query object, a candidate list of potential neighbors is selected (*sel*) that may improve the current neighborhood. Data associated with the query object is optionally entered into a data structure that can facilitate fast dot-product computations or pruning (*ins*). Then, dot-products are computed between the query and each of the chosen candidates (*sim*), skipping some of the candidates whose similarity has already been previously computed. Finally, some of the neighborhoods are updated (*upd*) with computed similarities that improve them.

Algorithm 6 Initial graph construction in L2Knng-a

```
1: function IC(D, k, μ)
2:     Create inverted index of D                          ▷ srt
3:     Sort vectors in D and inverted index lists          ▷ srt
4:     for each i = 1, 2, ..., |D| do
5:         Choose μ candidates for the ith object          ▷ sel
6:         Hash the ith object                             ▷ ins
7:         Compute similarities of d_i with all μ candidates ▷ sim
8:         Update Γ_i and candidate neighborhoods          ▷ upd
9:     N̂ = ⋃ Γ_i
10: return N̂
```

In an effort to gauge where the algorithm spends most of its time, we instrumented the L2Knng-a code with timers for each

Algorithm 7 Graph enhancement in L2Knng-a

```
1: function GE(D, k, μ)
2:     Create N, sparse matrix version of N̂            ▷ srt
3:     Create inverted index of N                      ▷ srt
4:     Sort vectors and inverted lists in N            ▷ srt
5:     for each i = 1, 2, ..., |D| do
6:         Choose μ candidates for the ith object      ▷ sel
7:         Hash the ith object                         ▷ ins
8:         Compute similarities of d_i with all μ candidates ▷ sim
9:         Update Γ_i and candidate neighborhoods      ▷ upd
10:    N̂ = ⋃ Γ_i
11: return N̂
```

Table 3: Percent of the computation time for different sections of the approximate graph construction.

initial construction

dataset	k	*sort*	*sel*	*ins*	*sim*	*upd*	*perc*
RCV1	10	3.17	5.57	0.16	**88.04**	3.07	78
RCV1	100	4.44	5.70	0.26	**80.30**	9.30	39
RCV1	500	1.11	5.27	0.06	**83.48**	10.07	57
WW500	10	**24.07**	0.94	1.15	**73.06**	0.78	69
WW500	100	7.92	0.91	0.31	**89.57**	1.29	52
WW500	500	2.46	0.82	0.10	**94.77**	1.84	53

graph enhancement

dataset	k	*sort*	*sel*	*ins*	*sim*	*upd*	*perc*
RCV1	10	1.74	**20.59**	3.05	**69.54**	5.08	22
RCV1	100	2.65	**20.98**	0.26	**72.29**	3.82	61
RCV1	500	3.03	**26.84**	0.06	**66.64**	3.42	43
WW500	10	0.27	3.97	5.01	**89.52**	1.24	31
WW500	100	0.37	2.38	0.33	**96.25**	0.67	48
WW500	500	0.59	2.44	0.11	**96.03**	0.84	47

The table shows, for the initial graph construction and neighborhood enhancement phases of the L2Knng-a method, the percent of execution time of different tasks within each phase discussed in Section 3.2. The *perc* column shows the percent of the overall L2Knng-a execution taken up by the current phase of the algorithm. For each experiment, tasks taking up a significant portion of the execution time are highlighted in bold.

of the tasks. Table 3 shows the percent of the overall execution time in each phase taken by each of the tasks in the initial construction and graph enhancement phases, when searching for 10, 100, and 500 nearest neighbors in two datasets described in Section 5. In each of the experiments, we only executed one round of neighborhood enhancements ($\gamma = 1$) and chose candidate list sizes (μ) that would lead to average recall of at least 95%, i.e., L2Knng-a finds most of the nearest neighbors for each object. The last column in the table (*perc*) shows the percent of the overall L2Knng-a execution taken up by the current phase (IC or GE) of the algorithm. The results of this experiment show that L2Knng-a spends the majority of its execution time selecting candidates and computing similarities between query and candidate objects. Indexing and sorting can also account for a significant portion of the execution time when k is small. While graph enhancement takes up less time for small values of k, it accounts for almost half of the overall execution for larger k values.

Given these observations, we focused our efforts to improve L2Knng-a on the similarity computation, sorting, and candidate selection tasks. In the following sections we will detail each of

the L2Knng-a tasks and our proposed improvements.

3.2.1. Index and sort

L2Knng-a chooses candidates in the IC phase by matching objects with common high weight features. To facilitate this search, it sorts the entries in each object vector and in each inverted index list in non-increasing weight order. Then, it selects candidates for a query object by iterating through the inverted index lists associated with its highest weight features.

Since only μ candidates are selected for each query object, it is not necessary to fully sort all entries of the object vectors and inverted lists. With high probability, each inverted list will contain more than two entries (one entry will be associated with the query object). Thus, as an enhancement to L2Knng-a, we propose sorting only the top-μ values in each vector and inverted list. For each vector and inverted list with lengths greater than μ, we first apply a select procedure [27], which partitions the list such that the leading μ values are greater or equal to the remaining values, and then sort only the leading μ values. This improvement reduces the complexity of sorting a list from $O(l \log l)$, where l is the size of the list, to $O(l + \mu \log \mu)$, and can be beneficial when μ is small or for datasets with very long vectors or inverted lists.

In each GE phase, L2Knng-a chooses candidates by matching neighbors and neighbors' neighbors with high similarity values. It first creates a sparse matrix version of the current approximate neighborhood graph, $\mathbf{N}$, such that the ith row of $\mathbf{N}$ corresponds to the k-neighborhood of the ith object. It sorts the entries in each row of $\mathbf{N}$ in non-increasing value order and selects candidates for a query object by iterating through rows in $\mathbf{N}$ associated with those objects that are the closest neighbors of the query, i.e., the column IDs of the leading entries in the query neighborhood row. For those query objects with less than μ candidates selected through this process, L2Knng-a further iterates through neighborhoods of objects that have the query object as their neighbor, in non-increasing order of their similarity with the query. We call this process reverse candidate selection. To facilitate this search, L2Knng-a creates an inverted index for $\mathbf{N}$ and sorts the inverted lists in the index in non-increasing value order. In our experiments, we have found *reverse candidate selection* rarely improves effectiveness and can often degrade GE efficiency. In pL2Knng, we skip this optimization, do not create an inverted index for $\mathbf{N}$, and only sort its row entries.

3.2.2. Candidate selection

In the IC phase, L2Knng-a selects candidates by iterating through two inverted lists at a time associated with the highest values in the query vector. Algorithm 8 describes this procedure. The function *nextList* provides the inverted list associated with the next non-increasing value in $\boldsymbol{q}$. The function *nextCand* provides the next candidate in the chosen list, skipping the query object and any other objects that have already been selected. L2Knng-a uses an accumulation data structure to both track whether an object has already been selected as a candidate and to compute its partial dot-product with the query, denoted as $\langle \boldsymbol{q}, \boldsymbol{a}^{\leq} \rangle$ in Algorithm 8. Here, $\boldsymbol{a}^{\leq}$ is the prefix of $\boldsymbol{a}$ up to and including the feature associated with the index list A. Given two potential candidates c_a and c_b, L2Knng-a chooses c_a only if its partial dot-product with the query considering features already processed is greater than that of c_b.

Algorithm 8 Candidate selection in the IC phase of L2Knng-a

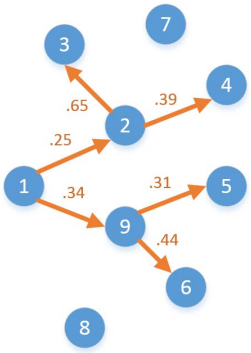
```
1: function SELECTCANDIDATESIC(D, q, μ)
2:     A ← nextList(q), B ← nextList(q), C = ∅
3:     while |C| < μ and A ≠ ∅ and B ≠ ∅ do
4:         if A = ∅ or B = ∅ then
5:             Choose candidates only from the remaining list
6:         a ← nextCand(A), b ← nextCand(B)
7:         if ⟨q, a^≤⟩ > ⟨q, b^≤⟩ then
8:             C ← C ∪ a
9:             A ← A \ a
10:            A ← nextList(q) if A = ∅
11:        else
12:            C ← C ∪ b
13:            B ← B \ b
14:            B ← nextList(q) if B = ∅
15:    end while
16: return C
```

We have improved candidate selection in the IC phase of L2Knng-a by simplifying the candidate choice condition (line 7 of Algorithm 8) to $d_{q,f(A)} \times d_{a,f(A)} < d_{q,f(B)} \times d_{b,f(B)}$, where $f(A)$ is the feature ID of inverted list A. This simplification keeps the original intent in the selection and has not shown decreased effectiveness in experiments. Instead, the efficiency of this step is increased by removing the need to compute partial dot-products. Furthermore, we use a bitvector data structure to track candidates that have already been selected, which uses less cache memory and may also help increase performance.

The GE phase selects candidates by iterating through neighbors' neighborhoods, selecting the neighbor a with the next smaller similarity value in the query's neighborhood. The neighbors of a are then visited in non-increasing similarity value order. While iterating through these neighbors, candidates are only accepted if their similarity value is greater than the similarity between a and the query. We have not made changes to the selection process in this phase of L2Knng-a. Figure 5 shows, as an example, the process of selecting candidates for an object d_1 during the GE phase of our method, given $k = 2$. In the figure, we only show relevant edges for d_1 and its neighbors. Edge weights represent object similarities. Given a candidate list size $\mu = 3$, L2Knng-a first follows the edge towards d_9 and selects its neighbors as candidates, in non-increasing order of edge weights; then, L2Knng-a follows the edge toward d_2 and adds its highest weight neighbor to the candidate list.

3.2.3. Query insertion and similarity computation

Since L2Knng-a computes the similarity of a query vector with many (namely, μ) different candidate vectors, it creates a dense version of the query vector, inserting its values into an array of size m. Each dot-product can then be computed as a sparse-dense vector dot-product, by iterating through the non-zero values of the candidate vector and looking up values of



$$C_{\mu=3}(d_1) = [d_6, d_3, d_4]$$

Figure 5: Graph enhancement example.

the query vector in the array. Given a vector $\boldsymbol{q}$ representing the dense version of $\boldsymbol{d}_q$, the dot-product $\langle \boldsymbol{q}, \boldsymbol{d_c} \rangle$ can be computed as,

$$\textbf{for each } j = 1, \ldots m \textbf{ s.t. } d_{c,j} > 0 \textbf{ do}$$
$$s \leftarrow s + d_{c,j} \times q_j$$

As computing dot-products takes up the most time in the L2Knng-a execution, we tried several other strategies for executing this operation, including (1) packing the larger of the two sparse vectors into the space of the smaller vector, trying to take advantage of vectorization capabilities of modern hardware. (2) computing sparse-sparse vector dot-products, and (3) the query vector mask-hashing technique described in [4]. In our experiments, none of the new dot-product computation strategies lead to improved performance under a wide range of execution parameters in the shared-memory parallel setting.

3.2.4. L2Knng filtering

After constructing the initial approximate NNG through L2Knng-a, our exact method uses the filtering framework presented in Section 3.1 to improve each object neighborhood until completion. The object processing order in L2AP, which was chosen to enhance some of the similarity bounds used in index construction and filtering, will not be appropriate for L2Knng. Instead, L2Knng processes objects in non-non-increasing minimum neighborhood similarity order. Moreover, the filtering bounds used in L2Knng do not depend on the maximum value in each object, which allows L2Knng to dynamically change the object processing order. In pL2Knng, we followed the same overall filtering strategy. The interested reader can find further details in [4].

4. Parallel algorithms

We now present parallel solutions to the ϵ-NNG and k-NNG problems. First, we summarize algorithmic choices in the method of Awekar and Samatova, pAPT. We then introduce pL2AP, which was designed based on the memory access observations we made in Section 3.1, with the goal of improving cache locality during similarity search. Finally, we present pL2Knng, our parallel k-NNG construction method, which is based on observations detailed in Section 3.2.

4.1. pAPT

Awekar and Samatova introduced the first shared memory parallel APSS algorithm [9], pAPT, based on their serial APT algorithm, which we describe in Algorithm 9. Their main idea was to pre-compute the partial inverted index (lines 4–5), rather than indexing each object after its processing, and allow threads to share the index structure. To prevent synchronization overheads when removing values associated with short vectors from the inverted index (line 5 of Algorithm 3), pAPT duplicates, for each thread, a list of offsets from the beginning of each inverted list. Then, each thread modifies its own offsets, incrementing them to remove only items at the start of inverted lists.

Algorithm 9 The pAPT Algorithm

```
1: function PAPT(D, ε)
2:     Set processing order for vectors and/or features
3:     O ← ∅, I_j ← ∅, for j = 1, ..., m
4:     for each q = 1, ..., n do
5:         Index(d_q, I, ε)
6:     for each q = 1, ..., n, in parallel do
7:         c_q ← GenerateCandidates(d_q, I, ε)
8:         O ← O ∪ VerifyCandidates(d_q, c_q, I, ε)
9: return O
```

Awekar and Samatova proposed three load balancing strategies in pAPT: block, round-robin, and dynamic partitioning. The object processing order in the filtering framework, namely in non-increasing maximum value order, after first normalizing object vectors, means that objects with few non-zeros are processed first, and those with many non-zeros last. As a result, statically assigning n/nt consecutive objects to each thread, where nt is the number of threads, leads to load imbalance. Awekar and Samatova attempted to fix the potential imbalance by assigning subsets of query objects with equal number of non-zeros to each thread, but found this strategy is still worse than round-robin or dynamic partitioning. The best performing load balancing strategy in their experiments was dynamic partitioning, which assigns a small set of objects to a thread as soon as it has finished processing its previously assigned set.

4.2. pL2AP

Our new method, pL2AP, uses the same indexing, candidate generation and verification pruning choices as L2AP. Additionally, pL2AP employs two strategies aimed at improving cache locality during search. First, cache-tiling breaks up the inverted index into blocks that can fit in the system cache, reducing latency during candidate generation. Second, for datasets with high dimensionality, mask-based hash tables can greatly reduce the amount of memory required for storing query object values and meta-data during search, allowing them to persist in the cache during candidate verification. Algorithm 10 provides an overview of our method.

Algorithm 10 The pL2AP Algorithm

1: **function** pL2AP($D, \epsilon, h, \zeta, \eta$)
2: Set processing order for vectors and features
3: **for each** $q = 1, \ldots, n$ **in parallel do**
4: $S \leftarrow$ FindIndexSplit($\boldsymbol{d}_q, \epsilon$)
5: $K \leftarrow$ FindIndexAssignments(S, ζ)
6: $O \leftarrow \varnothing,\ I_{k,j} \leftarrow \varnothing,$ **for** $j = 1, \ldots, m$ **and** $k = 1, \ldots, K$
7: **for each** $q = 1, \ldots, n$ **do**
8: Index($\boldsymbol{d}_q, \mathcal{I}, S, \epsilon$)
9: **for each** $k = 1, \ldots, K$ **do**
10: **for each** $l = S[k], \ldots, n$, **in increments of** η **do**
11: **for each** $q = l, \ldots, \min(l+\eta-1, \text{n})$, **in parallel do**
12: $\boldsymbol{c}_q \leftarrow$ GenerateCandidates($\boldsymbol{d}_q, \mathcal{I}_k, \epsilon$)
13: $O \leftarrow O \cup$ VerifyCandidates($\boldsymbol{d}_q, \boldsymbol{c}_q, \mathcal{I}_k, \epsilon$)
14: **return** O

4.2.1. *Cache-tiling*

Cache-tiling aims to increase cache locality during the candidate generation stage of the similarity search by ensuring the inverted index and accumulator structures fit in cache. To achieve this, the inverted index is split into several consecutive sections, called *tiles*, and each index is used in turn to find neighbors. Choosing the size of each cache tile is non-trivial in the APSS problem, due to the varying number of feature values being indexed for each object. For example, choosing to index the same number of objects in each tile will lead to large indexes for the final tiles to be processed which may not fit in cache. Instead, pL2AP first finds the first feature to be indexed in each object (line 4), which also provides the number of values to be indexed in each object. These counts are used to define the consecutive sets of objects to be indexed together in each tile. The list S, containing tile start and end offsets given the predefined processing order, is then used to index each object suffix in their assigned inverted index (line 8).

We use an array to track accumulated similarities for candidates. Since the accumulation array is randomly accessed for different candidates encountered while traversing the inverted index, nt accumulation arrays should also fit in cache along with the index, one for each thread. The size of the accumulation array is the same as the number of objects assigned to an index.

The un-indexed portion of each un-pruned candidate vector is sequentially accessed during candidate verification. To maximize cache locality, we explicitly create a sparse *forward index* containing prefix values for objects in each tile.

During parallel sections (lines 3 and 11), pL2AP follows a dynamic task partitioning approach, assigning a small set of objects to a thread to process as soon as it has finished processing its previous assigned set. Since candidate pruning is unpredictable, a thread may get assigned objects that finish processing quickly and may jump ahead many places in the processing order. This may lead to loss of cache locality if some threads read query objects from different portions of the dataset. To prevent this, we process queries η at a time, in a block synchronous fashion, where η is an input parameter, forcing threads to read from the same subset of query vectors, which should be located in close proximity in memory.

4.2.2. *Query vector mask-hashing*

During candidate verification, pL2AP traverses the candidate prefix sequentially, rather than the query prefix, and checks whether the query has non-zero values for the encountered features. When a common feature is found, query object meta-data (prefix ℓ^2-norm or maximum value) are used to check whether the candidate can be pruned. An efficient way to locate query vector values and meta-data during this process is to store them in arrays, as dense vectors. However, for datasets with high dimensionality (generally above 10^6), this technique can lead to polluting the cache with zero values from the dense arrays, evicting other necessary data.

Given that query vectors are sparse, and their features are always processed in a predefined order, we developed a heuristic hash-table data structure that uses a small amount of cache space, takes advantage of O(1) access times for most look-ups and leads to few collisions in practice. A small array of size $h + \max(||d_q||_0) - 1$ is used in pL2AP to store matching offsets in one or more lists containing the query data. Here, $h = 2^\alpha$ ($\alpha \geq 0$) is a predefined parameter, generally much smaller than m, and $\max(||d_q||_0)$ is the maximum number of non-zero features for any object. An efficient hashing function maps feature IDs to the $[0, h-1]$ domain, and collisions are entered in the hash-table array in order, starting with index h. Since partial dot-product computations with candidates follow the same traversal order, collisions can be quickly resolved by traversing only a subset of the overflow features. In practice, however, we have found that less than 1% of hash key look-ups end in collision.

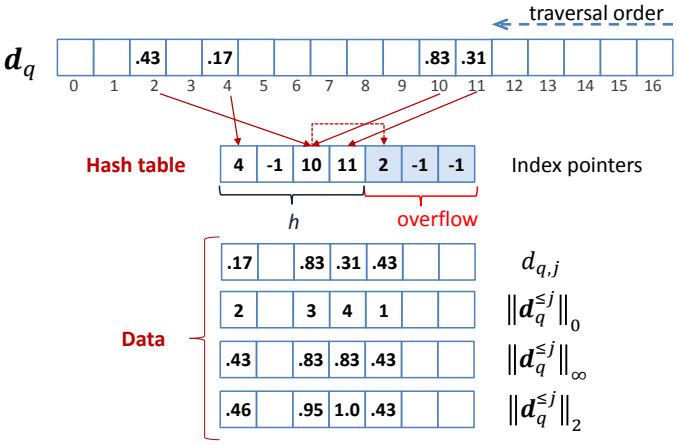


Figure 6: Example query hash table use in pL2AP.

Figure 6 provides an example of how a query object might use the hash table in pL2AP, for $h = 2^2$. The hash table array is initialized with negative values. Traversing the query non-zeros in reverse feature processing order, the 11th query feature is mapped to the 4th hash table cell, via an efficient truncate operation, $11\ \&\ (4-1)$, where $\&$ is the bitwise AND logical operator. The feature ID is stored in the hash table at the mapped key index, and one or more value arrays are populated with

salient information about the query at the same key index location. PL2AP tracks the query prefix value, size, maximum value, and ℓ^2-norm at each index, which are used to check different pruning bounds. In a similar fashion, the 10th query feature is mapped to the 3rd hash table cell, and the 4th query feature to the 1st hash table cell. When mapping the 2nd query feature, the collision is handled by entering the item in the overflow part of the hash table array, in traversal order. When verifying a candidate d_c, its forward index features are traversed in the same order as the query was traversed. Thus, when collisions occur, they can be found by partially traversing the overflow section of the hash table, keeping a pointer to the last cell with a feature ID greater or equal than the sought ID.

To avoid excessive collisions, pL2AP dynamically chooses whether to use the hash-table or dense arrays for the query object data. Specifically, objects with less than $h/2^3$ non-zeros will use the hash-table data structure, while the rest will use dense vector representations of the query and meta-data vectors.

4.3. PL2Knng

Algorithm 11 describes our parallel k-NNG construction method, pL2Knng. Our method follows the same computation strategy as L2Knng, incorporating the improvements described in Section 3.2. Namely, an approximate graph is first constructed, which provides filtering thresholds when deriving the exact neighborhood graph. Then, for each query object, pL2Knng indexes some of its prefix values, ensuring that the query object can be found in subsequent searches by objects that belong in the query neighborhood or whose neighborhood the query can enhance. During *candidate generation* (CG), using the index, pL2Knng selects a list of candidates for the query, which are a superset of its correct neighbors. Part of the query similarity value with each candidate is computed during the CG stage, and upper-bound estimates on the similarity are used to prune some of the candidates. Finally, pL2Knng completes the similarity computation in the *candidate verification* (CV) stage, performing additional pruning based on several upper-bound similarity estimates, and updates the query and candidate neighborhoods if the result can enhance them. For full details on the filtering process, see [4].

Threads concurrently process different query objects in pL2Knng. We devised a lock-less thread cooperation and neighborhood update strategy that allows threads to dynamically share available work and leads to good load balance in general. In the remainder of this section, we will describe these strategies, which are incorporated both in the initial approximate graph construction and the filtering stages in pL2Knng.

4.3.1. Cache-tiling

In order to enable cooperative processing of different query objects in its filtering phase, pL2Knng indexes objects prior to filtering. The index is split into several *tiles*, corresponding to a set of consecutive objects in the object processing order, and each index is used in turn to find neighbors. During filtering, threads can all read the sections of the index they need in order to find candidates for their respective query objects. Since many different sections of the index may be accessed concurrently, it is beneficial for the index to fit in the available cache memory on the system. The index size is highly data dependent.

Algorithm 11 The pL2Knng algorithm.

function PL2KNN($D, k, \zeta, \theta, \eta$)
  $\hat{\mathcal{N}} \leftarrow pL2KNN\text{-}a(D,\ k)$
  Set object processing order given $\hat{\mathcal{N}}$
  $z \leftarrow 0,\ r \leftarrow 0,\ i \leftarrow 1,\ \mathcal{I} \leftarrow \varnothing$
  while $i \leq n$ **do**
    $j \leftarrow i$
    for each $i = j, \ldots, n$ **do** ▷ Identify next tile
      $S \leftarrow$ FindIndexSplit($\boldsymbol{d}_i, \sigma_{d_i}$)
      $z \leftarrow z + nnz(\boldsymbol{d}_i^{>})$
      $r \leftarrow r + 1$
      if $z \geq \zeta$ **or** $r = \theta$ **then**
        $i \leftarrow i + 1$
        break
    for each $q = j, \ldots, i$ **in parallel do** ▷ Create index $\mathcal{I}$
      Index($\boldsymbol{d}_q, \mathcal{I}, S, \sigma_{d_q}$)
    for each $l = j, \ldots, n$, **in increments of** η **do** ▷ Filter
      for each $q = l, \ldots, \min(l+\eta-1, \text{n})$, **in parallel do**
        $\boldsymbol{c}_q \leftarrow$ GenerateCandidates($\boldsymbol{d}_q, \mathcal{I}, k$)
        VerifyCandidates($\boldsymbol{d}_q, \boldsymbol{c}_q, \mathcal{I}, \hat{\mathcal{N}}, k$)
    $\mathcal{I} \leftarrow \varnothing$
    Update un-processed object processing order given $\hat{\mathcal{N}}$
  end while
return $\hat{\mathcal{N}}$

Cache-tiling in pL2Knng is similar to the procedure described in Section 4.2.1 for pL2AP. However, the number of values that are indexed in pL2Knng depends on the magnitude of those values and the current minimum similarity in the object's neighborhood, which is not known a priori and changes throughout the algorithm execution. A poor quality (low recall) initial approximate graph, for example, will lead to more values that need to be indexed in each object to ensure a correct result. The size of each tile is dynamically chosen to contain at most θ indexed objects and ζ indexed non-zeros.

In pL2Knng, objects are processed in non-increasing value order of their minimum neighborhood similarity σ_{d_i}. Taking advantage of the commutative property of cosine similarity, pL2Knng only compares a query object against objects that come before it in the object processing order. After completing the filtering process using the current index, it can be discarded. The filtering leads to improved minimum neighborhood similarities of un-indexed objects in the neighborhood graph represented by $\hat{\mathcal{N}}$. As a result, pL2Knng then updates the object processing order of un-indexed objects, improving index reduction and pruning during subsequent searches that use the next index tile.

After indexing a set of objects, pL2Knng splits the set of query objects (those that come after the first indexed object in the processing order) into query blocks of size η. Threads are then dynamically assigned a small number of consecutive queries at a time, which they process sequentially. Our method keeps track of the k-nearest neighbors of an object by using

a heap data structure. Note that, after finding neighbors for a given query object, a thread can safely update the query neighborhood heap. However, it cannot also update the neighborhood of a candidate without locking, as another thread may be trying to concurrently update the same heap. As such, `pL2Knng` keeps a candidate list in memory for each of the objects in the query block, deferring candidate neighborhood updates until all query block objects have been processed. The parameter η should be chosen to ensure $\eta \times \theta$ values can be stored in memory, as each candidate list has a maximum size of θ. Moreover, moderately small η values can ensure the candidate lists are cache-resident, leading to improved performance. The same query set cache-tiling strategy is also used in the IC and GE phases of our method. However, each candidate list size is μ there, so the memory necessary to store candidates is $\eta \times \max(\mu, \theta)$.

4.3.2. Neighborhood updates

As mentioned in the previous section, since each object is independently processed by a thread, each thread can update the query neighborhood as soon as it has found a candidate object that can improve it. We have found it beneficial, however, to update the query neighborhood after finalizing similarity computations for all candidates. Given a set of candidates C with $|C| > k$, we first select [27] the top-k values in the list, filtering out those less than σ_q, the current minimum similarity in the query neighborhood, and then sequentially insert them in the query heap.

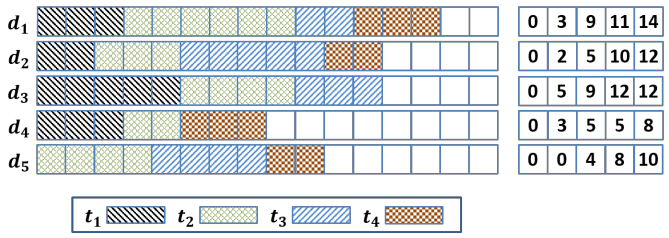


Figure 7: Segmentation of candidate lists for neighborhood updates.

Our strategy for updating candidate neighborhoods is slightly different. Each thread is assigned a sequential block of n/p candidate objects whose neighborhoods they are responsible to update, where p is the number of processing elements (threads). When a candidate list is constructed, candidates are added in the order they are found during the candidate selection process, which results in a semi-random ordering. After updating the query neighborhood, before moving on to the next query, the thread re-arranges the similarities in the candidate list to ensure efficient candidate list updates. Each value is checked against the minimum similarity σ_c of the candidate neighborhood and discarded if it cannot improve that neighborhood. The thread then partitions the remaining values into p sections s.t. the ith section contains similarities for objects in the ith candidate block, which will be updated by the ith thread after the query block has finished being processed. The thread also records the starting and ending offset of each segment in the candidate list. Figure 7 shows this strategy for objects d_1–d_5 from a set of 16 objects, given 4 threads. This partitioning enables fast candidate neighborhood updates at the end of each query block processing, as each thread only needs to traverse a subset of each candidate list to perform its required updates.

Table 4: Dataset Statistics.

Dataset	n	m	nnz	μ_r	σ_r	μ_c	σ_c
RCV1	0.80M	0.05M	62M	76	55	1347	8350
WW500	0.24M	0.66M	202M	830	386	306	3323
WW200	1.02M	0.66M	437M	430	302	659	8273

For each dataset, n is the number of vectors/objects (rows), m is the number of features (columns), nnz is the number of non-zero values, μ_r and σ_r are the mean and standard deviation of row lengths (number of non-zeros), and μ_c and σ_c are the same statistics for column lengths.

4.4. Parallel implementation details

Given the `pL2Knng` neighborhood update strategy described in Section 4.3.2, the parallel execution of our algorithm does not lead to any race conditions. In `pL2AP`, threads perform independent similarity searches. At the end of each query tile, a single thread consolidates the results from all threads into the output graph. The parallel implementation of our methods is thus straight-forward, relying only on OpenMP `parallel for` loops and `barrier` statements.

5. Experimental Methodology

In this section, we describe the datasets, baseline algorithms, and performance measures used in our experiments.

5.1. Datasets

We use three text-based datasets to evaluate each method. They represent some real-world and benchmark text corpora often used in text-categorization research. Their characteristics, including number of objects (n), features (m), and non-zeros (nnz), row/column length mean and standard deviation ($\mu_{r/c}$, $\sigma_{r/c}$), are detailed in Table 4. Standard pre-processing, including tokenization, lemmatization, and *tf-idf* scaling, were used to encode text documents as vectors. We present additional details below.

- **RCV1** is a standard benchmark corpus containing over 800,000 newswire stories provided by Reuters, Ltd. for research purposes, made available by Lewis et al. [32].
- **WW500** contains documents with at least 500 distinct features, extracted from the October 2014 article dump of the English Wikipedia[3] (Wiki dump).
- **WW200** contains documents from the Wiki dump with at least 200 distinct features.

As can be seen from the mean and standard deviation values listed in Table 4, the datasets we chose are quite varied

[3] http://download.wikimedia.org

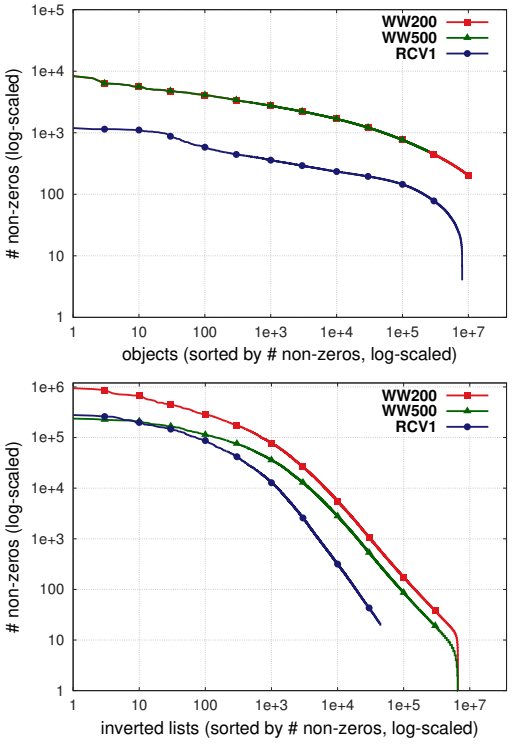


Figure 8: Non-zero distributions in rows / objects (top) and columns / inverted lists (bottom) in our three datasets. In the top graph, the WW200 and WW500 lines coincide, as WW500 is a proper subset of WW200.

with respect to their row and column lengths. Figure 8 provides another insight into the non-zero composition in the three datasets, showing the distribution of row (top) and column (bottom) lengths. As both row and column frequency distributions in the datasets follow the power-law, we plot the graphs log-log scaled. Note that the WW200 and WW500 lines coincide in the top graph for most points, as WW500 is a proper subset of WW200.

5.2. *Baseline approaches*

We compare our methods against the following baselines.

5.2.1. *ϵ-NNG construction experiments*

- `IdxJoin`, `APT`, and `L2AP` are baseline serial APSS search methods described in detail in [3]. We report speedup over the fastest execution time of any of the serial methods.
- `pL2AP` is our parallel ϵ-NNG construction method, detailed in Section 4.2.
- `pIdxJoin` uses similar cache-tiling as `pL2AP`, but does not use any pruning when computing similarities. For each block of queries, `pIdxJoin` sequentially retrieves a block of objects to search against and indexes all their values. Threads then share the index to compute similarities, via accumulation, of each assigned query object against all indexed objects, retaining those resulting pairs above the threshold ϵ.
- `pAPT` is the shared memory parallel APSS method by Awekar and Samatova [9], which we described in detail in 4.1.
- $\texttt{pL2AP}_{\texttt{rr}}$ follows the same parallelism strategy as `pAPT`, but takes advantage of the advanced pruning bounds of `L2AP`. After first indexing the suffixes of all objects, $\texttt{pL2AP}_{\texttt{rr}}$ dynamically assigns small sets of query objects for processing to available threads. For each query object, $\texttt{pL2AP}_{\texttt{rr}}$ indexes the same values and performs the same pruning in the candidate generation and verification stages as `pL2AP`.

5.2.2. *k-NNG construction experiments*

- `pL2Knng` is our parallel k-NNG construction method, detailed in Section 4.3.
- `pKIdxJoin` is a straight-forward baseline similar to *IDX* in [37]. The method uses similar cache-tiling as `pL2Knng`, but does not use any pruning when computing similarities. For each set of queries, `pKIdxJoin` sequentially retrieves a set of objects to search against and indexes all their values. Threads then share the index to compute similarities, via accumulation, of each assigned object in a query tile against all indexed objects, retaining the top-k matches for each object.
- *GF* is an approximate k-NNG construction method proposed by Park et al. [37]. We have created a shared memory parallel version of *GF*, which we call *pGF*, using the same thread cooperation strategy as in `pL2Knng-a`. Threads first work together to index enough high-weight features for each object to ensure μ candidate neighbors have at least one feature in common with each input object. Then, they dynamically split the work of computing similarities of each object in an inverted list against all other objects in the list. Each thread updates the neighborhood of an assigned query object as soon as it has finished computing the similarity with a candidate object. Threads synchronize at the end of each inverted index list, reading computed similarities by all threads in order to update neighborhoods for blocks of objects assigned to each thread.
- *NN-Descent* is a shared memory parallel approximate k-NNG construction method designed by Dong et al. [24] to work with generic similarity measures which has been shown effective for both sparse and dense input.

Locality sensitive hashing (LSH) has been a popular method for top-k search, but we have found that it does not in general perform well in the k-NNG construction setting when one requires high average recall. Both *GF* and *NN-Descent* have been shown to outperform LSH in this setting, for k typically ≥ 10. Moreover, `pL2Knng` significantly outperforms *GF* and *NN-Descent* in both serial and parallel execution environments. As a result, we have chosen not to compare against LSH in this work.

5.3. *Performance measures*

When comparing approximate k-NNG construction methods, we use average recall to measure the accuracy of the returned result. We obtain the correct k-NNG via a brute-force search, then compute the average recall as,

$$R = \frac{1}{|D|} \sum_{d_i \in D} \frac{\text{\# correct neighbors in } N_{d_i}}{|N_{d_i}|}.$$

An important characteristic in our experiments is *CPU runtime*, which is measured in seconds. I/O time needed to load the dataset into memory or write output to the file system should be the same for all methods and is ignored. Between a method A and a baseline method B, we report *speedup* as the ratio of B's execution time and that of A's. Additionally, we report strong scaling for parallel methods, in which multi-threaded execution times are compared with the 1-threaded execution of the same method.

As a way to compare the amount of time threads spend waiting for other threads to finish execution, we measure load imbalance, as suggested by DeRose et al. [21] as,

$$\% \text{ imbalance} = \frac{t_{\max} - t_{\text{mean}}}{t_{\max}} \times \frac{p}{p-1},$$

where p is the number of processing elements (threads) and $t_{\max}$ and t_{mean} are the maximum and mean thread times in the parallel block, respectively.

5.4. Execution environment

Our method and all baselines are implemented in C and compiled using gcc 5.1 with the -O3 optimization setting enabled. We used the OpenMP framework for implementing shared-memory parallel methods. Each method was executed on its own node in a cluster of Linux servers. Due to hardware availability restrictions, the ϵ-NNG and k-NNG sets of experiments (our methods and all related baselines for each) were executed on two different clusters. For the ϵ-NNG experiments, each server was a dual-socket machine, equipped with 64 GB RAM and two twelve-core 2.5 GHz Intel Xeon E5-2680v3 (Haswell) processors. Each core is equipped with 32 KB L1 cache and 256 KB of L2 cache, and the 12 cores on each socket share 30 MB of L3 cache. For the k-NNG experiments, each server was a dual-socket machine, equipped with 64 GB RAM and two eight-core 2.6 GHz Intel Xeon E5-2670 (Sandy Bridge) processors. Each core is equipped with 32 KB L1 cache and 256 KB of L2 cache, and the 8 cores on each socket share 20 MB of L3 cache. Both servers were running CentOS 6.9 (Final).

For each ϵ-NNG method, we varied the similarity threshold ϵ between 0.3 and 0.9, in increments of 0.1. For `pL2AP`, we fixed η at $25K$ objects and varied ζ between $250K$ and $4M$ in $250K$ increments. We set the masked hash-table size parameter h to 2^{13}.

We executed each k-NNG method for

$$k \in \{10, 25, 50, 75, 100, 200, 300, 400, 500\}$$

and tuned parameters for each method to achieve balanced high recall and efficient execution. For all `L2Knng` based methods, we set the parameter $\delta = 0.0001$. For all experiments, we set the `pL2Knng` parameter $\theta = 100K$. We used the latest version of the *NN-Descent*[4] library available at the time of our experiments (v.1.4), and set $\rho = 1$, and indexing $K = \mu$ (the candidate list size $\mu \geq k$). For all stochastic methods, we executed a minimum of 3 tries for each set of parameter values and we report averages of all tries.

[4] http://www.kgraph.org/releases/kgraph-1.4-x86_64.tar.gz

6. Results & Discussion

We will first present our ϵ-NNG construction experiment results, followed by those for the k-NNG construction experiments and a short discussion.

6.1. PL2AP results

We now present our `pL2AP` experiment results, along two directions. First, we study the effectiveness of our method with regards to filtering unpromising object pairs. We compare pruning effectiveness in `pL2AP` with that in `pAPT`, identify how early candidates are pruned, and measure cache locality improvements in our method. We also study the effect input parameters have on the performance of our method. Second, we study the efficiency of `pL2AP` in solving the APSS problem. We identify the best pruning choices in `pL2AP`, compare its execution time with that of other parallel methods and the best known serial method for solving the problem, study the strong scaling characteristics of our method, and measure how balanced the loads of different threads are during execution.

6.1.1. Effectiveness study

Pruning effectiveness comparison with pAPT. Our method, `pL2AP`, and the shared memory parallel baseline `pAPT`, follow the same strategy in solving the APSS problem. They build a partial inverted index that is used to identify, for each query object, a list of candidates the query should be compared with. While comparing query objects with candidates, they prune as many un-promising pairs as possible, and in the end fully compute the dot-product of a small subset of the candidate list, which is a superset of the correct set of neighbors. While their serial computation strategy is the same, the two methods rely on different theoretic similarity upper bounds to decide which values in the query object should be indexed, whether an object should become a candidate, and when a candidate should be pruned.

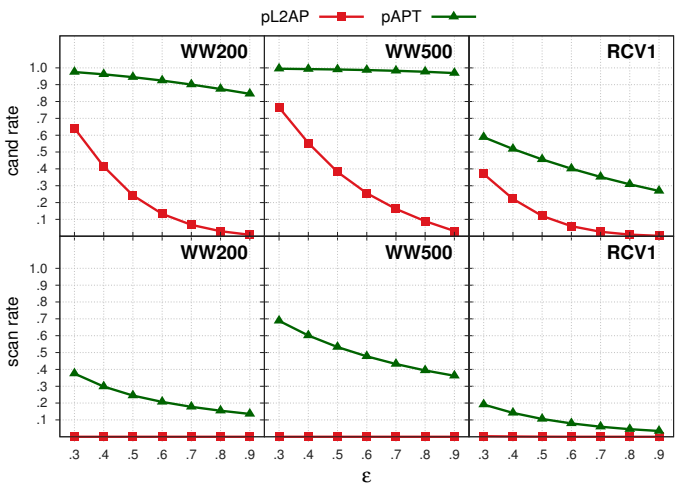


Figure 9: Percent of potential candidate pairs (cand rate), and percent of potential dot-products (scan rate) comparison between `pL2AP` and `pAPT`, for ϵ between 0.3 and 0.9.

Indexing fewer values can speed up index traversal and thus lead to performance improvements. In addition, it will lead to shorter candidate lists being generated. Considering fewer candidates, as well as more aggressive pruning, can lead to fewer dot-products being computed in full and to better performance. Figure 9 shows the percent of potential candidate pairs (cand rate), and percent of potential dot-products (scan rate) for pL2AP and pAPT, for ϵ between 0.3 and 0.9, for the three datasets. As compared to pAPT, our method considers fewer candidates, and evaluates fewer complete dot-products, especially at high similarity values; pL2AP is able to prune a much higher number of candidates than pAPT in all datasets, highlighting the improved pruning effectiveness in our method. The size of the un-pruned set of candidates in pL2AP was between 1.002x–2.179x the size of the set of correct neighbors. Interestingly, pruning was more aggressive in the Wikipedia datasets, where the un-pruned set ranged between 1.002x–1.055x the size of the set of correct neighbors, than in the RCV1 dataset, where its range was 1.460x–2.179x. This may be due to the fact that RCV1 has a much more compact feature space, which allows for more features in common between random objects.

Pruning effectiveness in pL2AP. Our method works by pruning the majority of the candidates that are not correct neighbors. Once an object becomes a candidate, it can be pruned by the $l2cg$ bound while accumulating values traversing the inverted index in the candidate generation stage (*cg* in figures), when checking the ps, dps_1 and dps_2 prefix similarity estimate bounds at the onset of the candidate verification stage (*ses* in figures), or by the $l2cv$ bound while accumulating values traversing the forward index in the candidate verification stage (*cv* in figures). Earlier pruning of candidates means less time spent accumulating dot-products in vain and will lead to improved performance. In an experiment in which we used consistent parameters for all datasets ($h = 2^{13}$, $\eta = 25K$, and $\zeta = 1M$), we counted the number of candidates pruned in each stage of the algorithm. We report these values in Figure 10, for all datasets and ϵ values, along with the number of candidates that were not pruned and had their dot-products computed in full (*dps* in figures).

Results show that pL2AP prunes the majority of objects soon after they become candidates, in the candidate generation stage (*cg*). Most of the remaining objects are pruned by the *ses* bounds, which are checked once, at the beginning of the candidate verification stage, and by additional pruning in the candidate verification stage (*cv*). At $\epsilon = 0.3$, for example, only 0.02%–0.90% of candidates survived all pruning across our datasets.

A large number of objects never become candidates in pL2AP, as a result of either the ℓ^2-norm based candidate acceptance bound in the candidate generation stage of the algorithm, or due to the prefix-filtering based index reduction. On average across all ϵ values, 11.6%–32.0% of all potential candidates actually became candidates for our datasets. Of those, most are pruned quickly, in the first stage of our method. As a way to gauge how quickly candidates are pruned, we measured the number of executed multiply-adds versus the number of possible multiply-adds (percent of accumulated non-zeros) in the similarity computation of each pruned candidate. In Figure 11, we report the mean percent accumulated non-zeros for our three datasets. In each experiment, we used consistent parameters for all datasets (1 thread, $h = 2^{13}$, $\eta = 25K$, and $\zeta = 1M$).

The results show that, for all of the datasets and most ϵ values, before pruning unsuitable candidates, pL2AP accumulates less than 4% of the common non-zeros. In the case of RCV1, which may not use as many rare terms as Wikipedia, and a high similarity threshold ($\epsilon = 0.9$), the algorithm checks on average about 20% of the vector non-zeros before pruning candidates.

Cache locality improvements in pL2AP. While pL2AP performs the same pruning as L2AP, it scans each query object multiple times to compare against objects in multiple constructed inverted indexes. The smaller inverted indexes and the mask-based hash table used during the search help avoid cache thrashing, improving efficiency by reducing time wasted waiting for data transfers from memory to cache. To measure the serial effect of this improvement, we compared the 1-threaded execution of pL2AP against the serial L2AP algorithm. We used $\eta = 25K$ objects and $\zeta = 1M$ non-zeros for this test. Figure 12 shows speedup results for each of the three datasets we tested, for ϵ between 0.3 and 0.9. The results show an improvement over L2AP for datasets with long inverted lists.

The small inverted index in pL2AP is shared by all threads in executing concurrent searches. As another way to quantify cache locality improvements, we compared the percent of cache misses when executing pL2AP and $\texttt{pL2AP}_{\texttt{rr}}$ with 24 threads. Both algorithms perform the same pruning, but $\texttt{pL2AP}_{\texttt{rr}}$ builds a single inverted index and does not consider cache locality in its execution. We used the *perf* Linux utility to count the number of instructions, cache references, and cache misses (`perf stat -v -B -e instructions,cache-references,cache-misses`) in the shared last-level cache (LLC) of the Xeon processors. Figure 13 shows our results when executing pL2AP with ζ between $0.5M$ and $4M$ non-zeros and $\texttt{pL2AP}_{\texttt{rr}}$, on the RCV1 and WW200 datasets, for $\epsilon = 0.3$. We show the size of the inverted index that $\texttt{pL2AP}_{\texttt{rr}}$ builds below its bar in the graph. The figure shows the percent of LLC misses as opposed to total number of instructions executed by the CPU. We observed similar results for most other datasets and ϵ values. In general, pL2AP improves cache locality, resulting in much fewer cache misses than the $\texttt{pL2AP}_{\texttt{rr}}$ variant. The results show that small inverted indexes ($\zeta \leq 1.5$M) fit better in cache and lead to fewer cache misses in general. While 1.2–1.6% of the $\texttt{pL2AP}_{\texttt{rr}}$ instructions result in cache misses, less than 0.05% of the pL2AP instructions do so for $\zeta = 1.5$M. Moreover, pL2AP cache misses represent 1.5–4.1% of the cache references, as oposed to 24–32% for $\texttt{pL2AP}_{\texttt{rr}}$.

Parameter sensitivity. Our method, pL2AP, is controlled by three parameters. The size of the mask-based hash table, h, is dependent on the dimensionality of the feature space. Choosing a small h value for a dataset with large dimensionality will

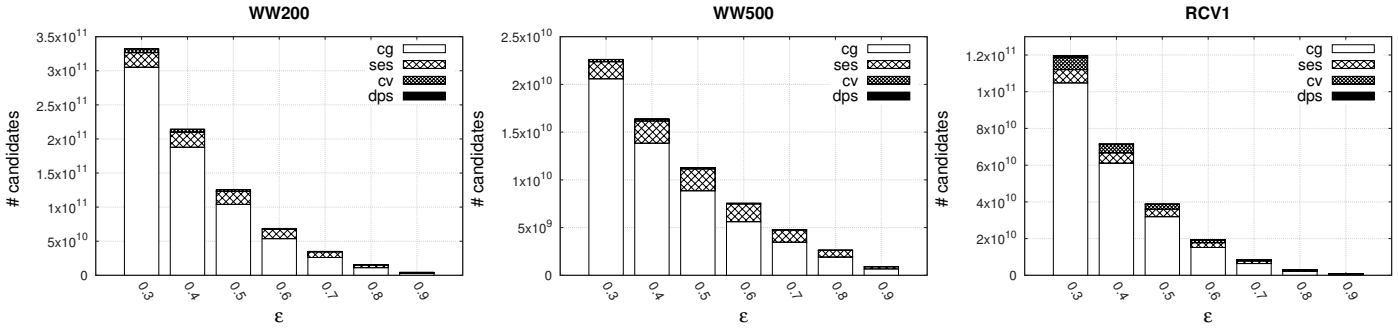


Figure 10: Pruning effectiveness in pL2AP. Each bar shows the number of candidates pruned during candidate generation (cg), at the onset of candidate verification (ses), during candidate verification (cv), and the number of candidates whose similarity was computed in full (dps), for ϵ ranging between 0.3 and 0.9.

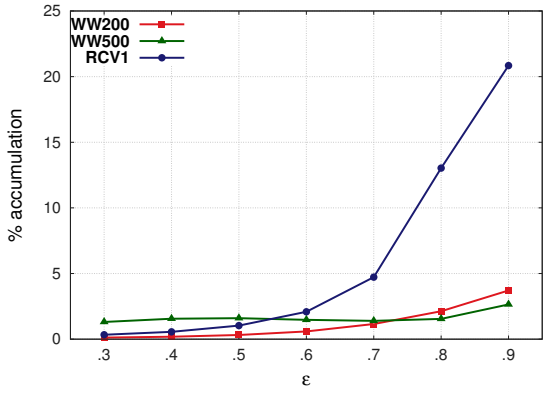


Figure 11: Mean percent accumulated non-zeros before pruning in pL2AP.

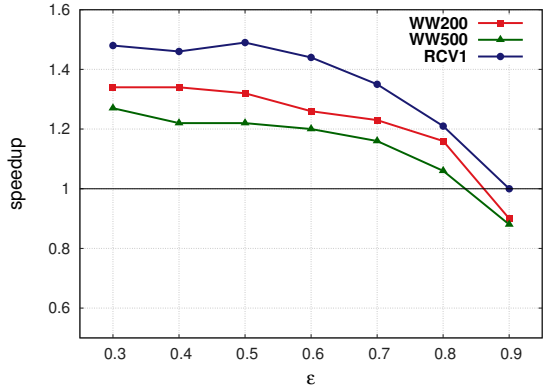


Figure 12: Speedup of 1-threaded pL2AP over L2AP.

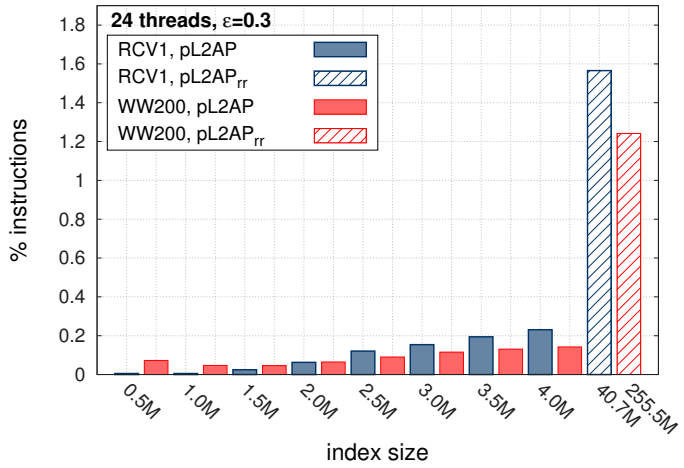


Figure 13: Percent of instructions resulting in LLC misses when executing pL2AP$_{rr}$ and pL2AP with ζ between 0.5M and 4M non-zeros on the RCV1 and WWW200 datasets.

likely cause many hash table collisions and slow down execution. Similarly, the ζ parameter dictates the number of non-zeros that should be included in each inverted index, which dynamically decides the size of each cache tile. Choosing a small ζ value will lead to many inverted indexes being created which may lead to slow-downs due to repeated traversals of the query objects. On the other hand, choosing a ζ value that is too large will diminish the cache locality benefits of our tiling strategy. To ascertain the sensitivity of pL2AP to these parameter choices, we tested different values of each parameter while keeping the other two unchanged.

In the first experiment, we set ζ to $1M$ non-zeros and η to $25K$ and varied h between 2^5 and 2^{15}. Results of these experiments over our three datasets are shown on the left side of Figure 14, as execution times relative to the $h = 2^{13}$ parameter choice for each dataset. Our method is not very sensitive to this parameter in general.

The middle section of Figure 14 shows execution times for each dataset, given $h = 2^{13}$ and $\zeta = 1M$, for η between $1K$ and $50K$, relative to the execution time for $\eta = 25K$. We found that choosing the size of each bulk synchronous block, η, does not affect performance in pL2AP, as long as the η value is not too small. We found any values above $5K$ to be adequate for all datasets.

Finally, we tested the sensitivity of the ζ parameter, for values between $0.25M$ and $3.0M$, given $\eta = 25K$ and $h = 2^{13}$, and show times relative to the $\zeta = 1M$ execution in the right section of Figure 14. While the ζ choice will be dependent on the cache configuration of the target system, our experiments showed that pL2AP performed well for most datasets given ζ set to at least $1M$ non-zeros.

6.1.2. *Efficiency study*

Effect of pruning choices on efficiency. Pruning is an effective mechanism for reducing the number of similarity computations

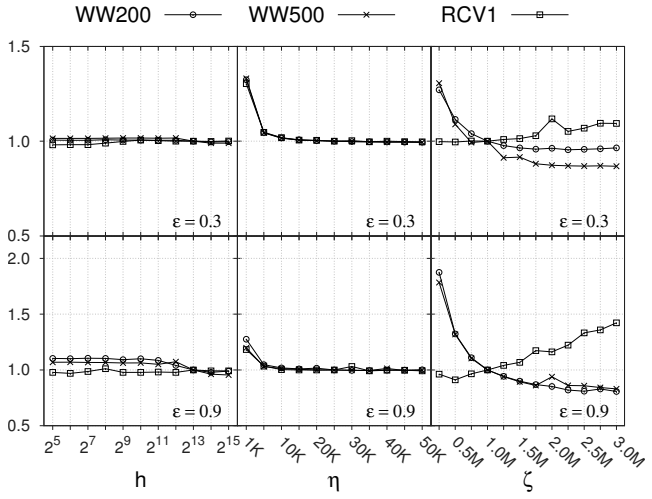


Figure 14: Relative execution times for different h, η, and ζ parameter choices, given $\epsilon = 0.3$ (top) and $\epsilon = 0.9$ (bottom).

Table 5: Tested pL2AP pruning strategies.

Strategy	Bounds checked	Index update
base	$\{idx, rs, ps, l2cg, l2cv, dps_1\}$	no
sz	base + $\{sz\}$	no
dp	base + $\{dps_2\}$	no
szdp	base + $\{sz, dps_2\}$	no
szdpupd	base + $\{sz, dps_2\}$	yes

Table 6: Performance of different pruning choice configurations in pL2AP.

versus	mean	stdv	min	max
	WW200			
nbase	**0.9933**	0.0163	0.9719	1.0240
dp6	1.0034	0.0112	0.9948	1.0296
sz	1.0000	0.0000	1.0000	1.0000
szdp	1.0152	0.0119	1.0047	1.0369
szdpupd	1.0661	0.0481	1.0139	1.1586
	WW500			
nbase	**0.9980**	0.0109	0.9894	1.0234
dp6	1.0115	0.0178	0.9975	1.0540
sz	1.0000	0.0000	1.0000	1.0000
szdp	1.0249	0.0212	1.0097	1.0744
szdpupd	1.0442	0.0354	1.0145	1.1241
	RCV1			
nbase	1.0017	0.0051	0.9909	1.0086
dp6	1.0090	0.0061	1.0027	1.0215
sz	**1.0000**	0.0000	1.0000	1.0000
szdp	1.0104	0.0055	1.0029	1.0212
szdpupd	1.0240	0.0137	1.0108	1.0515

Execution times for each configuration were normalized by respective execution times of the *sz* configuration. We present the mean, standard deviation (std), minimum and maximum of experiment results across all ϵ values, given $h = 2^{13}$, $\eta = 25K$ and $\zeta = 1M$ input parameters. The best mean performance is highlighted with bold.

that must be executed to solve the APSS problem. However, bounds checking incurs additional costs which may not outweigh their benefit. Previous experiments in [3] proved the effectiveness of our ℓ^2-norm based bounds in each stage of the search framework, and showed the sz and dps_2 bounds had little effect in general over the search efficiency. As a way to quantify this effect when executing with multiple concurrent threads, we tested pL2AP in four configuration scenarios, listed in Table 5. The "base" configuration did not effect any pruning based on the sz or dps_2 bounds. The "sz" and "dp" configurations enabled pruning based on the sz and dps_2 bounds, respectively, and the "szdp" configuration enabled pruning based on both the sz and dps_2 bounds. When checking the sz bound, pAPT removes values associated with short vectors from the beginning of inverted lists, which can potentially improve efficiency. We added this capability to pL2AP and tested it in the configuration "szdpupd", which enables all pruning strategies and also performs index updates. Using the same input parameters for all datasets ($nt = 24$, $h = 2^{13}$, $\eta = 25K$ and $\zeta = 1M$), we recorded search execution times under each scenario.

Table 6 reports the results of our experiment. For each ϵ value, times in all configuration scenarios were normalized by that of the sz scenario, and we report the mean, standard deviation (std), minimum and maximum of experiment results across all ϵ values. The best performing results are highlighted in bold. The sz and dp configurations showed little improvement over the base one, at times leading to slower execution times. Checking the sz bound was beneficial in most cases and had better performance than checking the dps_2 bound in general. The combined scenario szdp did not perform better than the sz scenario on average. The results in the remainder of this work assume the sz configuration.

In general, the index update strategy did not improve performance. For some datasets, its execution was 1.22x–1.40x slower than that of the szdp configuration, which effected the same pruning without updating the index. The worse efficiency is likely due to loss of cache locality having to interrupt traversing inverted lists to update their start pointer, as well as copying the list of pointers for each thread, which in pL2AP occurs for each constructed inverted index.

Comparison with serial methods. We compared the execution time of all parallel methods, executed with 24 threads, with the best serial execution time achieved by any of the serial algorithms. Figure 15 and Table 7 show the results of this experiment. In all cases, pL2AP had the best execution time of all parallel methods, achieving serial speedups of 12x–34x for our three datasets. Compared to previously published parallel baselines, pL2AP executed 7x–238x faster. While $\text{pL2AP}_{\texttt{rr}}$ uses the same type of pruning as pL2AP, it traverses the entire inverted index during each query and, as a result, cannot perform as well. Instead, by using tiling and other optimizations that promote cache locality, pL2AP is able to achieve very good speedup for datasets with long inverted index lists. At high similarity thresholds, however, pL2AP is able to prune candidates quickly and does not need to traverse many candidate and query vector features, rendering our cache locality optimizations less effective.

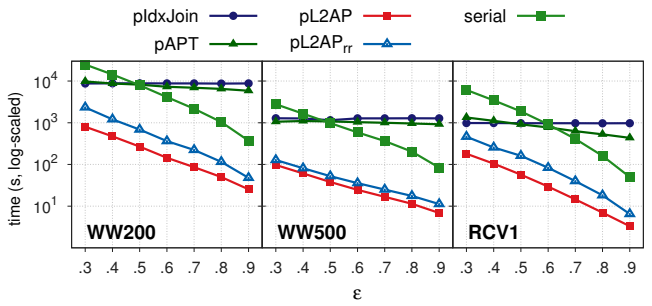


Figure 15: Execution times of parallel methods and the best serial alternative, at varying threshold values ϵ.

Table 7: ϵ-NNG construction efficiency comparison.

method	$\epsilon = 0.3$	0.5	0.7	0.9
	WW200			
serial	24923.36	7942.84	2173.20	371.19
pIdxJoin	8746.13	8824.70	8795.95	8797.46
pAPT	9766.05	8195.22	6901.25	5857.95
pL2AP$_{rr}$	2329.08	686.88	222.51	47.65
pL2AP	801.98	267.44	85.86	25.29
	WW500			
serial	2782.03	979.62	363.78	84.16
pIdxJoin	1279.46	1149.38	1276.69	1273.83
pAPT	1056.98	1076.20	997.40	918.88
pL2AP$_{rr}$	127.49	52.28	24.95	11.24
pL2AP	97.41	37.93	16.74	6.85
	RCV1			
serial	6122.15	1877.35	410.25	49.82
pIdxJoin	985.89	976.48	972.89	973.98
pAPT	1338.23	916.89	627.58	433.42
pL2AP$_{rr}$	462.48	161.32	39.76	6.48
pL2AP	180.79	56.74	14.51	3.30

Values represent execution times, measured in seconds. All methods except *serial* were executed with 24 threads. The *serial* values represent the best execution time of any serial method, which in all cases was achieved by L2AP.

As expected, the pIdxJoin algorithm, which does not perform any pruning, was very slow in comparison to the other parallel methods. It performed very poorly, much slower even than L2AP, the fastest serial method, potentially due to the high data dimensionality. The pAPT method of Awekar and Samatova was also slow in our experiments. It was not able to prune as many candidates as pL2AP in general, and ended up performing many more unnecessary similarity computations.

Strong scaling. Figure 16 shows the strong scaling results from our experiments. The amount of work pL2AP does when processing each query increases as the threshold ϵ decreases. At high values of ϵ, many of the objects never become candidates for a query due to the idx and rs bounds in our method, and pL2AP is able to quickly dismiss candidates. For example, the size of the candidate list when $\epsilon = 0.9$ is 0.5–4.0% of the candidate list size when $\epsilon = 0.3$. As a result, the cache locality improvements in pL2AP are not as beneficial, resulting in less pronounced scaling at $\epsilon = 0.9$. On the other hand, pL2AP shows linear scaling at $\epsilon = 0.3$.

It is interesting to note that pAPT and pL2AP$_{rr}$ both scale poorly above twelve threads. This may be an indication of thrashing, which is causing threads to waste time waiting for

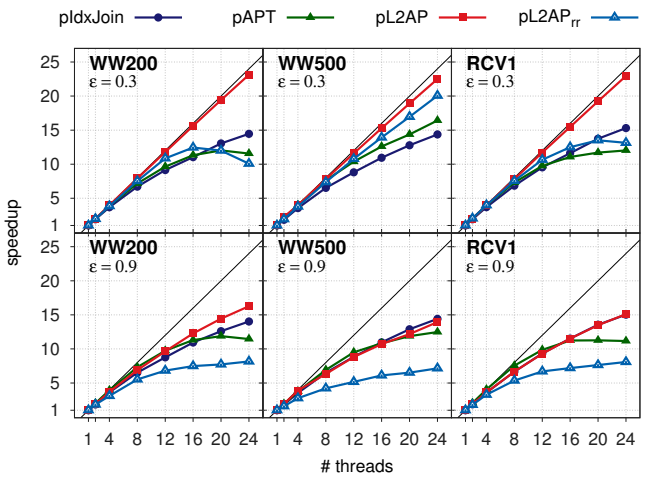


Figure 16: Strong scaling of parallel methods at $\epsilon = 0.3$ (top) and $\epsilon = 0.9$ (bottom).

cache lines to be fetched from main memory.

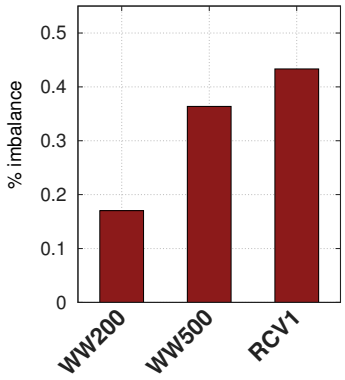


Figure 17: Load imbalance in pL2AP.

Load Balance. In order to test the effectiveness of the dynamic task partitioning approach in pL2AP, we measured the amount of time each thread spent searching for neighbors. Figure 17 shows the percent load imbalance averaged over all ϵ values, in experiments with consistent parameters ($nt = 24$, $h = 2^{13}$, $\eta = 25K$, and $\zeta = 1M$). Our method shows little imbalance between the threads, much less than 1% for our datasets.

6.2. PL2Knng

Our k-NNG construction experiment results are organized along two directions. First, we present results from evaluating the accuracy and efficiency of our parallel approximate method, pL2Knng-a, in comparison to two state-of-the-art approximate baselines. Second, we present results from evaluating our exact method, pL2Knng. We measure serial efficiency improvements compared to the original L2Knng algorithm, study our method's sensitivity to parameter choices, compare the efficiency and strong scaling characteristics of pL2Knng with parallel and approximate baselines, and study load imbalance in our method.

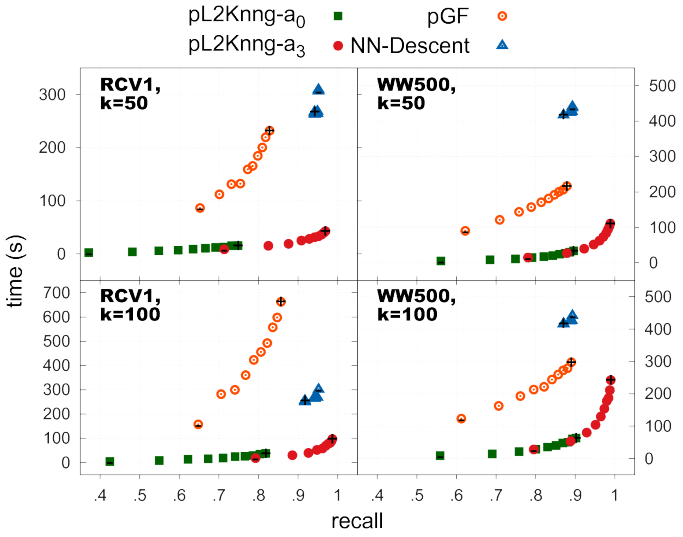


Figure 18: k-NNG construction effectiveness.

6.2.1. Approximate k-NNG construction

Effectiveness comparison. As a way to compare the effectiveness of the approximate methods, we executed each for $\mu \in \{1k, \ldots, 10k\}$, where μ is the size of the candidate list each method considers. Figure 18 shows the results for two datasets, RCV1 and WW500, and two k values, $k \in \{50, 100\}$. The best results in each quadrant of the figure are those in the lower right corner, representing high recall achieved in a short amount of time. We compared pL2Knng-a under two neighborhood update scenarios, $\gamma \in \{0, 3\}$, denoted by the subscript in the method name. Ignoring neighborhood enhancement in pL2Knng-a ($\gamma = 0$) leads to moderate recall faster than any other method. Executing even a few enhancement rounds ($\gamma = 3$) leads to almost perfect recall in pL2Knng-a in less time than either *pGF* or *NN-Descent*.

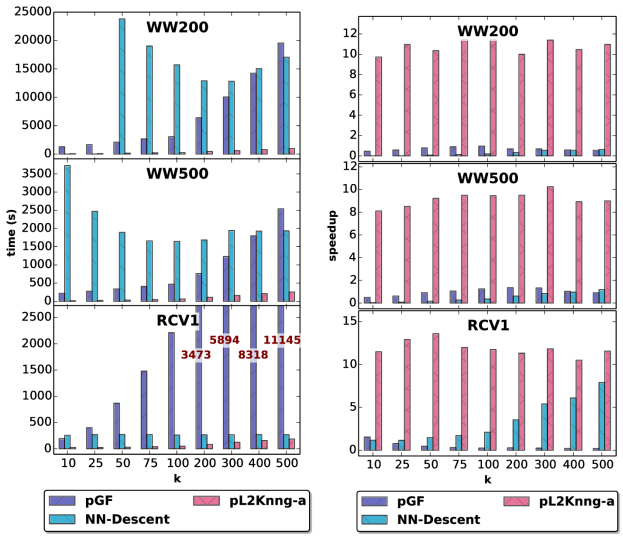


Figure 19: Approximate k-NNG construction efficiency.

Efficiency comparison. In a different experiment, we compared minimum execution times required for each method to achieve high recall (at least 95%), for k ranging from 10 to 500. We executed each method under a wide range of parameters to find its best execution time for each k value. Figure 19 shows the execution times (left) and speedups over the best serial approximate method (right) for each of the methods. Our method, pL2Knng-a, outperformed the best baseline by 1.5x – 21.7x. *NN-Descent* performed well on the RCV1 dataset, but was not competitive for the Wikipedia based datasets, likely due to high average number of non-zeros present in each vector in those datasets and the high number of similarity comparisons the method performs. *NN-Descent* was unable to find a k-NNG with high enough recall for $k \in \{10, 25\}$ for the WW200 dataset, probably due to its random choice of initial neighbors. Given its heuristic choice for initial neighbors, *pGF* performed well for small k values, but its execution time quickly increased with k due to the iterative local joins that the method performs.

6.2.2. Exact k-NNG construction

Serial improvement comparison. In order to gauge the efficiency improvements to our method that we described in Section 3.2, we compared the serial execution of our updated L2Knng variants against the original ones described in [4], for $k \in \{10, 25, 50, 75, 100\}$. We executed all methods with $\gamma = 1$ and tuned μ to achieve 95% recall for all approximate methods. Table 8 shows the results of this experiment, as speedup of the enhanced L2Knng variants. Improvements over 1.5x are presented in bold. While our updates lead to modest improvements for approximate graph construction, they contribute to achieving 1.44x – 1.73x speedup in the case of the exact version of L2Knng.

Table 8: Efficiency improvement in L2Knng.

dataset	method	k=10	25	50	75	100
WW200	L2Knng-a	1.10	1.26	1.18	1.21	1.15
WW200	L2Knng	**1.63**	**1.68**	**1.71**	**1.70**	**1.70**
WW500	L2Knng-a	1.31	1.27	1.35	1.26	1.31
WW500	L2Knng	1.49	**1.60**	**1.62**	**1.73**	**1.69**
RCV1	L2Knng-a	1.09	1.15	1.18	1.23	1.39
RCV1	L2Knng	1.46	1.50	1.49	**1.54**	1.44

Parameter sensitivity. Efficiency in the execution of our parallel method can be affected by our two parameters, the block synchronous query set size η and the inverted index block size ζ. To gauge the effects of these parameters on our algorithm execution, we tested pL2Knng on the RCV1 dataset in all combinations of $k \in \{10, 100, 500\}$, $\eta \in \{10K, 25K\}$, and $\zeta \in \{0.5M, 1M, 5M, 10M\}$. For all experiments, we chose $\gamma = 1$, and $\mu = 2k$. We present the results of this experiment in Table 9, as slowdown values compared to the $\eta = 25K$, $\zeta = 1M$ execution for each k value. The difference in performance shown in the *cmp* column for each k value is generally small, less than 1.5x slowdown in most cases, showing that our method is not greatly affected by bad choices in these parameters.

Table 9: Parameter sensitivity analysis in pL2Knng.

k=10			k=100			k=500		
η	ζ	cmp	η	ζ	cmp	η	ζ	cmp
10k	0.5M	0.98	10k	0.5M	0.99	10k	0.5M	1.15
10k	1M	1.03	10k	1M	1.02	10k	1M	1.18
10k	5M	1.60	10k	5M	1.43	10k	5M	1.42
10k	10M	1.80	10k	10M	1.54	10k	10M	1.49
25k	**0.5M**	**0.95**	**25k**	**0.5M**	**0.98**	25k	0.5M	1.14
25k	1M	1.00	25k	1M	1.00	**25k**	**1M**	**1.00**
25k	5M	1.57	25k	5M	1.41	25k	5M	1.41
25k	10M	1.77	25k	10M	1.51	25k	10M	1.49

Efficiency comparison. Figure 20 shows the efficiency comparison between pL2Knng and our efficient exact baseline, pKIdxJoin. The left side of the figure shows execution times for the methods, while the right side shows speedups of the methods over the best serial method at each k value. Our method significantly outperforms pKIdxJoin, especially for small k values. Table 10 shows the execution times for all exact and approximate methods, where parameters for approximate methods were tuned to achieve a minimum of 95% recall. Note that exact methods have 100% recall. Our exact method, pL2Knng, is more efficient than both approximate baselines for the Wikipedia datasets, and only 2.2x slower for the highest k value in the RCV1 dataset. On the other hand, our approximate method, pL2Knng-a, greatly outperforms both exact and approximate baselines.

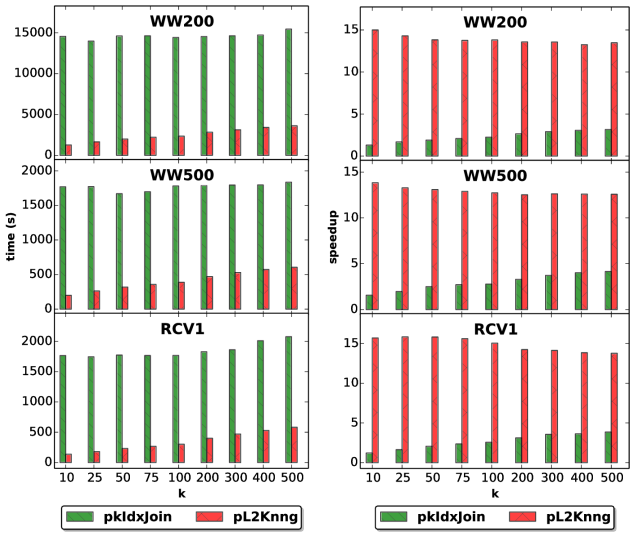


Figure 20: Exact k-NNG construction efficiency.

Strong scaling. Figure 21 shows the strong scaling characteristics of the exact methods we compared, for $k \in \{10, 100\}$. Our method scales linearly up to 16 threads, outperforming pKIdxJoin in all experiments. While pKIdxJoin also uses cache tiling, it shows decreased performance for high numbers of threads. The individual thread work unit in pKIdxJoin consists of finding the k-nearest neighbors in an index block and merging that list of neighbors with best already found k-nearest neighbors. The strategy of maintaining the k-nearest neighbors in heap data structures, combined with the cooperative neighborhood update strategy in pL2Knng, shows superior performance which is maintained even as the number of threads in increased.

Table 10: k-NNG construction efficiency comparison.

method	$k = 10$	50	100	300	500
WW200					
pKIdxJoin	14562.36	14614.19	14428.61	14632.32	15451.55
pL2Knng	1264.42	1999.10	2348.14	3120.61	3613.19
pGF	1291.51	2088.37	3043.09	10052.79	19528.90
NN-Descent	N/A	23800.08	15711.01	12807.02	17054.50
pL2Knng-a	59.51	157.12	253.31	604.02	962.25
WW500					
pKIdxJoin	1768.80	1669.78	1781.14	1793.87	1835.50
pL2Knng	199.34	318.33	387.88	528.91	604.73
pGF	217.58	337.73	470.03	1227.60	2538.85
NN-Descent	3727.96	1891.65	1645.84	1943.84	1934.60
pL2Knng-a	13.16	33.18	61.82	158.42	252.87
RCV1					
pKIdxJoin	1766.15	1774.28	1768.21	1862.52	2078.30
pL2Knng	137.22	231.52	301.52	468.87	581.71
pGF	191.71	866.42	2211.11	5894.57	11145.61
NN-Descent	254.74	271.19	261.50	265.89	268.37
pL2Knng-a	25.59	29.20	46.63	121.54	183.62

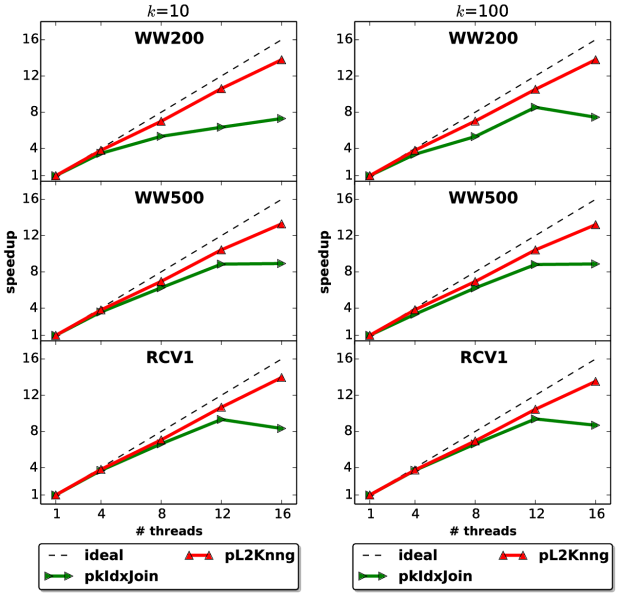


Figure 21: Strong scaling of exact k-NNG construction methods.

Load balance. As an alternate way to characterize the parallel performance of pL2Knng, we measured the load imbalance in the different sections of our method: initial graph construction (IG), graph enhancement (GE), candidate generation (CG), and candidate verification (CV). Table 11 shows the time and percent of imbalance in our experiments, for $k \in \{10, 100, 500\}$. Our method spends the majority of its time in the filtering sections (CG and CV), which display very good load balance in general, less than 1% on average. The approximate construction of the graph shows slightly worse imbalance in the IG stage, up to 12.71%. The IC stage of the method accounted for 6 – 24 % of the overall execution time in our experiments.

Table 11: Load imbalance in pL2Knng.

k	*time (s)*				*% imbalance*			
	IG	*GE*	*CG*	*CV*	*IG*	*GE*	*CG*	*CV*
RCV1								
10	33.11	1.01	99.91	32.98	1.83	1.33	0.19	0.78
100	35.98	20.51	217.67	66.11	11.28	2.23	0.07	0.34
500	175.35	84.85	359.22	98.96	12.47	5.42	0.16	0.52
WW200								
10	74.60	6.02	1176.66	125.56	0.73	0.30	0.12	0.60
100	158.57	144.79	1955.26	165.52	4.15	0.30	0.11	1.59
500	667.56	536.71	2711.16	194.48	12.71	0.98	0.14	1.67
WW500								
10	11.96	2.15	175.49	10.46	0.21	0.09	0.14	1.06
100	39.87	35.81	301.42	12.91	2.71	0.11	0.22	1.70
500	171.55	142.41	422.11	18.82	9.41	0.49	0.15	1.57

6.3. Discussion

The experiments in Section 6 show that both pL2AP and pL2Knng greatly outperform state-of-the-art baselines and scale linearly when increasing the number of processing units. For the ϵ-NNG construction problem, the threshold ϵ plays a big role in performance. For $\epsilon = 0.9$, pL2AP was able to build the neighborhood graph for the 804,414 documents in the RCV1 dataset in 3.30 seconds using 24 threads, compared to 433.42 seconds for the best parallel alternative. The success of these methods is primarily due to the aggressive *pruning of the search space*, eliminating the need to compute similarities for many pairs of objects, and discontinuing that computation for many others as soon as it is clear they will not be nearest neighbors. The best filtering in pL2AP is based on indexing a small subset of the object non-zeros, which in turn allows ignoring many candidate objects that only have features in common with the query in the non-indexed part of the vector. This strategy works especially well for high similarity thresholds, but looses some effectiveness as $\epsilon \rightarrow 0$, causing an increase in the number of candidate objects. Most of those objects are eventually pruned, yet some work must first be done to vet those candidates. The k-NNG construction problem is actually more difficult from the perspective of filtering. We have found that, even for small values of k (e.g., 1 or 5), there are some objects whose most dissimilar nearest neighbor has a very small similarity (< 0.01). In this case, to ensure correctness, most of the object's non-zeros will be indexed and many dot-products will have to be computed for the object due to the low filtering threshold.

7. Related Work

Having been studied for over a decade, the APSS problem has given rise to many serial solutions, some of which were described in Section 3. In a previous work [3], we gave an overview of existing methods and analyzed their pruning performance.

The size of data that need to be analyzed has increased dramatically in recent years, from megabytes to gigabytes (e.g., online shopping customer profiles) and terabytes (e.g., web document collections, DNA sequencing data). Traditional NNG construction methods could not scale to sets of object this large. Given the growing popularity of cloud computing, some of the traditional NNS methods were ported to cloud programming frameworks developed for dealing with big data (e.g., Hadoop, Spark) [1, 2, 14, 18, 31, 38, 39, 43, 46]. Most of the solutions use the MapReduce [20] framework and can be split into two categories. Many rely on the framework's built-in features to aggregate (reduce) partial similarities of object pairs computed in mappers [10, 19, 25, 33]. The computation efficiency can be greatly increased by first generating an inverted index for the set of objects, which can be done using one MapReduce task. The postings in the inverted index lists can then be combined with features in the object vectors or with other postings in the same list to generate partial similarity scores. While some pruning strategies can be used to avoid generating some partial scores, these methods often suffer from high communication costs which make then inefficient for large datasets [2].

The second category of MapReduce methods use a mapper-only scheme, with no reducers [1, 2, 43]. They partition the set of objects into subsets (blocks) and use serial APSS methods to find pairwise similarities of objects in block pairs. Certain block comparisons can be eliminated by relying on block-level filtering techniques, such as computing the similarity of the objects made up of the maximum values for features in the two blocks. When comparing two blocks, Alabduljalil et al. proposed locally building a full inverted index for one of the blocks and scanning through query objects in the other block to compute their similarity. They found that filtering candidates was detrimental to execution speed and suggested removing this optimization, rendering their local search identical to that performed in one tile by our naïve baseline, pIdxJoin. Within this context, they examined distributed load balancing strategies [43] and cache-conscious performance optimizations for the local searches [1]. They provided a cost based analysis aimed at finding sizes for comparison blocks that maximize cache locality. Their analysis is based on a full inverted index and mean vector and inverted list lengths, which can vary greatly in real datasets, as evidenced by the high σ values in Table 4.

Existing shared memory cosine APSS solutions are limited to the pAPT algorithm by Awekar and Samatova, detailed in Section 4. Jiang et al. [29] provided a parallel solution for the related problem of string similarity joins with edit distance constraints.

There have been few k-NNG construction algorithms that are designed to address cosine similarity. Park et al. [37] describe a heuristic serial approximate method which prioritizes computing similarities between objects with high weight features in common. In their shared memory parallel method *NN-Descent*, Dong et al. [24] follow an iterative neighborhood improvement strategy based on the intuition that similar objects may be found among the neighbors of a query object's neighbors. A number of methods have been devised for the related problem of document retrieval [12, 22, 23, 40, 42], yet our previous work [4] showed they did not work well for the graph construction task.

8. Conclusions and Future Work

In this work, we presented pL2AP and pL2Knng, our shared memory parallel solutions to the ϵ-NNG and k-NNG construction problems. Our methods use several cache-tiling optimizations, combined with fine-grained dynamically balanced parallel tasks, to solve the problems up to two orders of magnitude faster than state-of-the-art baselines. In the current work, we have focused on tiles that fit in the last-level cache. It would be interesting to evaluate strategies for maximizing the reuse of the L1 and L2 caches in similarity search. Additionally, while choosing a cache-tile size for our methods is fairly straight-forward, we may investigate designing cache-oblivious versions of the methods. Finally, we plan to investigate distributed algorithms for efficiently constructing cosine nearest neighbor graphs.

Acknowledgment

This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center (DTC) and the Minnesota Supercomputing Institute (MSI).

References

[1] Maha Alabduljalil, Xun Tang, and Tao Yang. Cache-conscious performance optimization for similarity search. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 713–722, New York, NY, USA, 2013. ACM.

[2] Maha Ahmed Alabduljalil, Xun Tang, and Tao Yang. Optimizing parallel algorithms for all pairs similarity search. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 203–212, New York, NY, USA, 2013. ACM.

[3] David C. Anastasiu and George Karypis. L2ap: Fast cosine similarity search with prefix l-2 norm bounds. In *30th IEEE International Conference on Data Engineering*, ICDE '14, 2014.

[4] David C. Anastasiu and George Karypis. L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning. In *24th ACM International Conference on Information and Knowledge Management*, CIKM '15, 2015.

[5] David C. Anastasiu and George Karypis. Pl2ap: Fast parallel cosine similarity search. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, in conjunction with SC'15*, IA3 2015, pages 1–8, New York, NY, USA, 2015. ACM.

[6] David C. Anastasiu and George Karypis. Fast parallel cosine k-nearest neighbor graph construction. In *Proceedings of the 6th Workshop on Irregular Applications: Architectures and Algorithms, in conjunction with SC'16*, IA3 2016, New York, NY, USA, 2016. ACM.

[7] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 918–929. VLDB Endowment, 2006.

[8] Amit Awekar and Nagiza F. Samatova. Fast matching for all pairs similarity search. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '09, pages 295–300, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Amit Awekar and Nagiza F Samatova. Parallel all pairs similarity search. In *Proceedings of the 10th International Conference on Information and Knowledge Engineering*, IKE '11, 2011.

[10] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. Document similarity self-join with mapreduce. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 731–736, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 131–140, New York, NY, USA, 2007. ACM.

[12] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 426–434, New York, NY, USA, 2003. ACM.

[13] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Selected papers from the sixth international conference on World Wide Web*, pages 1157–1166, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[14] Y. Chen, X. Tang, B. Liu, and D. Chen. Optimization of all pairs similarity search. In *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 637–642, Dec 2015.

[15] Evangelia Christakopoulou. Moving beyond linearity and independence in top-n recommender systems. In *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys '14, pages 409–412, New York, NY, USA, 2014. ACM.

[16] Evangelia Christakopoulou and George Karypis. Local item-item models for top-n recommendation. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pages 67–74, New York, NY, USA, 2016. ACM.

[17] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[18] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *Proc. VLDB Endow.*, 7(12):1059–1070, August 2014.

[19] G. De Francisci, C. Lucchese, and R. Baraglia. Scaling out all pairs similarity search with mapreduce. *Large-Scale Distributed Systems for Information Retrieval*, page 27, 2010.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[21] Luiz DeRose, Bill Homer, and Dean Johnson. Detecting application load imbalance on high end massively parallel systems. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Euro-Par'07, pages 150–159, Berlin, Heidelberg, 2007. Springer-Verlag.

[22] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 113–122, New York, NY, USA, 2013. ACM.

[23] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 993–1002, New York, NY, USA, 2011. ACM.

[24] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 577–586, New York, NY, USA, 2011. ACM.

[25] Tamer Elsayed, Jimmy Lin, and Douglas W. Oard. Pairwise document similarity in large collections with mapreduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, HLT-Short '08, pages 265–268, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[26] Taher H. Haveliwala, Aristides Gionis, and Piotr Indyk. Scalable techniques for clustering the web. In *In Proc. of the WebDB Workshop*, pages 129–134, 2000.

[27] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

[28] Andreas Hotho, Andreas Nürnberger, and Gerhard Paaß. A brief survey of text mining. *LDV Forum - GLDV Journal for Computational Linguistics and Language Technology*, 2005.

[29] Yu Jiang, Dong Deng, Jiannan Wang, Guoliang Li, and Jianhua Feng. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 341–348, New York, NY, USA, 2013. ACM.

[30] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, pages 247–254, New York, NY, USA, 2001. ACM.

[31] Jia-Ling Koh and Shao-Chun Peng. *A Maximum Dimension Partitioning Approach for Efficiently Finding All Similar Pairs*, pages 163–178. Springer International Publishing, Cham, 2016.

[32] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, December 2004.

[33] Jimmy Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 155–162, New York, NY, USA, 2009. ACM.

[34] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[35] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Detectives: Detecting coalition hit inflation attacks in advertising networks streams. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 241–250, New York, NY, USA, 2007. ACM.

[36] Xia Ning and George Karypis. Sparse linear methods with side information for top-n recommendations. In *Proceedings of the Sixth ACM Conference on Recommender Systems*, RecSys '12, pages 155–162, New York, NY, USA, 2012. ACM.

[37] Youngki Park, Sungchan Park, Sang-goo Lee, and Woosung Jung. Greedy filtering: A scalable algorithm for k-nearest neighbor graph construction. In SouravS. Bhowmick, CurtisE. Dyreson, ChristianS. Jensen, MongLi Lee, Agus Muliantara, and Bernhard Thalheim, editors, *Database Systems for Advanced Applications*, volume 8421 of *Lecture Notes in Computer Science*, pages 327–341. Springer-Verlag, 2014.

[38] Trong Nhan Phan, Markus Jäger, Stefan Nadschläger, Pablo Gómez-Pérez, Christian Huber, Josef Küng, and Cong An Nguyen. *TLCSim: A Large-Scale Two-Level Clustering Similarity Search with MapReduce*, pages 53–71. Springer International Publishing, Cham, 2016.

[39] Trong Nhan Phan, Markus Jäger, Stefan Nadschläger, Josef Küng, and Tran Khanh Dang. *An Efficient Document Indexing-Based Similarity Search in Large Datasets*, pages 16–31. Springer International Publishing, Cham, 2015.

[40] Cristian Rossi, Edleno S. de Moura, Andre L. Carvalho, and Altigran S. da Silva. Fast document-at-a-time query processing using two-tier indexes. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 183–192, New York, NY, USA, 2013. ACM.

[41] Mehran Sahami and Timothy D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 377–386, New York, NY, USA, 2006. ACM.

[42] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 175–182, New York, NY, USA, 2007. ACM.

[43] Xun Tang, Maha Alabduljalil, Xin Jin, and Tao Yang. Load balancing for partition-based similarity search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, pages 193–202, New York, NY, USA, 2014. ACM.

[44] Peter Willett, John M Barnard, and Geoffrey M Downs. Chemical similarity searching. *Journal of chemical information and computer sciences*, 38(6):983–996, 1998.

[45] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 131–140, New York, NY, USA, 2008. ACM.

[46] Byoungju Yang, Hyun Joon Kim, Junho Shim, Dongjoo Lee, and Sang-Goo Lee. Fast and scalable vector similarity joins with mapreduce. *J. Intell. Inf. Syst.*, 46(3):473–497, June 2016.

[47] Ying Zhao and George Karypis. Empirical and theoretical comparisons of selected criterion functions for document clustering. *Mach. Learn.*, 55(3):311–331, June 2004.