Fast Parallel Cosine K-Nearest Neighbor Graph Construction

David C. Anastasiu
San José State University
San José, CA
david.anastasiu@sjsu.edu

George Karypis
University of Minnesota, Twin Cities
Minneapolis, MN
karypis@cs.umn.edu

ABSTRACT

The k-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 for each object in the set. In contrast, we leverage shared memory parallelism and recent advances in computing similarity joins to solve the problem exactly, via a filtering based approach. Our method considers all pairs of potential neighbors but quickly filters pairs that could not be a part of the k-nearest neighbor graph, based on similarity upper bound estimates. The filtering is data dependent and not easily predicted, which poses load balance challenges in a parallel setting. We evaluated our solution on several real-world datasets and found that, using 16 threads, our method achieves up to 12.9x speedup over our exact baseline and is sometimes faster even than approximate methods. Moreover, an approximate version of our method is up to 21.7x more efficient than the best approximate state-of-the-art baseline at similar high recall. Our method displays linear strong scaling characteristics and filtering incurs less than 1% load imbalance.

Keywords

knn, nearest neighbors, similarity search, similarity join, similarity graph, cosine similarity

1. INTRODUCTION

Computing the nearest neighbor graph, or similarity graph, for a set of objects is a common task in many data analysis fields, including clustering [5, 10], online advertising [16], recommender systems [7], data cleaning [3, 21], and query refinement [4, 19]. For example, effective clustering methods [22] 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 [13]. Moreover, state-of-the-art online advertising [16] and recommender systems [6, 17] methods rely on an initially computed nearest neighbor graph 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 object feature. Similarity between objects is then computed as a function of their vector feature values. 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 [15] or text mining [12], user ratings or purchase history in recommender systems [13], and are often used to depict the structure of chemical compounds [20].

Given a set of n objects $D = \{d_1, d_2, \ldots, d_n\}$, the k-nearest neighbor graph (k-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 similarity value $\text{sim}(d_i, d_j)$ between the ith and jth objects is among the k highest values in the set $\{\text{sim}(d_i, d_l) \mid l \neq i\}$. A naïve approach to construct the nearest neighbor graph executes $O(n^2)$ object comparisons. Despite many existing works on the subject, efficient nearest neighbor graph construction algorithms addressing high dimensional sparse data are still being actively researched. In a recent work [1], we introduced `L2Knng`, a serial method that efficiently constructs the *exact* k-NNG by ignoring unimportant object pair comparisons. For each object in D, `L2Knng` considers all other objects as potential neighbors. However, most objects that are not one of the k nearest neighbors are pruned (removed from consideration) without fully computing their similarity. For a given *query* object, a *candidate* object can be pruned if an upper bound of its similarity with the query is smaller than the minimum similarity value among any of the current k closest query neighbors.

Although the filtering process in `L2Knng` works even when starting with empty neighborhoods for each object, it is more effective given higher minimum neighborhood similarities. `L2Knng` thus first identifies, for each object, k similar objects that may not be its nearest neighbors. We proposed `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.

In this work, we investigate cosine similarity 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 start our presentation by first describing serial algorithm enhancements over the initial `L2Knng` method that lead to 1.5x serial efficiency improvement. Then, we devise tiling and neighborhood update strategies that avoid locking, provide overall balanced loads for threads, and display very good strong scaling characteristics. Finally, we present evaluation results on three real-world datasets, over a large range of neighborhood sizes. Using 16 threads, our approximate method is 1.5x – 21.7x more efficient than the best approximate state-of-the-art baseline, and our exact variant achieves 3.0x – 12.9x speedup over an efficient exact baseline, while incurring less than 1% filtering imbalance.

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

2. DEFINITION & NOTATIONS

We adopt a similar notation as in [1]. 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*,

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

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 $\langle \boldsymbol{d}_i, \boldsymbol{d}_j \rangle$. 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 k nearest neighbors in D of an object d_i, 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 k-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 among the k nearest neighbors of the ith object. 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 true 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. 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.

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
$\boldsymbol{d}_i$	vector representing object d_i
$d_{i,j}$	value for jth feature in $\boldsymbol{d}_i$
$\boldsymbol{d}_i^{\leq p}, \boldsymbol{d}_i^{>p}$	prefix and suffix of $\boldsymbol{d}_i$ at dimension p
Γ_{d_i}	neighborhood for object d_i
σ_{d_i}	smallest similarity value in N_{d_i}
$\mathcal{I}$	inverted index
μ	candidate list sizes
γ	number of neighborhood enhancement updates
ϵ	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. METHODS

We will start our discussion with an analysis of `L2Knng` and present some improvements to its serial execution, and then introduce `pL2Knng`, our parallel method for cosine k-NNG construction.

3.1 Serial improvements in L2Knn

`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 `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 1 gives an overview of this process.

Algorithm 1 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 2 and 3 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 (*sort*). 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 2 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
```

Algorithm 3 Graph enhancement in L2Knng-a

```
1: function GE(D, k, μ)
2:     Create A, sparse matrix version of N̂               ▷ srt
3:     Create inverted index of A                          ▷ srt
4:     Sort vectors and inverted lists in A                ▷ 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
```

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

Table 2: 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	**80.04**	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 3.1. 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.

tasks. Table 2 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 4. 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.

3.1.1 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 decreasing 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 [11], 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, $\mathbf{A}$, such that the ith row of $\mathbf{A}$ corresponds to the k-neighborhood of the ith object. It sorts the entries in each row in non-increasing value order and selects candidates for a query object by iterating through rows in $\mathbf{A}$ associated with those objects that are the closest neighbors of the query, i.e., the column IDs of the leading entries in the sorted version of the row in $\mathbf{A}$ associated with the query. 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 decreasing 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 $\mathbf{A}$ and sorts the inverted lists in the index in decreasing value order. In our experiments, we have found reverse candidate selection rarely improves effectiveness and can often degrade GE efficiency. Thus, in pL2Knng, we do not create an inverted index for $\mathbf{A}$ and only sort its row entries.

3.1.2 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 4 describes this procedure. The function *nextList* provides the inverted list associated with the next smaller 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 4. 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 4 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 4) 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 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 decreasing 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.

3.1.3 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 the query vector in the array. Given a vector $\boldsymbol{q}$ representing the dense version of $\boldsymbol{d}_q$, the dot-product $\langle \boldsymbol{q}, \boldsymbol{c} \rangle$ 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 [2]. In our experiments, none of the new dot-product computation strategies lead to improved performance under a wide range of execution parameters.

3.2 pL2Knn

Along the enhancements presented in Section 3.1, pL2Knng uses the same filtering strategy as L2Knng. Namely, for each query object, L2Knng 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. Using the index, L2Knng selects a list of candidates for the query, which are a superset of its neighbors, a process we call *candidate generation* (CG). 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, L2Knng 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 [1].

In our parallel method, pL2Knng, threads concurrently process different query objects. 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.

3.2.1 Block processing

In order to enable cooperative processing of different query objects in its filtering phase, pL2Knng indexes objects prior to filtering. Threads can then all read the sections of the index they need 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 cache memory available on the system. The index size is highly data dependent, and each object indexes a different number of values that depends on the magnitude of those values and the current minimum similarity in the object's neighborhood. 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. As such, pL2Knng chooses the number of objects to index at a time dynamically, indexing a maximum of ϵ objects (which we call the *query set*) and ζ nonzeros at a time, where ϵ and ζ are input parameters for our method.

After indexing a block of objects, pL2Knng splits the set of query objects into blocks of size η, which is an input parameter. Threads are then dynamically assigned a small number of queries at a time from the set, 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 heap. However, it cannot also update the neighborhood of a candidate without locking, as another thread may be trying to concurrently update the same neighborhood. As such, pL2Knng keeps a candidate list in memory for each of the objects in the query set, deferring candidate neighborhood updates until all query set objects have been processed. The parameter η should thus be chosen to ensure $\eta \times \epsilon$ values can be stored in memory, as each candidate list has a maximum size of ϵ. 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 $\eta \times \max(\mu, \epsilon)$.

3.2.2 Neighborhood updates

As mentioned in the previous section, 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 [11] 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.

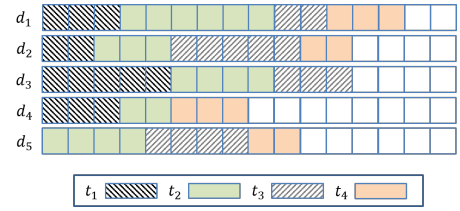

Figure 1: Segmentation of candidate neighborhood updates.

Our strategy for updating candidate neighborhoods is slightly different. Each thread is assigned a sequential block of n/nt candidate objects whose neighborhoods they are responsible to update, where nt is the number of threads. When a candidate list is constructed, candidates are added in the order they are found in the 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 nt sections s.t. the ith section contains similarities for objects in the ith candidate block, and records the starting and ending offset of each segment in the candidate list. Figure 1 shows this strategy for threads t_1–t_4, given objects d_1–d_5. This partitioning enables fast

candidate neighborhood updates at the end of each query block, as each thread only needs to traverse a subset of each candidate list to perform its required updates.

4. EXPERIMENTAL METHODOLOGY

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

4.1 Datasets

Table 3: Dataset Statistics

dataset	n	m	nnz	mrl	mcl
RCV1	804,414	45,669	62M	76.5	1347.3
WW200	1,017,531	663,419	437M	429.9	659.4
WW500	243,223	660,600	202M	830.3	305.7

For each dataset, n is the number of vectors (rows), m is the number of features (columns), nnz is the number of non-zero values, and mrl and mcl are the mean row and column lengths (number of non-zeros).

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 rows (n), columns (m), and non-zeros (nnz), and mean row/column length (mrl/mcl), are detailed in Table 3. Standard pre-processing, including tokenization, lemmatization, and *tf-idf* weighting, 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. [14].
- **WW500** contains documents with at least 500 distinct features, extracted from the October 2014 article dump of the English Wikipedia[2] (Wiki dump).
- **WW200** contains documents from the Wiki dump with at least 200 distinct features.

4.2 Baseline approaches

We compare our methods against the following baselines.

- `pKIdxJoin` is a straight-forward baseline similar to *IDX* in [18]. The method uses similar cache-tiling as `pL2Knng`, but does not use any pruning when computing similarities. For each block of queries, `pKIdxJoin` 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 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. [18]. 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. [9] 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.

4.3 Performance measures

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

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

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. [8] as,

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

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.

4.4 Execution environment

Our method and all baselines are implemented in C and compiled using gcc 5.1.0 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 HP Linux servers. Each server is a dual-socket machine, equipped with 64 Gb RAM and two eight-core 2.6 GHz Intel Xeon E5-2670 (Sandy Bridge) processors with 20 Mb Cache.

We executed each 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 $\epsilon = 100K$. We used the latest version of the *NN-Descent*[3] 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.

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

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

5. RESULTS & DISCUSSION

Our 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.

5.1 Approximate method evaluation

5.1.1 *Effectiveness comparison*

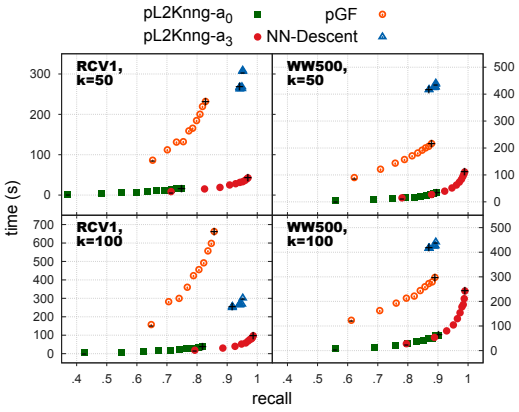

Figure 2: k-NNG construction effectiveness.

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 2 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*.

5.1.2 *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 3 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.

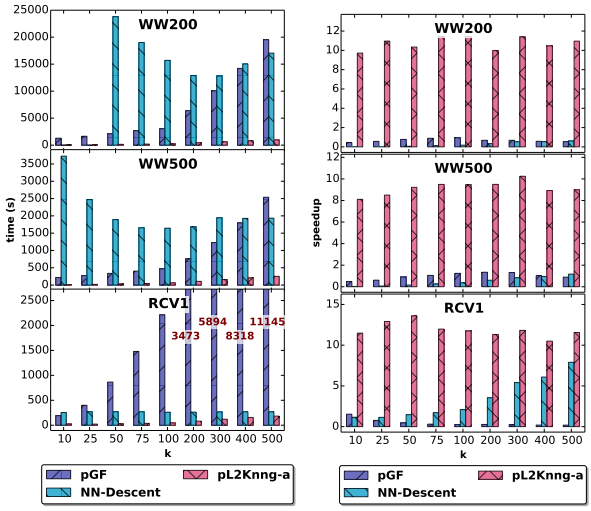

Figure 3: Approximate k-NNG construction efficiency.

5.2 Exact method evaluation

5.2.1 *Serial improvement comparison*

In order to gauge the efficiency improvements to our method that we described in Section 3.1, we compared the serial execution of our updated `L2Knng` variants against the original ones described in [1], 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 4 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 achieve 1.44x – 1.73x speedup in the case of the exact version of `L2Knng`.

Table 4: 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

5.2.2 *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 RTP 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 5, as slowdown values compared to the $\eta = 25K$, $\zeta = 1M$ execution for each k value. The variation 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.

5.2.3 *Efficiency comparison*

Figure 3 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

Table 5: 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

k value. Our method significantly outperforms pKIdxJoin, especially for small k values. Table 6 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.

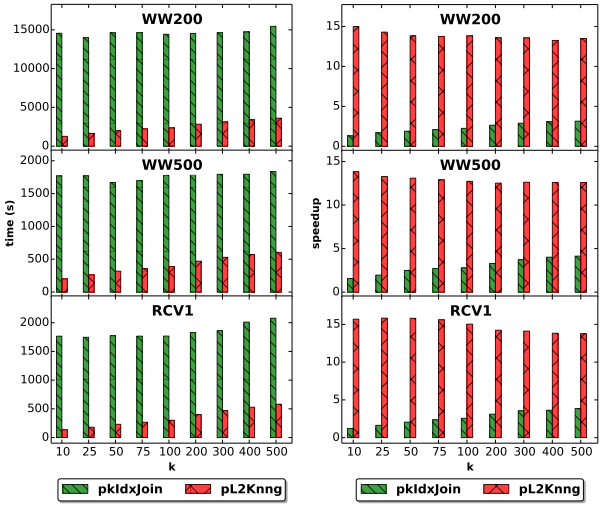

Figure 4: Exact k-NNG construction efficiency.

Table 6: 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

5.2.4 *Strong scaling*

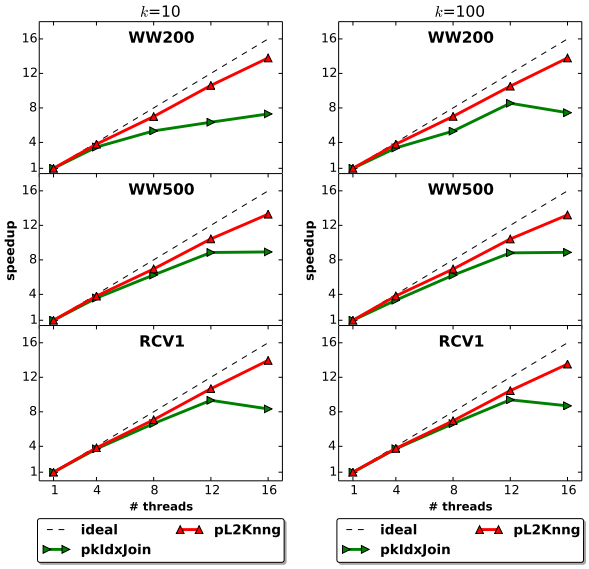

Figure 5: Strong scaling of exact k-NNG construction methods.

Figure 5 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.

5.2.5 *Load balance*

Table 7: 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

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 7 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%, which accounts for 6 – 24 % of the overall execution time.

6. CONCLUSIONS AND FUTURE WORK

In this work, we presented strategies to improve an earlier serial method for cosine similarity k-NNG constriction, and an efficient way to extract parallelism in this method, in the shared memory setting. Our method combines cache-tiling with an efficient neighborhood update strategy to solve the problem, using 16 threads, 1.5x – 21.7x faster than the best approximate and 3.0x – 12.9x faster than the best exact state-of-the-art baselines. Close analysis of execution kernels in our method revealed two potential areas of further improvement. The initial graph construction stage in our method is highly data-dependent in the choice of similarity candidates, which leads to worse load balance than other sections. Furthermore, the majority of the execution time in our method is spent computing sparse vector dot-products, and our method may benefit from alternate data structures that may further speed up this important kernel.

Acknowledgment: The author would like to thank the Graduate School at University of Minnesota for generously funding his research through the Doctoral Dissertation Fellowship. 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). We thank the reviewers for their helpful comments.

7. REFERENCES

[1] David C. Anastasiu and George Karypis. L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 791–800, New York, NY, USA, 2015. ACM.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] Youngki Park, Sungchan Park, Sang-goo Lee, and Woosung Jung. Greedy filtering: A scalable algorithm for k-nearest neighbor graph construction. In *Database Systems for Advanced Applications*, volume 8421 of *Lecture Notes in Computer Science*, pages 327–341. Springer-Verlag, 2014.

[19] 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.

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

[21] 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.

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