Fast Parallel Cosine K-Nearest Neighbor Graph Construction

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.

I. 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 $i$th and $j$th 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 L2Kng, a serial method that efficiently constructs the exact $k$-NNG by ignoring unimportant object pair comparisons. For each object in $D$, L2Kng 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 L2Kng works even when starting with empty neighborhoods for each object, it is more effective given higher minimum neighborhood similarities. L2Kng thus first identifies, for each object, $k$ similar objects that may not be its nearest neighbors. We proposed L2Kng-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

\(^1\)The method is called L2KngApprox in [1].
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 L2K\textsuperscript{nng}. We start our presentation by first describing serial algorithm enhancements over the initial L2K\textsuperscript{nng} 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.

II. DEFINITION & NOTATIONS

We adopt a similar notation as in [1]. Let \( d_i \) denote the \( i \)th of \( n \) objects in \( D \), \( d_i \in \mathbb{R}^m \) denote the feature vector in \( m \)-dimensional Euclidean space associated with the \( i \)th object, and \( d_{i,j} \) the value (or weight) of the \( j \)th feature of object \( d_i \). We measure vector similarity via the cosine function,

\[
\cos(d_i, d_j) = \frac{\sum_{j=1}^{m} d_{i,j} \times d_{j,j}}{|d_i|_2 \times |d_j|_2}.
\]

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 (\(|d_i|_2 = 1, \forall d_i \in D\)). Given that, the cosine between two vectors \( d_i \) and \( d_j \) is simply their dot-product, which we denote by \( \langle d_i, d_j \rangle \). 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 \( \Gamma_{d_i} \), 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 \( d_i \) is a directed graph \( G = (V, E) \) where vertices correspond to the objects and an edge \( (v_i, v_j) \) indicates that the \( j \)th object is among the \( k \) nearest neighbors of the \( i \)th 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 \( \sigma_{d_i} \), 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 \( \sigma_{d_i} \) 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 \( d_i \) may have less than \( k \) possible neighbors, as we ignore all null similarities and \( d_i \) may have non-zero features in common with less than \( k \) other objects in \( D \). By convention, the \( \sigma_{d_i} \) 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 I provides a summary of notation used in this work.

### Table I

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>set of objects</td>
<td>( D )</td>
</tr>
<tr>
<td>size of desired neighborhoods</td>
<td>( k )</td>
</tr>
<tr>
<td>vector representing object ( d_i )</td>
<td>( d_i )</td>
</tr>
<tr>
<td>value for ( j )th feature in ( d_i )</td>
<td>( d_{i,j} )</td>
</tr>
<tr>
<td>prefix of ( d_i ) at dimension ( p )</td>
<td>( d_{i,p}^\prec )</td>
</tr>
<tr>
<td>suffix of ( d_i ) at dimension ( p )</td>
<td>( d_{i,p}^\succ )</td>
</tr>
<tr>
<td>neighborhood for object ( d_i )</td>
<td>( \Gamma_{d_i} )</td>
</tr>
<tr>
<td>smallest similarity value ( \sigma_{d_i} ) in ( N_{d_i} )</td>
<td>( \sigma_{d_i} )</td>
</tr>
<tr>
<td>inverted index</td>
<td>( \mathcal{I} )</td>
</tr>
<tr>
<td>candidate list sizes</td>
<td>( \mu )</td>
</tr>
<tr>
<td>number of neighborhood enhancement updates</td>
<td>( \gamma )</td>
</tr>
<tr>
<td>number of objects in an inverted index tile</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>number of non-zeros in an inverted index tile</td>
<td>( \zeta )</td>
</tr>
<tr>
<td>number of objects in a query tile</td>
<td>( \eta )</td>
</tr>
</tbody>
</table>

### III. METHODS

We will start our discussion with an analysis of L2K\textsuperscript{nng} and present some improvements to its serial execution, and then introduce \( p\text{L2K}\textsuperscript{nng} \), our parallel method for cosine \textit{k}-NNG construction.

#### A. Serial improvements in L2K\textsuperscript{nng}

L2K\textsuperscript{nng} execution consists of two phases. First, in the approximate graph construction phase, L2K\textsuperscript{nng} finds an initial \( k \) neighbors for each of the objects in \( D \) by calling L2K\textsuperscript{nng}-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. L2K\textsuperscript{nng}-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 L2K\textsuperscript{nng} Algorithm

1. \textbf{function} \( L2K\textsuperscript{nng}(D, k, \gamma, \mu) \)
2. \( N \leftarrow IC(D, k, \mu) \) \quad \triangleright \text{Begin L2K}\textsuperscript{nng}-a
3. \textbf{for each} \( i = 1, \ldots, \gamma \) \textbf{do}
4. \( N \leftarrow GE(D, k, \mu, N) \) \quad \triangleright \text{End L2K}\textsuperscript{nng}-a
5. \( N \leftarrow \text{Filter}(D, k, N) \)
6. \textbf{return} \( N \)

Our serial improvements in L2K\textsuperscript{nng} focused on the approximate graph construction phase of the method. At a very high level, each of the steps in the L2K\textsuperscript{nng}-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
and each of the chosen candidates ($\sigma$ that can facilitate fast dot-product computations or pruning that may improve the current neighborhood. Data associated object, a candidate list of potential neighbors is selected ($\tau$ selection of neighbor candidates ($\sigma$) and candidate neighborhoods that contain more than two entries (one entry will be associated with those objects that are the closest to the query object). Thus, as an enhancement to $L2Knng$-a, we chose one round of neighborhood enhancements ($\gamma = 1$) and chose candidate list sizes ($\mu$) 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.

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 $\mu$ 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-$\mu$ values in each vector and inverted list. For each vector and inverted list with lengths greater than $\mu$, we first apply a select procedure [11], which partitions the list such that the leading $\mu$ values are greater or equal to the remaining values, and then sort only the leading $\mu$ 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 $\mu$ 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, $A$, such that the $i$th row of $A$ corresponds to the $k$-neighborhood of the $i$th 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 $A$ 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 $A$ associated with the query. For those query objects with less than $\mu$ candi-


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

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 III-A. 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.

Algorithm 2 Initial graph construction in $L2Knng$-a

1: function IC($D$, $k$, $\mu$)  \hfill $\triangleright$ sort
2: Create inverted index of $D$ \hfill $\triangleright$ sort
3: Sort vectors in $D$ and inverted index lists \hfill $\triangleright$ sort
4: for each $i = 1, 2, \ldots, |D|$ do
5: Choose $\mu$ candidates for the $i$th object \hfill $\triangleright$ sel
6: Hash the $i$th object \hfill $\triangleright$ ins
7: Compute similarities of $d_i$ with all $\mu$ candidates \hfill $\triangleright$ sim
8: Update $\Gamma_i$ and candidate neighborhoods \hfill $\triangleright$ upd
9: $\hat{N} = \bigcup \Gamma_i$
10: return $\hat{N}$

Algorithm 3 Graph enhancement in $L2Knng$-a

1: function GE($D$, $k$, $\mu$)  \hfill $\triangleright$ sort
2: Create $A$, sparse matrix version of $\hat{N}$ \hfill $\triangleright$ sort
3: Create inverted index of $A$ \hfill $\triangleright$ sort
4: Sort vectors and inverted lists in $A$ \hfill $\triangleright$ sort
5: for each $i = 1, 2, \ldots, |D|$ do
6: Choose $\mu$ candidates for the $i$th object \hfill $\triangleright$ sel
7: Hash the $i$th object \hfill $\triangleright$ ins
8: Compute similarities of $d_i$ with all $\mu$ candidates \hfill $\triangleright$ sim
9: Update $\Gamma_i$ and candidate neighborhoods \hfill $\triangleright$ upd
10: $\hat{N} = \bigcup \Gamma_i$
11: return $\hat{N}$

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 tasks. Table II 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 IV. In each of the experiments, we only executed one round of neighborhood enhancements ($\gamma = 1$) and chose candidate list sizes ($\mu$) 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.
dates 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 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. In pL2Knng, we skip this optimization, do not create an inverted index for A, and only sort its row entries.

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 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 \( \langle q, a \rangle \) in Algorithm 4. Here, \( a \) is the prefix of 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 4 Candidate selection in the IC phase of L2Knng-a

1: function SELECTCANDIDATESIC(D, q, \( \mu \))
2: \( A \leftarrow \text{nextList}(q), B \leftarrow \text{nextList}(q), C = \emptyset \)
3: while \( |C| < \mu \) and \( A \neq \emptyset \) and \( B \neq \emptyset \) do
4: if \( A = \emptyset \) or \( B = \emptyset \) then
5: Choose candidates only from the remaining list
6: \( a \leftarrow \text{nextCand}(A), b \leftarrow \text{nextCand}(B) \)
7: if \( \langle q, a \rangle \geq \langle q, b \rangle \) then
8: \( C \leftarrow C \cup a \)
9: \( A \leftarrow A \setminus a \)
10: \( A \leftarrow \text{nextList}(q) \) if \( A = \emptyset \)
11: else
12: \( C \leftarrow C \cup b \)
13: \( B \leftarrow B \setminus b \)
14: \( B \leftarrow \text{nextList}(q) \) if \( B = \emptyset \)
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 \( d_f(A) \times d_a.f(A) < d_f(B) \times d_b.f(B) \), 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) Query insertion and similarity computation: Since L2Knng-a computes the similarity of a query vector with many (namely, \( \mu \)) 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 q representing the dense version of \( d_q \), the dot-product \( \langle q, c \rangle \) can be computed as,

\[
\text{for each } j = 1, \ldots, m \text{ s.t. } d_{c,j} \geq 0 \text{ do} \\
\quad s \leftarrow s + d_{c,j} \times q_j
\]

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.

B. pL2Knng

Algorithm 5 describes our parallel k-NNG construction method, pL2Knng. Our method follows the same computation strategy as L2Knng, incorporating the improvements described in Section III-A. 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. Using the index, pL2Knng 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, 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 [1].

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 pL2Knn.

1) Block processing: In order to enable cooperative processing of different query objects in its filtering phase, pL2Knn indexes objects prior to filtering. The index is split into several sections, called 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 same 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. 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 approximation 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 in pL2Knn such that the tile contains at most $\epsilon$ indexed objects and $\zeta$ indexed non-zeros.

The processing order of objects in pL2Knn is in non-increasing value of their minimum neighborhood similarity $\sigma_{d_i}$. Taking advantage of the commutative property of cosine similarity, pL2Knn only compares a query object against objects that come before it in the object processing order. After completing the filtering process for all subsequent objects in the processing order using the current index, the index can be discarded. The filtering leads to improved minimum neighborhood similarities of un-indexed objects in the neighborhood graph represented by $\mathcal{N}$. As a result, pL2Knn then updates the object processing order of un-indexed objects, improving index reduction and pruning during searches using the following index tile.

After indexing a set of objects, pL2Knn splits the set of query objects (those that come after the first indexed object in the processing order) into query blocks of size $\eta$. Threads are then dynamically assigned a small number of consecutive queries at a time from a block, 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, pL2Knn 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 $\eta$ should be chosen to ensure $\eta \times \epsilon$ values can be stored in memory, as each candidate list has a maximum size of $\epsilon$. Moreover, moderately small $\eta$ 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 $\mu$ there, so the memory necessary to store candidates is $\eta \times \max(\mu, \epsilon)$.

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 $\mathcal{C}$ with $|\mathcal{C}| > k$, we first select [11] the top-$k$ values in the list, filtering out those less than $\sigma_q$, the current minimum similarity in the query neighborhood, and then sequentially insert them in the query heap.

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 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 $\sigma_i$ of the candidate neighborhood and discarded if it cannot improve that neighborhood. The thread

![Fig. 1. Segmentation of candidate lists for neighborhood updates.](image-url)
then partitions the remaining values into \( p \) sections s.t. the \( i \)th section contains similarities for objects in the \( i \)th candidate block, which will be updated by the \( i \)th 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 1 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, as each thread only needs to traverse a subset of each candidate list to perform its required updates.

IV. Experimental Methodology

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

A. Datasets

TABLE III

<table>
<thead>
<tr>
<th>dataset</th>
<th>( n )</th>
<th>( m )</th>
<th>( nnz )</th>
<th>( mrl )</th>
<th>( mcl )</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCV1</td>
<td>804,414</td>
<td>45,669</td>
<td>62M</td>
<td>76.5</td>
<td>1347.3</td>
</tr>
<tr>
<td>WW2000</td>
<td>1,017,531</td>
<td>663,419</td>
<td>437M</td>
<td>429.9</td>
<td>659.4</td>
</tr>
<tr>
<td>WW5000</td>
<td>243,223</td>
<td>660,600</td>
<td>202M</td>
<td>830.3</td>
<td>305.7</td>
</tr>
</tbody>
</table>

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 (\( mrl \) and \( mcl \)), are detailed in Table III. 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].
- **WW2000** contains documents with at least 500 distinct features, extracted from the October 2014 article dump of the English Wikipedia\(^2\) (Wiki dump).
- **WW5000** contains documents from the Wiki dump with at least 200 distinct features.

B. Baseline approaches

We compare our methods against the following baselines.

- **pIxIdxJoin** is a straightforward 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 set of queries, pIxIdxJoin 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. [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 \( \mu \) 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 \( \geq 10 \). 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.

C. 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,

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

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,

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

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.
D. 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 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.

V. 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.

A. Approximate method evaluation

1) Effectiveness comparison: As a way to compare the effectiveness of the approximate methods, we executed each for \( \mu \in \{1k, \ldots, 10k\} \), where \( \mu \) 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.

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.

B. Exact method evaluation

1) Serial improvement comparison: In order to gauge the efficiency improvements to our method that we described in Section III-A, 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 \( \mu \) to achieve 95% recall for all approximate

---

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

---
methods. Table IV 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 IV
#### Efficiency Improvement in L2Knng

<table>
<thead>
<tr>
<th>dataset</th>
<th>method</th>
<th>k=10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>WW200</td>
<td>L2Knng-a</td>
<td>1.10</td>
<td>1.26</td>
<td>1.18</td>
<td>1.27</td>
<td>1.15</td>
</tr>
<tr>
<td>WW200</td>
<td>L2Knng</td>
<td>1.63</td>
<td>1.68</td>
<td>1.71</td>
<td>1.70</td>
<td>1.70</td>
</tr>
<tr>
<td>WW500</td>
<td>L2Knng-a</td>
<td>1.31</td>
<td>1.27</td>
<td>1.35</td>
<td>1.26</td>
<td>1.31</td>
</tr>
<tr>
<td>WW500</td>
<td>L2Knng</td>
<td>1.49</td>
<td>1.60</td>
<td>1.62</td>
<td>1.73</td>
<td>1.69</td>
</tr>
<tr>
<td>RCV1</td>
<td>L2Knng-a</td>
<td>1.09</td>
<td>1.15</td>
<td>1.18</td>
<td>1.23</td>
<td>1.39</td>
</tr>
<tr>
<td>RCV1</td>
<td>L2Knng</td>
<td>1.46</td>
<td>1.50</td>
<td>1.49</td>
<td>1.54</td>
<td>1.44</td>
</tr>
</tbody>
</table>

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 RCV1 dataset in all combinations of k ∈ {10, 100, 500}, η ∈ {10K, 25K}, and ζ ∈ {0.5M, 1M, 5M, 10M}. For all experiments, we chose γ = 1, and µ = 2k. We present the results of this experiment in Table V, as slowdown values compared to the η = 25K, ζ = 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 V
#### Parameter Sensitivity Analysis in pL2Knng

<table>
<thead>
<tr>
<th>k=10</th>
<th>η</th>
<th>ζ</th>
<th>cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>10k</td>
<td>0.5M</td>
<td>0.98</td>
<td>10k</td>
</tr>
<tr>
<td>10k</td>
<td>1M</td>
<td>1.03</td>
<td>10k</td>
</tr>
<tr>
<td>10k</td>
<td>5M</td>
<td>1.43</td>
<td>10k</td>
</tr>
<tr>
<td>10k</td>
<td>10M</td>
<td>1.54</td>
<td>10k</td>
</tr>
<tr>
<td>25k</td>
<td>0.5M</td>
<td>0.95</td>
<td>25k</td>
</tr>
<tr>
<td>25k</td>
<td>1M</td>
<td>1.00</td>
<td>25k</td>
</tr>
<tr>
<td>25k</td>
<td>5M</td>
<td>1.41</td>
<td>25k</td>
</tr>
<tr>
<td>25k</td>
<td>10M</td>
<td>1.51</td>
<td>25k</td>
</tr>
</tbody>
</table>

3) Efficiency comparison: Figure 4 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 VI 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.

4) Strong scaling: Figure 5 shows the strong scaling characteristics of the exact methods we compared, for k ∈ {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) 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 VII shows the time and percent of imbalance in our experiments, for k ∈ {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.

VI. CONCLUSIONS AND FUTURE WORK

In this work, we presented strategies to improve an earlier serial method for cosine similarity k-NNG construction, and an efficient way to extract parallelism in this method, in the shared memory setting. Our method combines caching with an efficient neighborhood update strategy to solve the problem, using 16 threads, 3.0x – 12.9x faster than the best exact state-of-the-art baselines. Moreover, an approximate version of our method outperforms the best approximate state-of-the-art baseline by 1.5x – 21.7x. 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: 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, the Digital Technology Center at the University of Minnesota, and the Graduate School at University of Minnesota through the Doctoral Dissertation Fellowship program. 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.

REFERENCES


