November, 2016

Fast Parallel Cosine K-Nearest Neighbor Graph Construction

David C. Anastasiu, *San Jose State University*
George Karypis, *University of Minnesota - Twin Cities*

Fast Parallel Cosine K-Nearest Neighbor Graph Construction

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

George Karypis
University of Minnesota, Twin Cities
Minneapolis, MN, USA
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 $\mathbf{n^2}$ similarities for a set of n objects. This has led researchers to seek approximate methods, which find many but not all of the nearest neighbors. In contrast, we leverage shared memory parallelism and recent advances in similarity joins to solve the problem *exactly*, via a filtering based approach. Our method considers all pairs of potential neighbors but quickly filters those that could not be a part of the k-nearest neighbor graph, based on similarity upper bound estimates. 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.**

I. Introduction

Computing the nearest neighbor graph for a set of objects is a common task in many data analysis fields, including clustering, online advertising, recommender systems, and data cleaning. 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 vectors. 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 $\{\text{sim}(d_i, d_l) \mid l \neq i\}$. 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. To improve pruning effectiveness, `L2Knng` first identifies, for each object, k similar objects that may not be its nearest neighbors. We proposed `L2Knng-a` for this task, a fast *approximate graph construction* method that 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 graph construction is data dependent and not easily predicted, which poses load balance challenges. Furthermore, marshaling neighborhood updates may cause contention in both the initial approximate graph construction and the exact filtering phases of `L2Knng`. We address these issues by devising tiling and neighborhood update strategies that avoid locking, provide overall balanced loads for threads, and display very good strong scaling characteristics. We evaluate the effectiveness of our strategy on three real-world datasets, over a large range of neighborhood sizes. Results show that, 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.

IA^3 2016; Salt Lake City, Utah, USA; November 2016
978-1-5090-3867-1/16/$31.00 ©2016 IEEE

TABLE I
Notation used throughout the work

Symbol	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
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

II. Definition & notations

We adopt a similar notation as in [1], which is summarized in Table I. 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 $\langle \boldsymbol{d}_i, \boldsymbol{d}_j \rangle$. We denote by the *minimum (neighborhood) similarity* σ_{d_i} the minimum similarity between object d_i and one of its current k neighbors. An *inverted index* 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.

III. Methods

We will first introduce `L2Knng` and present some improvements to its serial execution, and then present `pL2Knng`, our parallel method for cosine k-NNG construction.

TABLE II
APPROXIMATE GRAPH CONSTRUCTION PERCENT TIMES

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

The table shows, for the initial graph construction phase of the L2Knng-a method, the percent of execution time of different tasks within the algorithm.

A. Serial improvements in L2Knng

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.

Our serial improvements in L2Knng focused on the approximate graph construction phase of the method, which itself consists of an initial construction (IC) phase and a graph enhancement phase (GE). At a high level, each of the steps in the L2Knng-a execution is composed of the following tasks. Input data or the current neighborhoods are sorted by value and indexed to facilitate the selection of neighbor candidates (*srt*). Then, for each query object, a candidate list of potential neighbors is selected (*sel*) that may improve the current neighborhood. Data associated with the query object is optionally entered into a data structure that can facilitate fast dot-product computations or pruning (*ins*). Then, dot-products are computed between the query and each of the chosen candidates (*sim*), skipping some pruned candidates. Finally, some of the neighborhoods are updated (*upd*) with computed similarities that improve them.

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 during the initial graph construction, when searching for 10, 100, and 500 nearest neighbors in two datasets described in Section IV. 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.

Given these observations, we focused our efforts on improving the similarity computation, sorting, and candidate selection tasks in L2Knng-a. We improved our sort routines by first applying a select procedure [2], which partitions the lists to be sorted such that the leading μ values are greater or equal to the remaining values, which reduced the complexity of sorting a list of size $l > \mu$ from $O(l \log l)$ to $O(l + \mu \log \mu)$. We simplified the candidate selection process by eliminating reverse neighbor selection and prefix dot-product computations for the purpose of identifying candidates. We improved the neighborhood update efficiency by using a select procedure to reduce the number of neighborhood heap updates from μ to k. Although we attempted several strategies to improve vector dot-products, none proved more efficient than the sparse-dense dot-product strategy used in L2Knng-a. The details of these changes can be found in [3].

TABLE III
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

Table III shows the results of comparing the serial execution of our updated L2Knng variants against the original ones described in [1], as speedup of the enhanced L2Knng variants, for $k \in \{10, 25, 50, 75, 100\}$. We executed all methods with one round of neighborhood enhancement ($\gamma = 1$) and tuned the candidate list size μ to achieve 95% recall for all approximate methods. Improvements over 1.5x are presented in bold. While our updates led 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, which is affected by both the efficiency and effectiveness of L2Knng-a.

B. PL2Knng

Algorithm 1 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). 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 estimates, and updates the query and candidate neighborhoods if the result can enhance them. For full details on the filtering process, see [1].

1) Block processing: In order to enable cooperative processing of different query objects in its filtering phase, pL2Knng indexes objects prior to filtering. The index is split into several 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. 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.

Algorithm 1 The pL2Knng algorithm.

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

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 size of each tile is thus dynamically chosen in pL2Knng such that the tile contains at most ϵ indexed objects and ζ indexed non-zeros.

After indexing a set of objects, pL2Knng splits the set of query objects (those that come after the first indexed object in the processing order) into query blocks of size η. Threads are then dynamically assigned a small number of consecutive queries at a time 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, pL2Knng keeps a candidate list in memory for each of the objects in the query block, deferring candidate neighborhood updates until all query block objects have been processed. The parameter η should be chosen to ensure $\eta \times \epsilon$ 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 block 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 $\eta \times \max(\mu, \epsilon)$.

The processing order of objects in pL2Knng is in non-increasing value of their minimum neighborhood similarity σ_{d_i}. After completing the filtering process using the current index, the index can be discarded. The filtering, however, leads to improved minimum neighborhood similarities of un-indexed objects. As a result, pL2Knng updates the object processing order of un-indexed objects, improving index reduction and pruning during searches using the following index tile.

2) Neighborhood updates: Each thread updates the query neighborhood as soon as it has finished the filtering process

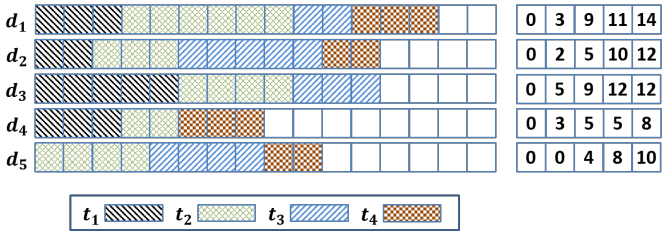

Fig. 1. Segmentation of candidate lists for neighborhood updates.

TABLE IV
DATASET STATISTICS

dataset	# objects	# features	# non-zeros
RCV1	804,414	45,669	62M
WW200	1,017,531	663,419	437M
WW500	243,223	660,600	202M

for its list of candidates. Given a set of candidates C with $|C| > k$, we first select [2] the top-k values in the list, filtering out those less than σ_q, 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, the thread re-arranges the similarities in the candidate list to ensure efficient candidate list updates. Filtering out similarities that cannot improve their respective candidate neighborhoods, the thread partitions similarities into p sections s.t. the ith section contains similarities for objects in the ith candidate block, which will be updated by the ith thread after the query block has finished being processed. The thread also records the starting and ending offset of each segment in the candidate list. Figure 1 shows this strategy for objects d_1–d_5 from a set of 16 objects, given 4 threads.

IV. Experimental Methodology

Table IV details the number of objects, features, and non-zeros of the text-based datasets we used in our experiments. The RCV1 [4] dataset is a standard benchmark corpus of newswire stories provided by Reuters, Ltd., while WW200 and WW500 contain documents with at least 200 and 500 distinct features, respectively, extracted from the October 2014 article dump of the English Wikipedia.

We compare our methods against three baselines. PKIdxJoin is a straight-forward baseline that uses similar cache-tiling as pL2Knng, but does not use any pruning when computing similarities. *GF* is an approximate k-NNG construction method proposed by Park et al. [5]. We have created a shared memory parallel version of *GF*, which we call *pGF*, using the same thread cooperation strategy as in pL2Knng-a. Finally, *NN-Descent* is a shared memory parallel approximate k-NNG construction method designed by Dong et al. [6] to work with generic similarity measures. Additional

details on baselines, performance measures, and execution environment can be found in [3].

V. Results & Discussion

Due to lack of space, we only present experiment results on the efficiency of our approximate and exact methods, and summarize strong scaling and load balance results. Additional results on these topics, along with effectiveness and parameter sensitivity experiments, are presented in [3].

A. Efficiency comparison

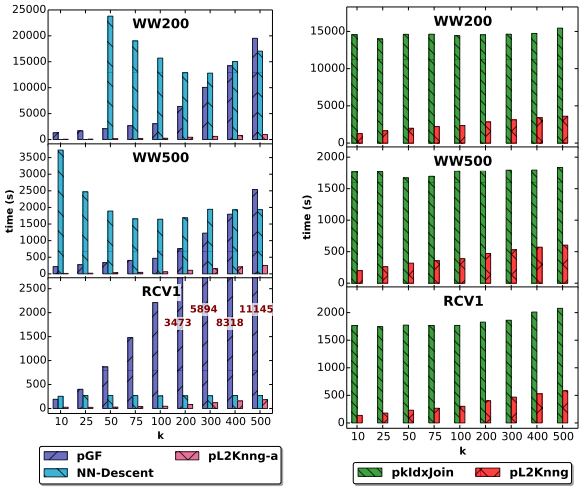

Fig. 2. Approximate k-NNG construction efficiency.

We compared minimum execution times required for each approximate method to achieve high recall (at least 95%), and for each exact method to solve the problem, for k ranging from 10 to 500. We executed each approximate method under a wide range of parameters to find its best execution time for each k value. Figure 2 shows the execution times for approximate methods (left) and exact methods (right). Our approximate method, `pL2Knng-a`, outperformed the best approximate baseline by 1.5x – 21.7x. *NN-Descent* performed well on the RCV1 dataset, but was not competitive for the Wikipedia based datasets. It 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. Our exact method significantly outperformed `pKIdxJoin`, especially for small k values. `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 outperformed both exact and approximate baselines.

B. Strong scaling and load balance

We compared the scaling characteristics of the exact methods for $k \in \{10, 100\}$. Our method scales linearly up to 16 threads, outperforming `pKIdxJoin` in all experiments. While `pKIdxJoin` achieves less than 8x speedup over its single threaded execution using 16 threads, `pL2Knng` is able to achieve almost 14x speedup over its single threaded execution.

TABLE V
Load imbalance in `pL2Knng`

	RCV1				WW200				WW500			
k	*IG*	*GE*	*CG*	*CV*	*IG*	*GE*	*CG*	*CV*	*IG*	*GE*	*CG*	*CV*
10	0.2	0.1	0.1	1.1	0.7	0.3	0.1	0.6	1.8	1.3	0.2	0.8
100	2.7	0.1	0.2	1.7	4.2	0.3	0.1	1.6	11.3	2.2	0.1	0.3
500	9.4	0.5	0.2	1.6	12.7	1.0	0.1	1.7	12.5	5.4	0.2	0.5

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 V shows the 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, which accounts for 6 – 24 % of the overall execution time, shows slightly worse imbalance in the IG stage, up to 12.71%.

VI. Conclusion

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 exact and approximate methods combine cache-tiling with an efficient neighborhood update strategy to solve the problem, using 16 threads, 3.0x – 12.9x faster than the best exact and 1.5x – 21.7x faster than the best approximate state-of-the-art baselines.

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).

References

[1] D. C. Anastasiu and G. 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*, ser. CIKM '15. New York, NY, USA: ACM, 2015, pp. 791–800.

[2] C. A. R. Hoare, "Algorithm 65: Find," *Commun. ACM*, vol. 4, no. 7, pp. 321–322, Jul. 1961.

[3] D. C. Anastasiu and G. Karypis, "Fast Parallel Cosine K-Nearest Neighbor Graph Construction," Department of Computer Engineering, San José State University, San José, CA, USA, Tech. Rep., 2016, http://davidanastasiu.net/pdf/papers/2016-AnastasiuK-IA3-pl2knn.pdf, accessed October 2016.

[4] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *J. Mach. Learn. Res.*, vol. 5, pp. 361–397, Dec. 2004.

[5] Y. Park, S. Park, S.-g. Lee, and W. Jung, "Greedy filtering: A scalable algorithm for k-nearest neighbor graph construction," in *Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2014, vol. 8421, pp. 327–341.

[6] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 577–586.