Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 15-017

PL2AP: Fast Parallel Cosine Similarity Search

David C. Anastasiu, George Karypis

November 16, 2015

PL2AP: Fast Parallel Cosine Similarity Search*

David C. Anastasiu
University of Minnesota, Twin Cities
Minneapolis, USA
dragos@cs.umn.edu

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

ABSTRACT

Solving the *AllPairs similarity search* problem entails finding all pairs of vectors in a high dimensional sparse dataset that have a similarity value higher than a given threshold. The output form this problem is a crucial component in many real-world applications, such as clustering, online advertising, recommender systems, near-duplicate document detection, and query refinement. A number of serial algorithms have been proposed that solve the problem by pruning many of the possible similarity candidates for each query object, after accessing only a few of their non-zero values. The pruning process results in unpredictable memory access patterns that can reduce search efficiency. In this context, we introduce `pL2AP`, which efficiently solves the AllPairs cosine similarity search problem in a multi-core environment. Our method uses a number of cache-tiling optimizations, combined with fine-grained dynamically balanced parallel tasks, to solve the problem 1.5x–232x faster than existing parallel baselines on datasets with hundreds of millions of non-zeros.

Keywords

similarity search, similarity join, bounded cosine similarity graph, cosine similarity

1. INTRODUCTION

Given a set of objects, AllPairs similarity search (APSS) finds, for each object in the set, all other *similar objects*, those with a similarity value above a certain threshold ϵ. The output of APSS is a crucial component in many applications, including clustering [8,13], online advertising [19], recommender systems [9], near-duplicate document detection [23], and query refinement [7, 21]. Executing a search naïvely over a set of n objects requires $O(n^2)$ object comparisons. To solve this challenging problem, recently proposed serial APSS solutions [3, 4, 7, 16] rely on theoretic similarity upper bounds to stop comparing a pair of objects as soon as it is clear their similarity cannot reach the desired threshold ϵ. Parallel versions of these algorithms could further speed up computation. However, the active pruning of the search space in serial APSS methods is highly data dependent and results in unpredictable memory access patterns, making their effective parallelization non-trivial.

Parallel APSS solutions target either multi-core or distributed systems. Most approaches use MapReduce [11] to distribute the computation [6, 10, 12, 18], using an inverted index data structure and performing analogous pruning as in serial methods. However, these methods suffer from high communication costs which make them inefficient for large datasets [2]. Partition based MapReduce methods [1, 2, 22] address this problem via block data decomposition, using serial APSS methods on MapReduce nodes to compute pairwise similarities between objects in block pairs. These methods could further benefit from multi-core parallel APSS solutions, which are not prevalent in the literature.

In this work, we address multi-core parallel solutions for the exact APSS problem using cosine similarity as a way to compare objects. Awekar and Samatova [5] provide the only existing multi-core parallel algorithm to solve this problem, which we call `pAPT`. Their method is based on an existing serial APSS algorithm they developed, `APT` [4], and uses *index sharing* as a way to allow threads to execute independent searches. In essence, an inverted index is pre-computed and shared among the threads, while each thread keeps and updates its own version of index meta-data to avoid synchronization overheads. The authors devise and test three load balancing strategies, and find that both dynamic and round-robin task assignments perform similarly.

The APSS parallelization strategy of Awekar and Samatova does not take into account the available memory hierarchy in current systems, and can lead to slow performance due to thrashing when searching large datasets. As each thread traverses portions of the inverted index associated with their own query object, they likely evict the working data of other threads from cache. With this in mind, we design a new multi-core parallel algorithm, `pL2AP`, which uses a number of cache-tiling optimizations, combined with fine-grained dynamically balanced parallel tasks, to solve the APSS problem, using 24 threads, 1.5x–232x faster than parallel baselines and 2x–34x faster than the fastest serial method on datasets with hundreds of millions of non-zeros.

The remainder of the paper is organized as follows. Section 2 introduces the problem and notation used throughout the paper. Section 3 details the serial APSS computation framework, while Section 4 presents parallel solutions to the problem. We describe our evaluation methodology and analyze experimental results in Sections 5 and 6. Section 7 summarizes related works, and Section 8 concludes the paper.

2. DEFINITION & NOTATIONS

Let $D = \{d_1, d_2, \ldots, d_n\}$ be a set of objects such that each object d_i is a (sparse) vector in an m dimensional feature space. We will use d_i to indicate the ith object, $\boldsymbol{d}_i$ to indicate the feature vector associated with the ith object, and $d_{i,j}$ to indicate the value (or weight) of the jth feature of object d_i.

The AllPairs similarity search problem seeks, for each object d_i in D, all other objects d_j such that $\text{sim}(d_i, d_j) \geq \epsilon$. We use the *cosine function* to measure vector similarity. To simplify the pre-

*This technical report contains additional experiment results and some clarifications that were not included in the original PL2AP paper published in the IA^3 workshop.

sentation of the algorithms, 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 similarity between two vectors $\boldsymbol{d}_i$ and $\boldsymbol{d}_j$ is simply their dot-product, which we denote by $\langle \boldsymbol{d}_i, \boldsymbol{d}_j \rangle$.

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. List I_j contains pairs $(d_i, d_{i,j})$, also called postings, where d_i is an indexed object that has a non-zero value for feature j and $d_{i,j}$ is that value. Postings may store additional information, such as the position of the feature in the given document or other statistics.

Given a vector $\boldsymbol{d}_i$ and a dimension p, we will denote by $\boldsymbol{d}_i^{\leq p}$ the vector $(d_{i,1}, \ldots, d_{i,p}, 0, \ldots, 0)$, obtained by keeping the p leading dimensions in $\boldsymbol{d}_i$, which we call the *prefix* (vector) of $\boldsymbol{d}_i$. We refer to $\boldsymbol{d}_i^{>p} = (0, \ldots, 0, d_{i,p+1}, \ldots, d_{i,m})$ as the *suffix* of $\boldsymbol{d}_i$, obtained by setting the first p dimensions of $\boldsymbol{d}_i$ to 0. Similarly, $\boldsymbol{d}_i^{<p} = \boldsymbol{d}_i^{\leq p-1}$ denotes the prefix keeping dimensions up to, but not including, p, and $\boldsymbol{d}_i^{\geq p} = \boldsymbol{d}_i^{>p-1}$ denotes the suffix keeping dimensions starting at p, inclusive. One can then verify that

$$\begin{aligned} \boldsymbol{d}_i &= \boldsymbol{d}_i^{\leq p} + \boldsymbol{d}_i^{>p}, \\ \boldsymbol{d}_i &= \boldsymbol{d}_i^{<p} + \boldsymbol{d}_i^{\geq p}, \\ ||\boldsymbol{d}_i||_2^2 &= ||\boldsymbol{d}_i^{\leq p}||_2^2 + ||\boldsymbol{d}_i^{>p}||_2^2, \text{ and} \\ \langle \boldsymbol{d}_i, \boldsymbol{d}_j \rangle &= \langle \boldsymbol{d}_i, \boldsymbol{d}_j^{\leq p} \rangle + \langle \boldsymbol{d}_i, \boldsymbol{d}_j^{>p} \rangle. \end{aligned}$$

Given an object processing order, let $\mathbf{D}$ represent the sparse matrix made up by stacking all object vectors in that order, such that vector $\boldsymbol{d}_i$ is the ith row in $\mathbf{D}$. We keep $\mathbf{D}$ in memory in Compressed Sparse Row (CSR) format. Similarly, an inverted index containing all values in $\mathbf{D}$ is equivalent to a sparse representation of $\mathbf{D}^T$. Here, T represents the matrix transpose operation. We keep the index in memory as a Compressed Sparse Column (CSC) matrix. Table 1 provides a summary of notation used in this work.

Table 1: Notation used throughout the work

	Description
D	set of objects
d_i	the ith object
$\boldsymbol{d}_i$	vector representing the ith object
$d_{i,j}$	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
$\boldsymbol{d}_i^{\leq}, \boldsymbol{d}_i^{>}$	un-indexed/indexed portion of $\boldsymbol{d}_i$
$d_{<i}, d_{>i}$	any object coming before/after i in the processing order
$\mathbf{D}$	sparse matrix representation of objects in D
n, m	number of objects/features in D
$\mathcal{I}$	inverted index
ϵ	minimum desired similarity
η	size of a bulk synchronous query block
ζ	maximum number of inverted index non-zeros
α	masked hash-table size exponent

3. SERIAL ALGORITHMS

Most serial APSS solutions follow a similar computation framework, first introduced by Bayardo et al. [7]. The main idea in the framework is to decompose the computation of $\mathbf{DD}^T$, which finds all pairwise similarities for objects in D, into

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

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

Algorithm 1 The AllPairs Framework

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

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

In the remainder of this section, we highlight the pruning choices in the `APT` algorithm by Awekar and Samatova [4] and in the `L2AP` algorithm in our previous work [3], on which the parallel algorithms described in Section 4 are based. Additionally, we analyze memory access patterns inherent in the computations in each stage of the framework. Table 2 provides a quick reference for these pruning choices.

3.1 Indexing

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

Algorithm 2 Indexing in the AllPairs Framework

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

While it improves computation efficiency by limiting the number of non-zeros traversed when identifying neighbors for a query object, the *partial indexing* of only suffix values in each query object is also an effective pruning strategy. Note that some objects may

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

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

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

not have any features in common with the query suffix. These objects are automatically removed from consideration, without even starting to compare them to the query.

APT computes the prefix similarity estimate $\text{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{>q})$, which we call the idx bound, as the dot product between the query vector and the *max vector*, the vector made up of all maximum feature values, denoted as $\boldsymbol{mx}$. The estimate is improved by processing objects in decreasing order of their maximum feature weights, and bounding the max vector by the maximum feature weight in the query,

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

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

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

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

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

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

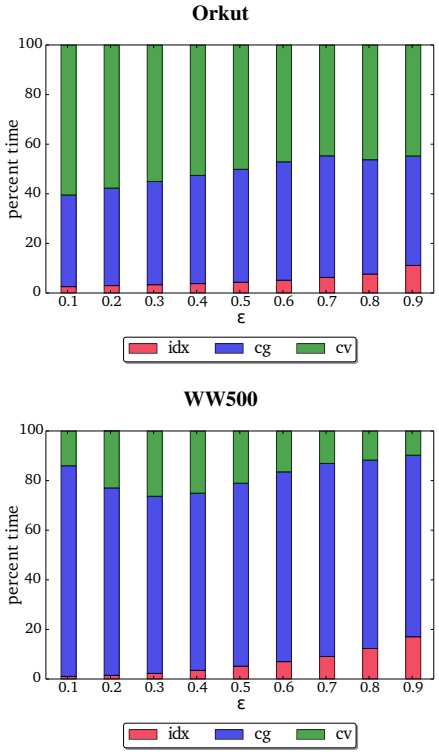

Figure 1: Percent execution times for the Orkut and WW500 datasets. For each dataset, the stacked bars show the percent of search time taken by the indexing (*idx*), candidate generation (*cg*), and candidate verification (*cv*) phases in L2AP, for similarity thresholds ranging from 0.1 to 0.9.

3.2 Candidate generation

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

Algorithm 3 Candidate Generation in the AllPairs Framework

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

data structure that accumulates values for given keys) is used to keep track of partial dot-products between the query and encountered objects. Once accumulation has started for an object, it becomes a *candidate*.

Accumulation is prevented for a new object in two ways. First, the size of the candidate vector (number of non-zeros) is checked against a minimum size estimate, which we call the *size* (sz) bound, and candidates with too few non-zeros are ignored. Both APT and L2AP use the same bound in this step[1]. Second, no *new candidates* are accepted if the query prefix does not have enough weight to achieve at least ϵ similarity with an indexed object. Index lists are traversed in inverse feature processing order, and the similarity estimate $\mathrm{sim}(d_c, d_q)$ in line 6 of Algorithm 3 is approximated as the similarity of the query prefix with any indexed object, $\mathrm{sim}(\boldsymbol{d}_q^{\leq j}, \boldsymbol{d}_{<q})$, which we call the *remaining similarity* (rs) bound. In APT, the approximation is based on computing the similarity of the query with the max vector, while L2AP additionally bounds it by the prefix ℓ^2-norm of the query,

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

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

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

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

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

3.3 Candidate verification

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

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

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

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

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

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

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

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

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

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

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

Algorithm 4 Candidate Verification in the AllPairs Framework

1: **function** VERIFYCANDIDATES($\boldsymbol{d}_q, \boldsymbol{c}_q, \mathcal{I}, \epsilon$)
2: $\quad$ **for each** d_c s.t. $c_{q,c} > 0$ **do**
3: $\quad\quad$ *check whether to prune* d_c ▷ ps and dps_1 bounds
4: $\quad\quad$ Find highest j s.t. $d_{c,j}^{\leq} > 0 \wedge d_{q,j} > 0$
5: $\quad\quad$ *check whether to prune* d_c ▷ dps_2 bound
6: $\quad\quad$ **for each** j s.t. $d_{c,j}^{\leq} > 0 \wedge d_{q,j} > 0$ **do**
7: $\quad\quad\quad c_{q,c} \leftarrow c_{q,c} + d_{q,j} \times d_{c,j}$
8: $\quad\quad\quad$ *check whether to prune* d_c ▷ $l2cv$ bound
9: $\quad\quad$ *store similarity* **if** $c_{q,c} \geq \epsilon$

4. PARALLEL ALGORITHMS

In this section, we present two parallel solutions to the APSS problem. First, we summarize algorithmic choices in the method

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

of Awekar and Samatova, pAPT. We then introduce pL2AP, which was designed based on the memory access observations we made in Section 3, with the goal of improving cache locality during similarity search.

4.1 pAPT

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

Algorithm 5 The pAPT Algorithm

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

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

4.2 pL2AP

Our new method, pL2AP, uses the same indexing, candidate generation and verification pruning choices as L2AP. Additionally, pL2AP employs two strategies aimed at improving cache locality during search. First, cache-tiling breaks up the inverted index into blocks that can fit in the system cache, reducing latency during candidate generation. Second, for datasets with high dimensionality, mask-based hash tables can greatly reduce the amount of memory required for storing query object values and meta-data during search, allowing them to persist in the cache during candidate verification. Algorithm 6 provides an overview of our method.

4.2.1 Cache-tiling

Cache-tiling aims to increase cache locality during the candidate generation stage of the similarity search by ensuring the inverted index and accumulator structures fit in cache. To achieve this, the inverted index is split into several consecutive sections, called *tiles*, and each index is used in turn to find neighbors. Choosing the size of each cache tile is non-trivial in the APSS problem, due to the varying number of feature values being indexed for each object. For

Algorithm 6 The pL2AP Algorithm

```
1: function pAPT(D, ϵ, h, ζ, η)
2:     Set processing order for vectors and features
3:     for each q = 1, ..., n in parallel do
4:         S ← FindIndexSplit(d_q, ϵ)
5:     K ← FindIndexAssignments(S, ζ)
6:     O ← ∅, I_{k,j} ← ∅, for j = 1, ..., m and k = 1, ..., K
7:     for each q = 1, ..., n do
8:         Index(d_q, I, S, ϵ)
9:     for each k = 1, ..., K do
10:        for each l = S[k], ..., n, in increments of η do
11:            for each q = l, ..., min(l+η −1, n), in parallel do
12:                c_q ← GenerateCandidates(d_q, I_k, ϵ)
13:                O ← O ∪ VerifyCandidates(d_q, c_q, I_k, ϵ)
14: return O
```

example, choosing to index the same number of objects in each tile will lead to large indexes for the final tiles to be processed which may not fit in cache. Instead, pL2AP first finds the first feature to be indexed in each object (line 4), which also provides the number of values to be indexed in each object. These counts are used to define the consecutive sets of objects to be indexed together in each tile. The list S, containing tile start and end offsets given the predefined processing order, is then used to index each object suffix in their assigned inverted index (line 8).

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

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

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

4.2.2 Query vector mask-hashing

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

Given that query vectors are sparse, and their features are always processed in a predefined order, we developed a heuristic hash-table data structure that uses a small amount of cache space, takes advantage of O(1) access times for most look-ups and leads to few collisions in practice. A small array of size $h + \max(||d_q||_0) - 1$ is used in pL2AP to store matching offsets in one or more lists containing

the query data. Here, $h = 2^{\alpha}$ ($\alpha \geq 0$) is a predefined parameter, generally much smaller than m, and $\max(||d_q||_0)$ is the maximum number of non-zero features for any object. An efficient hashing function maps feature IDs to the $[0, h-1]$ domain, and collisions are entered in the hash-table array in order, starting with index h. Since partial dot-product computations with candidates follow the same traversal order, collisions can be quickly resolved by traversing only a subset of the overflow features. In practice, however, we have found that less than 1% of hash key look-ups end in collision.

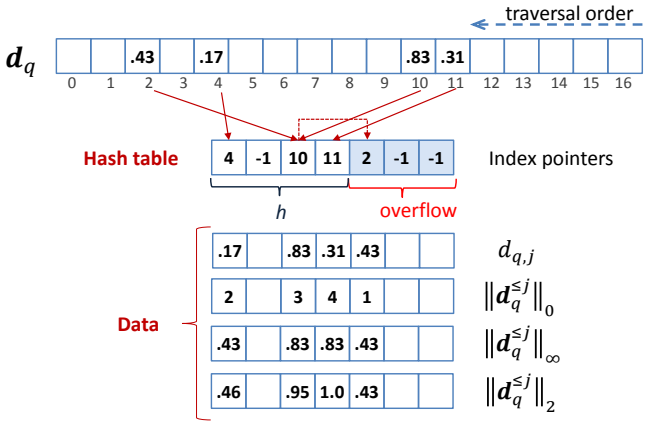

Figure 2: Example query hash table use in `pL2AP`.

Figure 2 provides an example of how a query object might use the hash table in `pL2AP`, for $h = 2^2$. The hash table array is initialized with negative values. Traversing the query non-zeros in reverse feature processing order, the 11th query feature is mapped to the 4th hash table cell, via an efficient truncate operation, $11 \ \& \ (4-1)$, where $\&$ is the bitwise AND logical operator. The feature ID is stored in the hash table at the mapped key index, and one or more value arrays are populated with salient information about the query at the same key index location. `PL2AP` tracks the query prefix value, size, maximum value, and ℓ^2-norm at each index, which are used to check different pruning bounds. In a similar fashion, the 10th query feature is mapped to the 3rd hash table cell, and the 4th query feature to the 1st hash table cell. When mapping the 2nd query feature, the collision is handled by entering the item in the overflow part of the hash table array, in traversal order. When verifying a candidate d_c, its forward index features are traversed in the same order as the query was traversed. Thus, when collisions occur, they can be found by partially traversing the overflow section of the hash table, keeping a pointer to the last cell with a feature ID greater or equal than the sought ID.

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

5. EXPERIMENTAL EVALUATION

In this section, we present our experimental methodology. For both serial and parallel methods, we measure runtime (wallclock), in seconds, for the similarity search phase of the algorithm. 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.

5.1 Datasets

Table 3: Dataset Statistics

Dataset	n	m	nnz	μ_r	σ_r	μ_c	σ_c
RCV1	0.80M	0.05M	62M	76	55	1347	8350
WW500	0.24M	0.66M	202M	830	386	306	3323
WW200	1.02M	0.66M	437M	430	302	659	8273
Wiki	3.71M	3.71M	111M	30	68	56	561
Orkut	3.07M	3.07M	117M	38	131	38	51
Twitter	0.15M	0.15M	200M	1370	2275	1395	2262

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

We use six datasets to evaluate each method. They represent some real-world and benchmark text corpora often used in text-categorization research, and some web/social networks. Their characteristics, including number of objects (n), features (m), and non-zeros (nnz), row/column length mean and standard deviation ($\mu_{r/c}$, $\sigma_{r/c}$), are detailed in Table 3. Standard pre-processing, including tokenization, lemmatization, and *tf-idf* scaling, were used to encode text documents as vectors. Network datasets contain the *tf-idf* scaled binary adjacency structure in the underlying graphs.

- **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. [17].
- **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.
- **Wiki** represents a directed graph of hyperlinks between Wikipedia articles in the Wiki dump.
- **Orkut** contains the friendship network of the Orkut social media site, made available by Mislove et al. [20].
- **Twitter**, first provided by Kwak et al. [15], contains *follow* relationships of a subset of Twitter users that follow at least 1,000 other users.

As can be seen from the mean and standard deviation values listed in Table 3, the datasets we chose are quite varied with respect to their row and column lengths. Figure 3 provides another insight into the non-zero composition in the six datasets, showing the distribution of row (top) and column (bottom) lengths for the six datasets. As both row and column frequency distributions in the datasets follow the power-law, we plot the graphs log-log scaled.

5.2 Baseline methods

In addition to the `pAPT` algorithm by Awekar and Samatova, which we described in Section 4, we compare `pL2AP` against the following algorithms.

1. `IdxJoin`, `APT`, and `L2AP` are baseline serial APSS search methods described in detail in [3]. We report speedup over the fastest execution time of any of the serial methods.

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

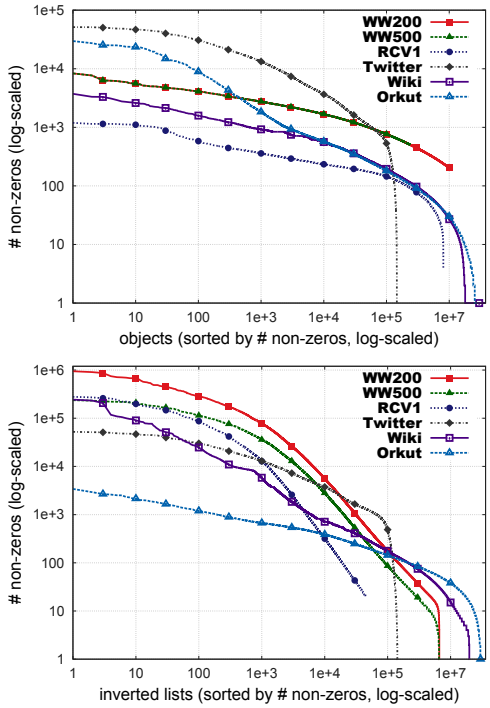

Figure 3: Non-zero distributions in rows / objects (top) and columns / inverted lists (bottom) in our six datasets.

2. `pIdxJoin` uses similar cache-tiling as `pL2AP`, but does not use any pruning when computing similarities. For each block of queries, `pIdxJoin` sequentially retrieves a block of objects to search against and indexes all their values. Threads then share the index to compute similarities, via accumulation, of each assigned query object against all indexed objects, retaining those resulting pairs above the threshold ϵ.

3. $\texttt{pL2AP}_{\texttt{rr}}$ follows the same parallelism strategy as `pAPT` (see Section 4), but takes advantage of the advanced pruning bounds of `L2AP`. After first indexing the suffixes of all objects, $\texttt{pL2AP}_{\texttt{rr}}$ dynamically assigns small sets of query objects for processing to available threads. For each query object, $\texttt{pL2AP}_{\texttt{rr}}$ indexes the same values and performs the same pruning in the candidate generation and verification stages as `pL2AP`.

5.3 Execution environment

Our method and all baselines are implemented in C and compiled using gcc 4.4.7 with -O3 optimization. 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, equiped with 64 Gb RAM and two twelve-core 2.5 GHz Intel Xeon E5-2680v3 processors with 30 Mb Cache. For each method, we varied the similarity threshold ϵ between 0.3 and 0.9, in increments of 0.1. For `pL2AP`, we fixed η at $25K$ objects and varied ζ between $250K$ and $4M$ in $250K$ increments. We set the masked hash-table size parameter h to 2^{13}.

6. RESULTS & DISCUSSION

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

6.1 Effectiveness study

6.1.1 Pruning effectiveness comparison with pAPT

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

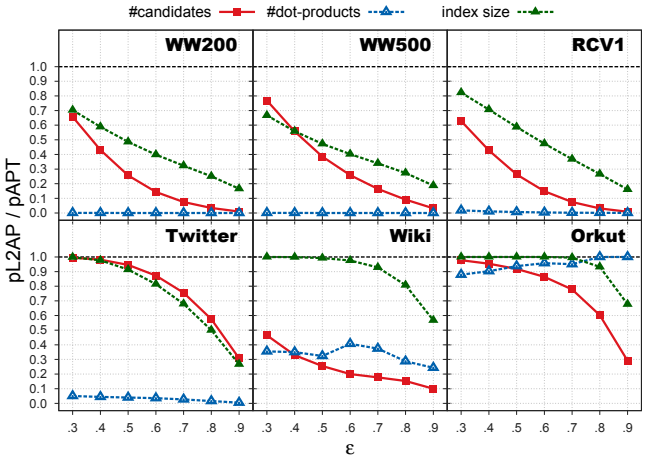

Figure 4: Index size, number of candidates, and number of dot-products in `pL2AP` executions vs. respective values in `pAPT`.

Indexing fewer values can speed up index traversal and thus lead to performance improvements. In addition, it will lead to shorter candidate lists being generated. Considering fewer candidates, as well as more aggressive pruning, can lead to fewer dot-products being computed in full and to better performance. Figure 4 shows the number of indexed non-zeros, candidates, and dot-products when executing `pL2AP`, normalized by the respective values when executing `pAPT`, for ϵ between 0.3 and 0.9, for all six datasets. As compared to `pAPT`, our method generally indexes fewer values, considers fewer candidates, and evaluates fewer complete dot-products, especially at high similarity values. While the difference in the number of indexed values and considered candidates is smaller at $\epsilon = 0.3$, `pL2AP` is able to prune a much higher number of candidates than `pAPT` in all datasets except Orkut, highlighting the improved pruning effectiveness in our method. Orkut is a binary dataset, with short columns that deviate little in length, as shown in Table 3. After our pre-processing, objects in the Orkut set will have fairly uniform small values, which contribute little to accumu-

lation operations and cannot be approximated well. While the size of the un-pruned set of candidates in pL2AP was in most cases between 1x–3x the size of the set of true neighbors, it ranged between 13x–137x for the Orkut dataset. The pAPT method had a similar high number of un-pruned candidates for Orkut, highlighting the inherent similarity estimation difficulty for this dataset.

6.1.2 *Pruning effectiveness in pL2AP*

Our method works by pruning the majority of the candidates that are not true neighbors. Once an object becomes a candidate, it can be pruned by the $l2cg$ bound while accumulating values traversing the inverted index in the candidate generation stage (*cg* in figures), when checking the ps, dps_1 and dps_2 prefix similarity estimate bounds at the onset of the candidate verification stage (*ses* in figures), or by the $l2cv$ bound while accumulating values traversing the forward index in the candidate verification stage (*cv* in figures). Earlier pruning of candidates means less time spent accumulating dot-products in vain and will lead to improved performance. In an experiment in which we used consistent parameters for all datasets ($h = 2^{13}$, $\eta = 25K$, and $\zeta = 1M$), we counted the number of candidates pruned in each stage of the algorithm. We report these values in Figure 5, for all datasets and ϵ values, along with the number of candidates that were not pruned and had their dot-products computed in full (*dps* in figures).

Results show that pL2AP prunes the majority of objects soon after they become candidates, in the candidate generation stage (*cg*). Most of the remaining objects are pruned by the *ses* bounds, which are checked once, at the beginning of the candidate verification stage, and by additional pruning in the candidate verification stage (*cv*). At $\epsilon = 0.3$, for example, only 0.02%–4.89% of candidates survived all pruning across our datasets.

A large number of objects never become candidates in pL2AP, as a result of either the ℓ^2-norm based candidate acceptance bound in the candidate generation stage of the algorithm, or due to the prefix-filtering based index reduction. On average across all ϵ values, only 0.7%–50.4% of all potential candidates actually became candidates for our datasets. Of those, most are pruned quickly, in the first stage of our method. As a way to gauge how quickly candidates are pruned, we measured the number of executed multiply-adds versus the number of possible multiply-adds (percent of accumulated non-zeros) in the similarity computation of each pruned candidate. In Figure 6, we report the mean percent accumulated non-zeros for our six datasets. In each experiment, we used consistent parameters for all datasets (1 thread, $h = 2^{13}$, $\eta = 25K$, and $\zeta = 1M$).

The results show that, for most of the datasets and ϵ values, pL2AP accumulates much less than 10% of the common non-zeros between a query and a non-neighbor candidate on average. Before pruning unsuitable candidates, pL2AP generally accumulates less than 4% of the common non-zeros for text datasets, but it traverses between 10%–60% of the common values for network datasets with short rows, like Wiki and Orkut.

6.1.3 *Cache locality improvements in pL2AP*

While pL2AP performs the same pruning as L2AP, it scans each query object multiple times to compare against objects in multiple constructed inverted indexes. The smaller inverted indexes and the mask-based hash table used during the search help avoid cache thrashing, improving efficiency by reducing time wasted waiting for data transfers from memory to cache. To measure the serial effect of this improvement, we compared the 1-threaded execution of pL2AP against the serial L2AP algorithm. We used $\eta = 25K$ objects and $\zeta = 1M$ non-zeros for this test. Figure 7 shows speedup results

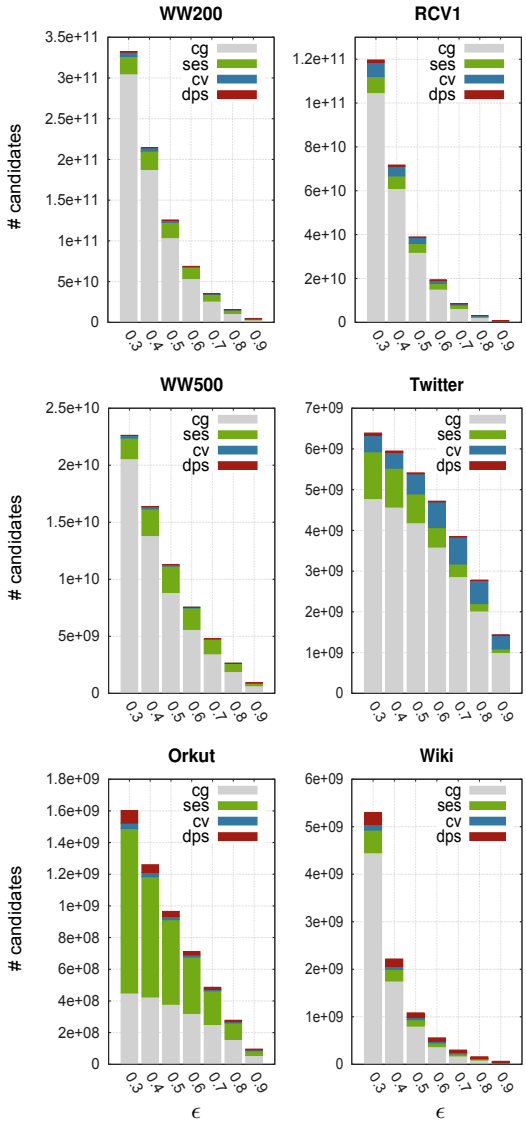

Figure 5: Candidate pruning in pL2AP.

for each of the six datasets we tested, for ϵ between 0.3 and 0.9. The results show an improvement over L2AP for datasets with long inverted lists, whether text or network based. The short inverted lists in the Orkut and Wiki dataset do not provide enough cache reuse for 1 thread to hide the additional work of multiple query searches, leading to slower execution than that of L2AP.

The small inverted index in pL2AP is shared by all threads in executing concurrent searches. As another way to quantify cache locality improvements, we compared the percent of cache misses when executing pL2AP and pL2AP_{rr} with 24 threads. Both algorithms perform the same pruning, but pL2AP_{rr} builds a single inverted index and does not consider cache locality in its execution. We used the *perf* Linux utility to count the number of cache references and cache misses. Figure 8 shows our results when executing pL2AP with ζ between $0.5M$ and $4M$ non-zeros and pL2AP_{rr}, on the RCV1 (top) and Orkut (bottom) datasets, for $\epsilon = 0.3$. We show the size of the inverted index that pL2AP_{rr} builds below its bar in the graph. We observed similar results for most other datasets and ϵ values. In general, pL2AP improves cache locality, and the im-

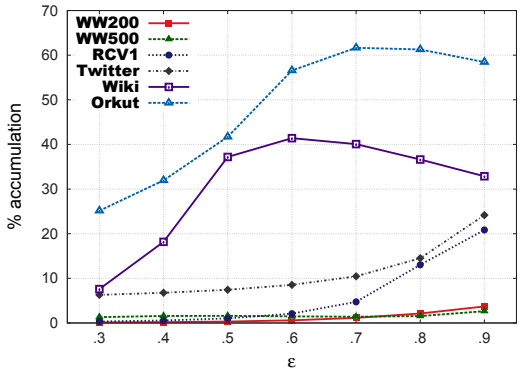

Figure 6: Mean percent accumulated non-zeros before pruning in pL2AP.

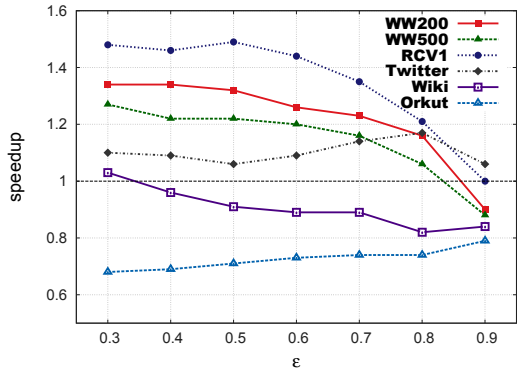

Figure 7: Speedup of 1-threaded pL2AP over L2AP.

provement is more pronounced for text based datasets, which tend to have longer inverted lists.

6.1.4 *Parameter sensitivity*

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

In the first experiment, we set ζ to $1M$ non-zeros and η to $25K$ and varied h between 2^5 and 2^{15}. Results of these experiments over our six datasets are shown on the left side of Figure 9, as execution times relative to the $h = 2^{13}$ parameter choice for each dataset. Our method is not sensitive to this parameter for text and the Twitter datasets, which have smaller dimensionality, but can incur over 2.5x slowdown when choosing a small hash table size for the Orkut or Wiki datasets, which both have over $3M$ dimensions.

The middle section of Figure 9 shows execution times for each dataset, given $h = 2^{13}$ and $\zeta = 1M$, for η between $1K$ and $50K$, relative to the execution time for $\eta = 25K$. We found that choosing the size of each bulk synchronous block, η, does not affect performance in pL2AP, as long as the η value is not too small. We found any values above $5K$ to be adequate for all datasets.

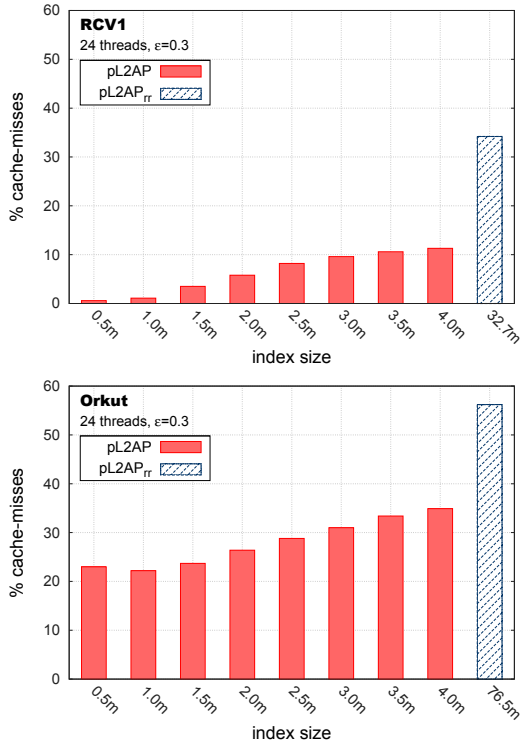

Figure 8: Percent cache misses of pL2AP$_{rr}$ and pL2AP with ζ between 1.5M and 4M non-zeros for the RCV1 (top) and Orkut (bottom) datasets.

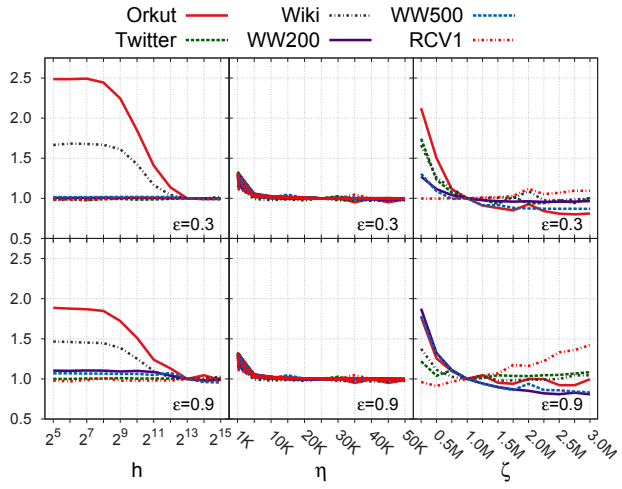

Figure 9: Relative execution times for different h, η, and ζ parameter choices. Figure is best viewed in color.

Finally, we tested the sensitivity of the ζ parameter, for values between $0.25M$ and $3.0M$, given $\eta = 25K$ and $h = 2^{13}$, and show times relative to the $\zeta = 1M$ execution in the right section of Figure 9. While the ζ choice will be dependent on the cache configuration of the target system, our experiments showed that pL2AP performed well for most datasets given ζ set to at least $1M$ non-zeros.

6.2 Efficiency study

6.2.1 Pruning choices in pL2AP

Table 4: Tested pL2AP pruning strategies

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

Pruning is an effective mechanism for reducing the number of similarity computations that must be executed to solve the APSS problem. However, bounds checking incurs additional costs which may not outweigh their benefit. Previous experiments in [3] proved the effectiveness of our ℓ^2-norm based bounds in each stage of the search framework, and showed the sz and dps_2 bounds had little effect in general over the search efficiency. As a way to quantify this effect when executing with multiple concurrent threads, we tested pL2AP in four configuration scenarios, listed in Table 4. The "base" configuration did not effect any pruning based on the sz or dps_2 bounds. The "sz" and "dp" configurations enabled pruning based on the sz and dps_2 bounds, respectively, and the "szdp" configuration enabled pruning based on both the sz and dps_2 bounds. When checking the sz bound, pAPT removes values associated with short vectors from the beginning of inverted lists, which can potentially improve efficiency. We added this capability to pL2AP and tested it in the configuration "szdpupd", which enables all pruning strategies and also performs index updates. Using the same input parameters for all datasets ($nt = 24$, $h = 2^{13}$, $\eta = 25K$ and $\zeta = 1M$), we recorded search execution times under each scenario.

Table 5 reports the results of our experiment. For each ϵ value, times in all configuration scenarios were normalized by that of the sz scenario, and we report the mean, standard deviation (std), minimum and maximum of experiment results across all ϵ values. The best performing results are highlighted in bold. The sz and dp configurations showed little improvement over the base one, at times leading to slower execution times. Checking the sz bound was beneficial in most cases, especially for network datasets, and had better performance than checking the dps_2 bound instead. The combined scenario szdp did not perform better than the sz scenario on average. The results in the remainder of this work assume the sz configuration.

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

6.2.2 Comparison with serial methods

We compared the execution time of all parallel methods, executed with 24 threads, with the best serial execution time achieved by any of the serial algorithms. Figure 10 shows the results of this experiment. In all cases, pL2AP had the best execution time of all parallel methods, achieving speedups of 2x–20x for network datasets and 12x–34x for text datasets. Compared to existing parallel baselines, pL2AP executed 1.5x–3x faster for network datasets and 7x–238x faster for text datasets. While $\texttt{pL2AP}_{\texttt{rr}}$ uses the same type of pruning as pL2AP, it traverses the entire inverted index during each query and, as a result, cannot perform as well. Instead, by using tiling and other optimizations that promote cache locality, pL2AP is able to achieve very good speedup for datasets with long inverted index lists, such as text datasets. At high similarity thresholds, however, pL2AP is able to prune candidates quickly and does not need to traverse many candidate and query vector features, rendering our cache locality optimizations less effective.

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

Table 5: Performance of different pruning choice configurations in pL2AP.

versus	mean	stdv	min	max
	Orkut			
nbase	1.0642	0.0237	1.0234	1.0929
dp6	1.1017	0.0249	1.0684	1.1377
sz	**1.0000**	0.0000	1.0000	1.0000
szdp	1.0443	0.0082	1.0319	1.0606
szdpupd	1.4624	0.0714	1.3222	1.5298
	Twitter			
nbase	1.0191	0.0127	0.9974	1.0345
dp6	1.0416	0.0230	0.9869	1.0581
sz	**1.0000**	0.0000	1.0000	1.0000
szdp	1.0416	0.0219	1.0156	1.0736
szdpupd	1.0671	0.0273	1.0345	1.1279
	Wiki			
nbase	1.0373	0.0109	1.0190	1.0540
dp6	1.0540	0.0118	1.0305	1.0675
sz	**1.0000**	0.0000	1.0000	1.0000
szdp	1.0128	0.0051	1.0076	1.0243
szdpupd	1.2326	0.0489	1.1484	1.3092
	WW200			
nbase	**0.9933**	0.0163	0.9719	1.0240
dp6	1.0034	0.0112	0.9948	1.0296
sz	1.0000	0.0000	1.0000	1.0000
szdp	1.0152	0.0119	1.0047	1.0369
szdpupd	1.0661	0.0481	1.0139	1.1586
	WW500			
nbase	**0.9980**	0.0109	0.9894	1.0234
dp6	1.0115	0.0178	0.9975	1.0540
sz	1.0000	0.0000	1.0000	1.0000
szdp	1.0249	0.0212	1.0097	1.0744
szdpupd	1.0442	0.0354	1.0145	1.1241
	RCV1			
nbase	1.0017	0.0051	0.9909	1.0086
dp6	1.0090	0.0061	1.0027	1.0215
sz	**1.0000**	0.0000	1.0000	1.0000
szdp	1.0104	0.0055	1.0029	1.0212
szdpupd	1.0240	0.0137	1.0108	1.0515

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

6.2.3 Strong scaling

Figure 11 shows the strong scaling results from our experiments. The amount of work pL2AP does when processing each query increases as the threshold ϵ decreases. At high values of ϵ, many of the objects never become candidates for a query due to the idx and

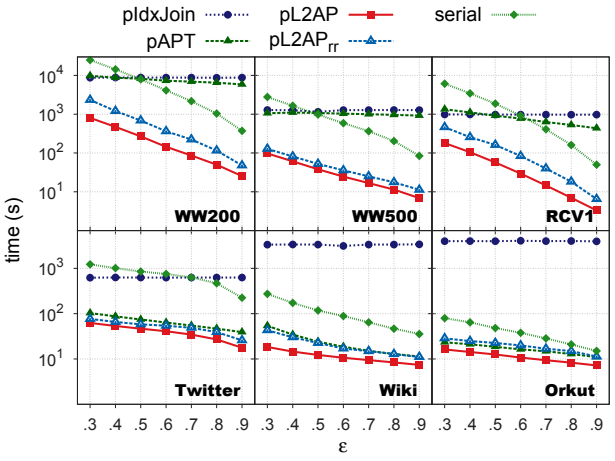

Figure 10: Execution times of parallel methods and the best serial alternative.

rs bounds in our method, and pL2AP is able to quickly dismiss candidates. For example, the size of the candidate list when $\epsilon = 0.9$ is 0.5–4.0% of the candidate list size when $\epsilon = 0.3$ for text datasets. As a result, the cache locality improvements in pL2AP are not as beneficial, resulting in less pronounced scaling at $\epsilon = 0.9$. On the other hand, pL2AP shows linear scaling at $\epsilon = 0.3$ for text datasets. While its scaling is not as dramatic for network datasets, pL2AP still exhibits very strong scaling, in most cases better than the other baselines.

It is interesting to note that pAPT and pL2AP$_{rr}$ both scale poorly above twelve threads on text datasets. This may be an indication of thrashing, which is causing threads to waste time waiting for cache lines to be fetched from main memory.

6.2.4 *Load Balance*

In order to test the effectiveness of the dynamic task partitioning approach in pL2AP, we measured the amount of time each thread spent searching for neighbors. Figure 12 shows the percent load imbalance averaged over all ϵ values, in experiments with consistent parameters ($nt = 24$, $h = 2^{13}$, $\eta = 25K$, and $\zeta = 1M$), for the six datasets. Load imbalance is computed as $(t_{max}/t_{mean} - 1) \times 100$, where t_{max} and t_{mean} are the maximum and mean search times among the threads. Our method shows little imbalance between the threads, much less than 1% for text datasets and less than 2% for network datasets.

7. RELATED WORK

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

Existing distributed solutions to the problem generally use the MapReduce [11] framework and can be split into two categories. Most rely on the framework's built-in features to aggregate (reduce) partial similarities of object pairs computed in mappers [6, 10, 12, 18]. The computation efficiency can be greatly increased by first generating an inverted index for the set of objects, which can be done using one MapReduce task. The postings in the inverted index lists can then be combined with features in the object vectors or with other postings in the same list to generate partial similarity scores. While some pruning strategies can be used to avoid generating some partial scores, these methods often suffer from high communication costs which make then inefficient for large datasets [2].

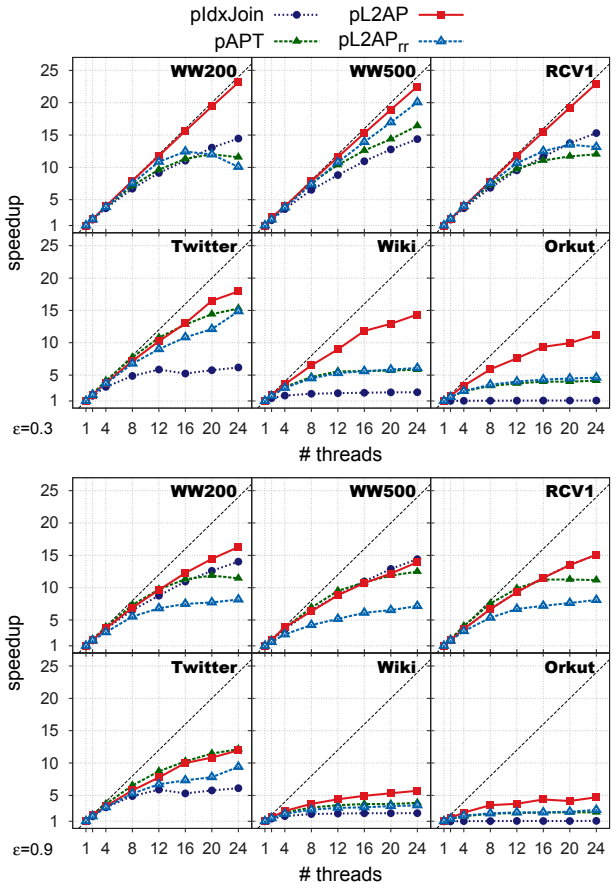

Figure 11: Strong scaling of parallel methods at $\epsilon = 0.3$ (top) and $\epsilon = 0.9$ (bottom).

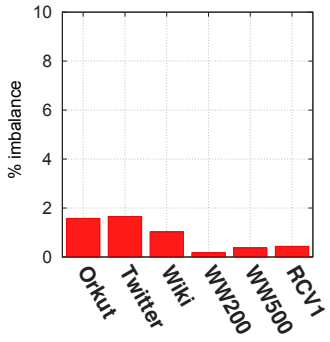

Figure 12: Load imbalance in pL2AP.

The second category of MapReduce methods use a mapper-only scheme, with no reducers [1, 2, 22]. They partition the set of objects into subsets (blocks) and use serial APSS methods to find pairwise similarities of objects in block pairs. Certain block comparisons can be eliminated by relying on block-level filtering techniques, such as computing the similarity of the objects made up of the maximum values for features in the two blocks. When comparing two blocks, Alabduljalil et al. proposed locally building a full inverted index for one of the blocks and scanning through query objects in the other block to compute their similarity. They found that filtering candidates was detrimental to execution speed and suggested removing this optimization, rendering their local search identical to that performed in one tile by our naïve baseline, pIdxJoin. Within this context, they examined distributed load bal-

ancing strategies [22] and cache-conscious performance optimizations for the local searches [1]. They provided a cost based analysis aimed at finding sizes for comparison blocks that maximize cache locality. Their analysis is based on a full inverted index and mean vector and inverted list lengths, which can vary greatly in real datasets, as evidenced by the high σ values in Table 3.

Existing multi-core cosine APSS solutions are limited to the `pAPT` algorithm by Awekar and Samatova, detailed in Section 4. Jiang et al. [14] provided a parallel solution for the related problem of string similarity joins with edit distance constraints.

8. CONCLUSIONS AND FUTURE WORK

We presented `pL2AP`, our multi-core parallel solution to the All-Pairs cosine similarity search problem. Our method uses several cache-tiling optimizations, combined with fine-grained dynamically balanced parallel tasks, to solve the problem, using 24 threads, 1.5x–232x faster than existing parallel baselines and 2x–34x faster than the fastest serial method on datasets with hundreds of millions of non-zeros. In the current work we have focused on tiles that fit in L3 cache. It would be interesting to evaluate strategies for maximizing the reuse of the L1 and L2 caches in similarity search. Additionally, while choosing a cache-tile size for `pL2AP` is fairly straight-forward, we may investigate designing a cache-oblivious parallel APSS method. Finally, we may explore distributed algorithms for efficiently constructing cosine similarity graphs.

Acknowledgment

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

9. REFERENCES

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

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

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

[15] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[16] Dongjoo Lee, Jaehui Park, Junho Shim, and Sang-goo Lee. An efficient similarity join algorithm with cosine similarity predicate. In *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part II*, DEXA'10, pages 422–436, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] 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.

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

[19] 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.

[20] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proc. Internet Measurement Conf.*, 2007.

[21] 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.

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

[23] 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.