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–238x 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 t. 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 t. 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
IA^3 2015, November 15-20, 2015, Austin, TX, USA.
© 2015 ACM. ISBN 978-1-4503-4001-4/15/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/2833179.2833182.

Parallel APSS solutions target either multi-core or distributed systems. Most distributed approaches use MapReduce [11] to compute similarities in parallel [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–238x 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 $\text{sim}(d_i, d_j) \geq t$. We use the *cosine function* to measure vector similarity. To simplify the presentation 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 $\text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_j)$.

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 $\langle d_{i,1}, \ldots, d_{i,p}, 0, \ldots, 0\rangle$, obtained by keeping the p leading dimensions in $\boldsymbol{d}_i$, which we call the *prefix* (vector) of $\boldsymbol{d}_i$. Similarly, we refer to $\boldsymbol{d}_i^{>p} = \langle 0, \ldots, 0, d_{i,p+1}, \ldots, d_{i,m}\rangle$ as the *suffix* of $\boldsymbol{d}_i$, obtained by setting the first p dimensions of $\boldsymbol{d}_i$ to 0. One can then verify that

$$\boldsymbol{d}_i = \boldsymbol{d}_i^{\leq p} + \boldsymbol{d}_i^{>p},$$

$$||\boldsymbol{d}_i||^2 = ||\boldsymbol{d}_i^{\leq p}||^2 + ||\boldsymbol{d}_i^{>p}||^2, \text{ and}$$

$$\text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_j) = \text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_j^{\leq p}) + \text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_j^{>p}).$$

3. SERIAL ALGORITHMS

Most serial APSS solutions follow a similar computation framework, first introduced by Bayardo et al. [7], which we describe in Algorithm 1. In this section, we present an overview of the framework, with a focus on its memory access patterns. Specific pruning theoretic bounds included 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, are described in detail in [3].

Algorithm 1 The AllPairs Framework

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

The APSS search proceeds in three stages. While processing an object d_i, which we call the *query*, a list of potential *candidates* is generated, which is a superset of the set of objects similar to the query. A partial similarity score is computed for each candidate and used to *prune* (remove from further consideration) many of those candidates that cannot meet the similarity threshold requirement. Then, in the second stage, each candidate is verified, eliminating the remaining dissimilar objects from the list and finalizing the similarity value computation for the similar objects. Finally, the query object is analyzed and some of its suffix features and other metadata are added to a growing *inverted index*, which is used to generate the candidate list for the next object in the processing order. Processing objects in this manner takes advantage of the commutative property of cosine similarity, computing only one similarity score for each pair of objects.

Since parts of 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 t (line 3 in Algorithm 2). Any unprocessed *similar* object is guaranteed in this way to have at least one feature in common with the query object. Then, when later processing that *similar* object, the query object will be found while traversing the index in the candidate generation stage. `APT` computes the estimate $\hat{\text{sim}}(\boldsymbol{d}_i^{\leq j+1}, \cdot)$ as the dot product between the query vector and the vector made up of all maximum feature values, which we call the *max vector*. The estimate is further improved by requiring that objects be processed in decreasing order of their maximum feature weights. Furthermore, `L2AP` uses the ℓ^2-norm of the query prefix ending at index j, inclusive, $||d_i^{\leq j+1}||$, as an estimate of the query object similarity with any other object, which includes unprocessed objects. When indexing suffix values of the query vector (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.

Algorithm 2 Indexing in the AllPairs Framework

1: **function** INDEX($\boldsymbol{d}_i, \mathcal{I}, t$)
2: **for each** $j = 1, \ldots, m$, s.t. $d_{i,j} > 0$ **do**
3: **if** $\hat{\text{sim}}(\boldsymbol{d}_i^{\leq j+1}, \cdot) \geq t$ **then**
4: $I_j \leftarrow I_j \cup \{(d_i, d_{i,j})\}$ ▷ add suffix to index

Indexing requires traversing the sparse query vector and accessing values in the max vector, which could be stored in 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 t 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 t. Higher thresholds allow delaying indexing further and lead to a smaller inverted index, as shown in Figure 1 of [3].

The candidate generation and verification stages are described in Algorithms 3 and 4, respectively. During candidate generation, the lists in the current version of the inverted index associated with non-zero feature values in the query object are scanned, one list at a time. An accumulator (map based data structure that accumulates values 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*. However, accumulation is prevented for a new object if its prefix does not have enough weight to achieve at least t similarity with the query. The similarity estimate $\hat{\text{sim}}(d_c, d_i)$ in line 6 of Algorithm 3 is based on computing the similarity of the query with the max vector in `APT`, and additionally based on the prefix ℓ^2-norm of the query in `L2AP`. Vectors that do not have enough non-zeros given their maximum value are also pruned by `APT` and removed from the index, which avoids checking them in future index traversals. While this pruning strategy has little effect for some datasets (e.g., text-based ones), it can improve efficiency for datasets with many objects with few non-zero values. After each accumulator change, at features the query and candidate objects have in common, `L2AP` also checks whether the candidate should be pruned, based on the prefix ℓ^2-norm of the query and candidate vectors. While `L2AP` stores the prefix ℓ^2-norm of the candidate in the inverted index along with its feature values, it computes the ℓ^2-norm of the query vector on-the-fly while iterating through its values.

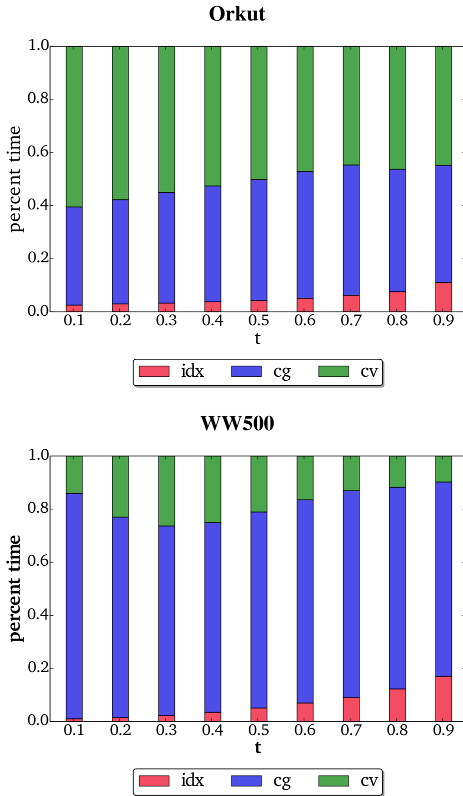

Figure 1: Percent execution times for the Orkut and WW500 datasets. For each dataset, the stacked bars show the percent of search time taken by the indexing (*idx*), candidate generation (*cg*), and candidate verification (*cv*) phases in L2AP, for similarity thresholds ranging from 0.1 to 0.9.

Algorithm 3 Candidate Generation in the AllPairs Framework

```
1: function GENERATECANDIDATES(d_i, I, t)
2:     A ← ∅                                          ▷ accumulator array
3:     for each j = 1, ..., m, s.t. d_{i,j} > 0 do
4:         for each (d_c, d_{c,j}) ∈ I_j do
5:             check whether to remove d_c from index
6:             if d_c is not pruned and
                 sîm(d_c, d_i) can be at least t then
7:                 A[d_c] ← A[d_c] + d_{i,j} × d_{c,j}
8:                 check whether to prune d_c           ▷ L2AP only
9: return A
```

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 lists in the inverted index. If these structures take up more than the available cache memory, the computation will incur additional delay while data is loaded from the main memory.

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 partial similarity (line 5 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. As in the candidate generation stage, after each similarity accumulation, L2AP checks whether the candidate should be pruned, based on the prefix ℓ^2-norm of the query and candidate vectors, or based on the maximum feature value in the candidate prefix.

Algorithm 4 Candidate Verification in the AllPairs Framework

```
1: function VERIFYCANDIDATES(d_i, A, I, t)
2:     for each d_c s.t. A[d_c] > 0 do
3:         check whether to prune d_c
4:         for each j s.t. d^>_{c,j} > 0 ∧ d_{i,j} > 0 do
5:             A[d_c] ← A[d_c] + d_{i,j} × d_{c,j}
6:             check whether to prune d_c               ▷ L2AP only
7:         store similarity if A[d_c] ≥ t
```

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.

4. PARALLEL ALGORITHMS

In this section, we present two parallel solutions to the APSS problem. First, we summarize algorithmic choices in the method of Awekar and Samatova, pAPT. We then introduce pL2AP, which was designed based on the memory access observations we made in Section 3, 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. Their main idea was to pre-compute the partial inverted index (lines 4–5 in Algorithm 5), rather than indexing each object after its processing, and allow threads to share the index structure. To prevent synchronization overheads when removing values 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, t)
2:     Set processing order for vectors and/or features
3:     O ← ∅, I_j ← ∅, for j = 1, ..., m
4:     for each i = 1, ..., n do
5:         Index(d_i, I, t)
6:     for each i = 1, ..., n, in parallel do
7:         A ← GenerateCandidates(d_i, I, t)
8:         O ← O ∪ VerifyCandidates(d_i, A, I, t)
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, which has been shown to outperform those in APT, both theoretically and experimentally [3]. 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.

Algorithm 6 The pL2AP Algorithm

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

4.2.1 Cache-tiling

Cache-tiling aims to increase cache locality during the candidate generation stage of the similarity search by ensuring the inverted index and accumulator structures fit in cache. The inverted index in pL2AP is highly dependent on the values of objects being indexed and the required minimum similarity threshold t. As such, 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 *tiles*, consecutive sets of objects to be indexed together. 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.

Choosing the size of each cache tile is non-trivial in the APSS problem, due to the varying number of feature values being indexed for each object. For example, choosing to index the same number of objects in each tile will lead to large indexes for the final tiles to be processed which may not fit in cache. Instead, we assign up to inz non-zero values to be indexed in each tile, where inz is an input parameter in our method.

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 qsz at a time, where qsz is an input parameter, forcing threads to read from the same subset of query vectors, which should be located in sequential memory blocks.

4.2.2 Query vector mask-hashing

During candidate verification, pL2AP sequentially traverses the prefix of a candidate and checks whether the query has non-zero values for the features encountered. When those are encountered, 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 by storing 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 hashing technique that uses a small amount of cache space for each hash structure, takes advantage of O(1) access times for most look-ups and leads to few collisions in practice. pL2AP uses a small *hash-table* array of size $h + \max(||d_i||_\infty)$ to store matching offsets in one or more lists containing the query data. Here, h is a predefined parameter, generally much smaller than m, and $\max(||d_i||_\infty)$ is the maximum number of non-zero features for all objects in the set. The hash-table array is initialized with negative values. For each feature in the query vector, pL2AP efficiently computes a hash key by using the mask $(1 \ll h)$ to truncate the feature ID, where the ID is its position in the predefined global feature processing order, to the $[0, h-1]$ domain. Offsets into the sparse query vector are stored at locations in the hash-table corresponding to feature hash keys. In case of a collision, the offset for the feature is added to the end of the hash-table array, starting at index h, in processing order, and can thus be quickly looked up through a limited linear scan. In practice, however, we have found that less than 1% of hash key look-ups end in collision.

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.

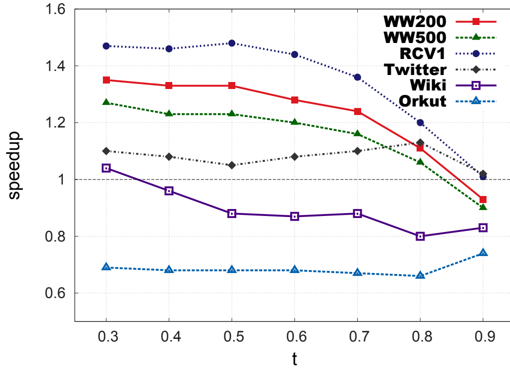

Figure 2: Speedup of 1-threaded `pL2AP` over `L2AP`.

5.1 Datasets

Table 1: 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 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 1. 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[1] (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.

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]http://download.wikimedia.org

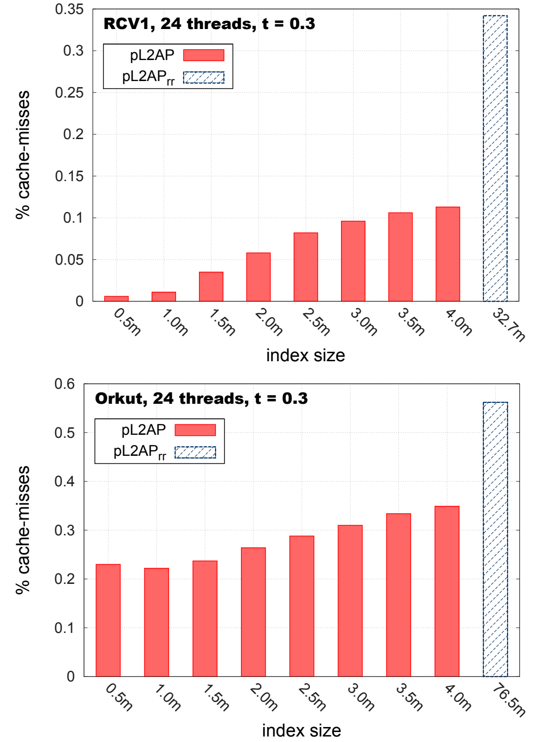

Figure 3: Percent cache misses of $\texttt{pL2AP}_{\texttt{rr}}$ and `pL2AP` with inz between 1.5M and 4M non-zeros for the RCV1 (top) and Orkut (bottom) datasets.

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. `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 pairs above the threshold t.
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 with two twelve-core 2.5 GHz Intel Xeon E5-2680v3 processors and 64 Gb RAM. For each method, we varied the similarity threshold t between 0.3 and 0.9, in increments of 0.1. For `pL2AP`, we fixed qsz at 25K objects and varied inz between 250K and 3M in 250K increments. We set the masked hash-table size parameter h to 2^{13}.

6. RESULTS & DISCUSSION

6.1 Improvements in cache locality

In this section, we show the effectiveness of our proposed techniques to improve cache locality. 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 qsz =25K objects and inz =1M non-zeros for this test. Figure 2 shows speedup results for each of the six datasets we tested, for t 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. However, the small inverted index in pL2AP is shared by all threads in executing concurrent searches. As another way to quantify cache locality improvements, we compared the percent of cache misses when executing pL2AP and $\texttt{pL2AP}_{\texttt{rr}}$ with 24 threads. Both algorithms perform the same pruning, but $\texttt{pL2AP}_{\texttt{rr}}$ builds a single inverted index and does not consider cache locality in its execution. Figure 3 shows our results when executing pL2AP with inz between 0.5M and 4M non-zeros and $\texttt{pL2AP}_{\texttt{rr}}$, on the RCV1 (top) and Orkut (bottom) datasets, for $t = 0.3$. We show the size of the inverted index $\texttt{pL2AP}_{\texttt{rr}}$ builds below its bar in the graph. We observed similar results for most other datasets and t values. In general, pL2AP improves cache locality, and the improvement is more pronounced for text based datasets, which tend to have longer inverted lists.

6.2 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 inz 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 inz 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 inz 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 inz to 1M non-zeros and qsz 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 4, 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.

Choosing the size of each bulk synchronous block, qsz, does not affect performance in pL2AP, as long as the qsz value is not too small. We found any values above 5K to be adequate for all datasets. The middle section of Figure 4 shows execution times for each dataset, given $h = 2^{13}$ and inz =1M, for qsz between 1K and 50K, relative to the execution time for qsz =25K.

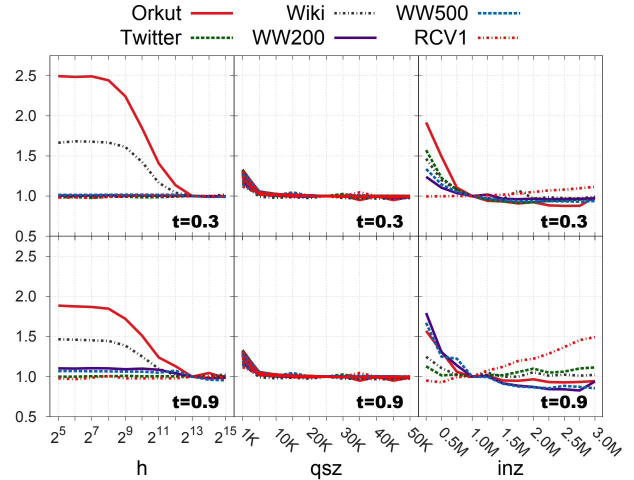

Figure 4: Relative execution times for different h, qsz, and inz parameter choices.

Finally, we tested the sensitivity of the inz parameter, for values between 0.25M and 3.0M, given qsz =25K and $h = 2^{13}$, and show times relative to the inz =1M execution in the right section of Figure 4. While the inz choice will be dependent on the cache configuration of the target system, our experiments showed that pL2AP performed well for most datasets given inz set to at least 1M non-zeros.

6.3 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, at similarity thresholds between 0.3 and 0.9. Figure 5 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 columns, such as text datasets. At high similarity thresholds, 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 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.

6.4 Strong scaling

Figure 6 shows some of the strong scaling results from our experiments. The amount of work pL2AP does when processing each query increases as the threshold t decreases. At high values of t, there are few similar objects for each query and pL2AP is able to

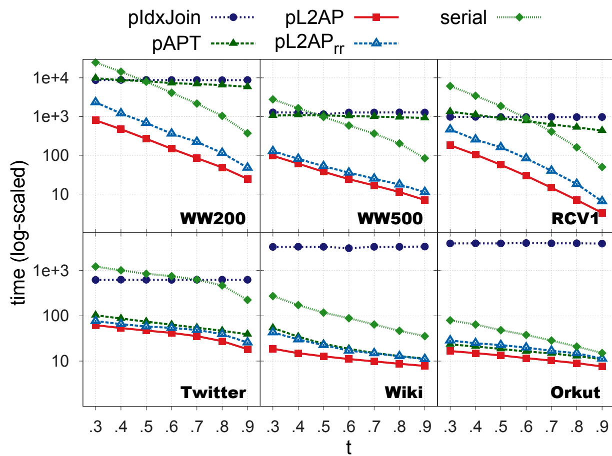

Figure 5: Execution times of parallel methods and the best serial alternative.

quickly dismiss candidates. As a result, its scaling is not as pronounced at $t = 0.9$. On the other hand, queries will have many similar objects at $t = 0.3$, and `pL2AP` shows linear scaling at this threshold 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.

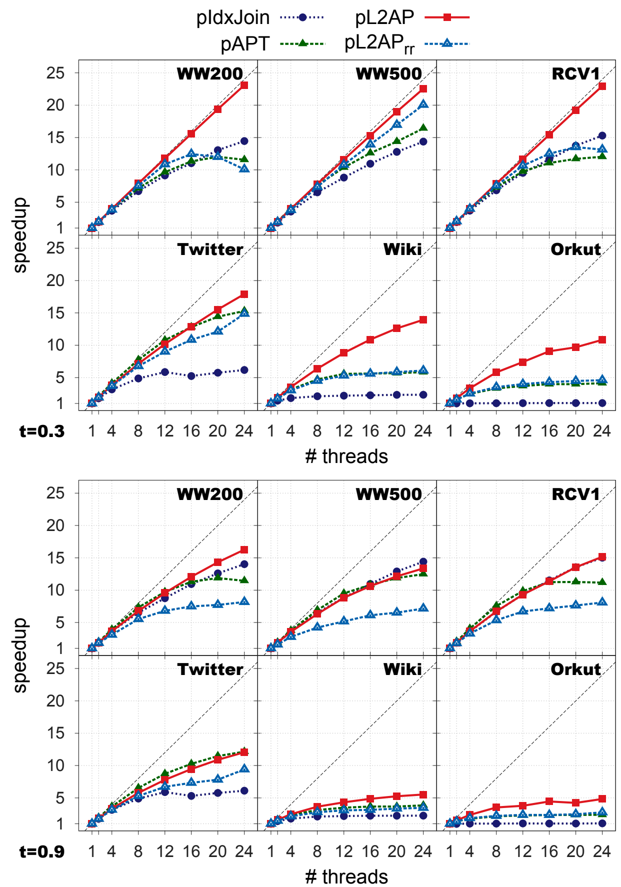

Figure 6: Strong scaling of parallel methods at $t = 0.3$ (top) and $t = 0.9$ (bottom).

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].

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 balancing 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 1.

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–238x 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. 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.